Frontiers For High-Level Synthesis of Digital Circuits

A Thesis Submitted to the College of
Graduate Studies and Research
in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan

by

Lakshmikanth Ghatraju

April 1997

© Copyright Lakshmikanth Ghatraju, 1997. All rights reserved.
Permission To Use

In presenting this thesis in partial fulfillment of the requirements for a Post-graduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 0W0
Abstract

This thesis presents a novel approach to the synthesis of combinational and sequential digital circuits from recursive behavioral specifications based on the functional programming paradigm. Recursion removal is accomplished by computing the fixpoints of recursion equations. The techniques used for computing the fixpoints are based on domain theory and frontiers algorithms. Two-level combinational logic synthesis is a direct consequence of the application of the frontiers algorithm, which derives a correct sum-of-products expression. Multi-level logic synthesis is based on the iterative approach used for computing the fixpoints. Algorithms for common sub-expression identification and derivation of factored forms are presented. These algorithms rely on the frontiers approach, further strengthening the choice of a semantics-based approach. Sequential circuits are modeled as finite state machines. Sequential circuit synthesis uses the techniques established for combinational logic synthesis to derive the computational elements needed to realize the corresponding finite state machine.

A significant advantage of the approach used here is that the structure of recursion in the original specification is immaterial for hardware synthesis, i.e. any recursive function can be synthesized without the use of a stack. Since the techniques used are semantics-based, synthesis from two denotationally equivalent specifications always results in the same two-level hardware.
Acknowledgements

I am greatly indebted to my advisor and mentor, Prof. Carl McCrosky, for the supervision of this thesis, and for introducing the concept of frontiers to me. Carl has been a constant source of inspiration, encouragement, and interesting ideas. His attention to the detail and astute suggestions have vastly influenced the quality of presentation of this thesis. It has been an immensely enjoyable and rewarding experience to work with Carl.

I would like to extend my sincere gratitude to Prof. Mostafa Abd-El-Barr, my co-supervisor, and the other members of my thesis committee, Prof. Ron Bolton, Prof. Eric Neufeld and Dr. Ken Sailor for their contributions. Many thanks to Ken Sailor for a thorough reading of this thesis and a number of interesting discussions we had on this topic that helped me to clarify the concepts presented here.

A great many of people have contributed to this work in a variety of ways. In particular, my fellow students at the Department of Computer Science with a stimulating work environment, my co-workers at Northern Telecom and Sun Microsystems for making me realize my priorities, and a number of friends and families in Saskatoon who made my stay here memorable. I am especially thankful to Dr. & Mrs. Krishnamurthy for their gracious hospitality during my stay at Saskatoon, and Venkat Josyula for handling all the administrative details at the university. Special thanks are also due to my wife, Jeyanthi, for sacrificing a number of holidays to get this thesis to completion and taking care of our new born son, Pranav, during the final preparations of this thesis, and to my parents without whose love and support would have never made this possible.

Financial support for this research work in the form of Graduate Scholarship from the College of Graduate Studies and Research at the University of Saskatchewan is gratefully acknowledged.
# Table Of Contents

Permission To Use .............................................. i

Abstract .......................................................... ii

Acknowledgements .................................................. iii

Table Of Contents ................................................... iv

List of Tables ........................................................ v

List of Figures ....................................................... vi

1 Introduction ....................................................... 1
  1.1 Motivation and Objectives .................................... 2
  1.2 Overview ..................................................... 3
  1.3 Outline ....................................................... 5

2 High-Level Synthesis — An Overview ......................... 8
  2.1 Hardware Description Languages ................................. 9
    2.1.1 Structural Specification ................................ 9
    2.1.2 Behavioral Specification ................................. 11
  2.2 Synthesis from Behavioral Specifications .................. 14
    2.2.1 Compilation ................................................ 15
    2.2.2 General Purpose Transformations ....................... 16
    2.2.3 Architecture Specific Transformations .................. 17
    2.2.4 Scheduling ................................................ 18
4.3 A Solution – Frontiers Approach .............................................. 52
  4.3.1 Motivation ................................................................. 53
  4.3.2 Frontiers ................................................................. 54
  4.3.3 Frontiers Algorithm ....................................................... 56
4.4 Concluding Remarks ............................................................ 58

5 Specification Languages .......................................................... 59
  5.1 Introduction to FHDL ......................................................... 60
    5.1.1 Formal Semantics of FHDL ........................................... 64
  5.2 The Kernel Language ....................................................... 66
    5.2.1 Formal Semantics of KL ............................................. 70
  5.3 Translation Scheme .......................................................... 72
  5.4 Concluding Remarks ........................................................ 77

6 Foundations for Synthesis ......................................................... 78
  6.1 Boolean Function Domain ................................................... 79
    6.1.1 Background ............................................................ 79
    6.1.2 Boolean Algebra ...................................................... 80
    6.1.3 Boolean Functions .................................................... 82
  6.2 Domain Inflation ............................................................ 83
    6.2.1 Frontiers to the Rescue .............................................. 85
    6.2.2 An Example ............................................................ 87
  6.3 Computing Frontiers ........................................................ 88
    6.3.1 Algorithm for Computing Frontiers ................................ 88
  6.4 Concluding Remarks ........................................................ 93

7 Circuit Synthesis ................................................................. 94
  7.1 Overview of Synthesis Process .......................................... 95
  7.2 Advantages of A Semantics Based Approach ............................ 96
  7.3 Interface to the Layout Tools ............................................ 97
    7.3.1 PLA Generation ....................................................... 97
7.4 VHDL Code Generation ........................................... 102
7.5 Runtime Performance of Algorithms ......................... 107
7.6 Concluding Remarks ............................................ 108

8 Multi-level Logic Synthesis ...................................... 109
  8.1 Multi-level Logic Synthesis .................................. 110
  8.2 Frontiers for Multi-level Synthesis ......................... 112
    8.2.1 Synthesis of Nested Forms .............................. 112
    8.2.2 Factoring ................................................. 118
  8.3 Concluding Remarks ........................................... 125

9 Synthesis of Sequential Circuits ................................ 127
  9.1 Model of Behavior ............................................. 128
    9.1.1 List Notation ............................................ 129
    9.1.2 Extension to the Formal Semantics of FHDL .......... 130
  9.2 Model of Finite-State Machines .............................. 131
  9.3 Semantics of Sequential Machines ......................... 133
  9.4 Synthesis of Function Definitions .......................... 134
    9.4.1 Synthesis of Output functions ......................... 134
    9.4.2 Synthesis of Next-State Functions .................... 137
    9.4.3 Handling Initial State .................................. 140
  9.5 Extensions to Sequential Circuits .......................... 141
  9.6 Synthesis of Sequential Circuits ........................... 143
    9.6.1 An Example .............................................. 144
  9.7 Concluding Remarks ........................................... 147

10 Conclusions .................................................... 148
  10.1 Review ....................................................... 148
    10.1.1 Specification Language ................................ 148
    10.1.2 Recursion Removal and Fixpoints .................... 150
    10.1.3 Frontiers Algorithms and Hardware Synthesis ....... 152
10.1.4 Back-end Tool Support ......................... 154
10.1.5 Circuit Optimizations .......................... 154
10.1.6 Sequential Circuits ............................. 155

10.2 Prospects for Research ............................ 156
10.2.1 Communication and Asynchronous Circuits ........... 156
10.2.2 Partitioning of Control .......................... 157
10.2.3 Other Topics ..................................... 158

References .............................................. 160

A VHDL Specification of Traffic Light Controller ............. 170
List of Tables

7.1 Run-time performance of the algorithms. ....................... 107

8.1 Common sub-expressions for the example. ....................... 116

8.2 Summary of results obtained after factorization. .............. 124
List of Figures

1.1 Overview of the synthesis tool. ........................................ 5
2.1 A 4-to-1 multiplexor. .................................................. 10
2.2 VHDL description of multiplexor. ................................... 11
2.3 CIRCAL description of inverter. ....................................... 12
2.4 State transition diagram for inverter. ............................... 13
2.5 A behavioral specification fragment in VHDL .................... 13
2.6 Different forms of internal representation. ....................... 16
2.7 An inverter. ............................................................. 23
3.1 Binary search algorithm in Miranda ............................... 30
3.2 Triangle function. ..................................................... 34
3.3 Geometrical interpretation of composition function. .......... 35
3.4 Description of a full adder in vFP. ................................ 35
3.5 Functional description of greatest common divisor problem. 37
3.6 Two versions of factorial function. (a) Recursive. (b) Iterative. 38
3.7 Circuit realization for the greatest common divisor problem. 42
3.8 Factorial function definition ........................................ 45
3.9 Inverted factorial function ........................................... 46
4.1 Lattice of values in $T^2$. ............................................. 55
4.2 Computation of an irredundant frontier set. ..................... 56
4.3 0-frontiers for fac#. .................................................. 57
5.1 Primitive functions of FHDL. ......................................... 62
5.2 Two equivalent definitions of factorial function in FHDL. ... 62
5.3 Typical definitions in FHDL ........................................ 64
5.4 Geometrical interpretation of transformation on primitive functions. . 68
5.5 An example of KL library and a translated definition. .................. 69
5.6 Definition to *sum* during the translation process. .................... 75
5.7 Result of TW translation scheme on *sum*. .......................... 75

6.1 Lattice of values in $T^3$ ........................................... 80
6.2 Lattice of values for *xnor* ......................................... 84
6.3 Inflated lattice of *xnor* ........................................... 85
6.4 Lattice of values for the function $f = ab + \overline{ab}$ ............... 87

7.1 Steps in the the synthesis process. .................................. 95
7.2 Two denotationally equivalent specifications. .......................... 96
7.3 Programmable logic array structure. .................................. 98
7.4 NMOS PLA code for 2-bit greatest common divisor problem. ........ 99
7.5 CMOS PLA code for 2-bit greatest common divisor problem. .......... 100
7.6 NMOS PLA for the 2-bit greatest common divisor. ..................... 100
7.7 CMOS PLA for the 2-bit greatest common divisor. ..................... 101
7.8 Entity description generation. ....................................... 103
7.9 Architecture of the circuit implementing SOP expression. .......... 104
7.10 Abstract syntax tree generated for VHDL code generation. .......... 104
7.11 VHDL code generated. ............................................. 105
7.12 Layout generated from the VHDL code. .............................. 106

8.1 Monotonicity between approximations. .............................. 113
8.2 Function space in AKC. ............................................. 114
8.3 Nested circuit realization. ......................................... 115
8.4 Graphical representation of logical relations in functions. ......... 115
8.5 Weighted adjacency matrix for the example. .......................... 117
8.6 Logical relationship between nodes with common sub-expressions. . 118
8.7 Nested circuit realization after identifying common sub-expressions. . 119
8.8 An example to illustrate the localization of sub-expressions in a function. 121
8.9 Given node set along with their parent nodes. 123

9.1 General model for realizing a finite-state machine. 132
9.2 An example of deriving the output function specification. 137
9.3 Schematic interpretation of a recursive definition. 138
9.4 Next-state functions derived for the parity detector example. 139
9.5 Circuit for data registers in the synthesized hardware. 140
9.6 Extended model sequential circuits. 142
9.7 Function definition synthesized for Ready? circuit. 143
9.8 FHDL specification of the diversity switch. 145
9.9 Output function definition for the diversity switch. 145
9.10 Next-state function definition for the diversity switch. 146
9.11 Ready? function definition for the diversity switch. 146
9.12 Inter-connection for the diversity switch. 147

10.1 FHDL description of traffic light controller. 150
10.2 Lattice of values in $T^2$. 152
10.3 Geometric interpretation of f . g. 158
Chapter 1

Introduction

The Very Large Scale Integrated (VLSI) circuit revolution has been facilitated by dramatic changes in the field of Computer-Aided Design (CAD). Advances in CAD made possible the realization of Application Specific Integrated Circuits (ASICs) and have helped spur growth of the electronic industry.

The most dramatic of these changes in how VLSI CAD is done is an elevation in the level at which the design is undertaken. Initially, physical design tools, and later logic synthesis programs were used in the design of integrated circuits. As VLSI densities grew, these technique were found to be cumbersome. Now, high-level synthesis tools are emerging as an effective environment for automating the integrated circuit design, and permitting designers to cope with extremely complex systems.

*High-level synthesis* is a broad term used to define circuit synthesis from abstract specifications. Specifications model the behavior of a circuit. Specifications are usually written in a high-level description language called *Hardware Description Languages* (HDLs). HDLs are usually similar to existing conventional programming languages such as C and ADA. The most commonly ascribed reason for this similarity is the wide spread familiarity with and use of these languages.
1.1 Motivation and Objectives

In addition to the shortening of design time, use of high-level synthesis tools provides several advantages. First, these tools isolate the concerns of target technology from the design. This provides portability between different implementation technologies. The second advantage provided by synthesis tools is the design abstraction. Behavioral modeling languages help designers conceptualize their designs. This is a growing concern as designs become larger and more complex. Other advantages of using high-level synthesis tools include optimality and correctness of realizations. Existing synthesis tools offer various optimizations involving area and time to produce high quality designs.

Lack of effective formal verification techniques to prove the correctness of the synthesized designs is a major impediment for high-level synthesis. Formal verification is a methodology for proving that the specification and the implementation are functionally equivalent. The inability of present synthesis tools to provide formal verification can be directly attributed to basing HDLs on conventional programming languages. Application of formal verification techniques to conventional HDLs is complicated by the:

- immense complexity or lack of formal semantics of HDLs, and
- lack of formal description of how the circuit was constructed due to the absence of a valid underlying formalism.

Research in programming languages has led to a new programming paradigm – the functional or applicative paradigm, in which computation is carried out entirely through the evaluation of expressions. In addition to the simple and precise semantics of these languages, the abstract nature of programs in these languages render them appropriate for high-level behavioral specifications. This research describes a method based on functional programming languages for the specification and synthesis of hardware. The goal is to show that functional languages provide a viable approach to specifying and synthesizing circuits. This applicability of functional
languages is based on domain theoretic concepts. These mathematical foundations, which provide the formal basis for these languages, can be utilized in the synthesis process for generating hardware descriptions and in proving properties of the specification.

1.2 Overview

This thesis describes a method based on functional languages and domain theory for the specification and synthesis of hardware. The system described here uses a strict, first-order functional language, FHDL (Functional Hardware Description Language), to represent combinational and sequential behavior. The research concentrates mainly on synthesis from recursive specifications. Synthesis from non-recursive specifications is a direct consequence of the approach used for recursive/combinatorial synthesis and can be adopted without any changes to the algorithms used.

Recursion needs to be eliminated during hardware synthesis in order to avoid the necessity of a stack. Recursion removal is accomplished by computing the fixpoints of a recursion equation using an iterative approach. This iterative approach replaces recursive specifications with their equivalent non-recursive fixpoints. This approach has a solid basis in domain theory. The theoretical underpinnings guarantee equivalence between the specification and the hardware realization. Hence the realization is guaranteed to be correct with respect to the specifications.

The iterative approach adopted here requires that equality on functions be defined. Frontiers, a canonical representation of a function, are used to test the equality of functions. Using this representation of functions, the equality of two functions reduces to the comparison of two frontiers. Using frontiers to compute the fixpoints has other advantages. It turns out that a frontier set corresponds to minimal sum-of-product expression of the given function in the domain of application of the synthesis algorithms. Frontiers algorithms are applicable in domain 2 (the set with \{0, 1\} with (partial) ordering 0 < 1) of monotonic functions. Optimization of the synthesized circuits in the domain of specification can be accomplished through the
use of existing logic minimization tools.

Synthesis is accomplished by transforming high-level specifications in FHDL to a lower level of abstraction using a set of transformation rules. The transformations convert the specifications in FHDL into Kernel Language (KL) specifications while preserving the meaning of the representation. KL is defined on domain $2$ with $\text{and}$, $\text{or}$, and $\text{not}$ as its domain operations. Functions in KL assume the form:

$$f : T^n \rightarrow \ldots \rightarrow T^n \rightarrow T$$

where $T$ is domain $2$. Any such mapping on domain $2$ has a sum-of-product representation. Our approach to synthesis is to use frontiers to find this mapping.

KL has $\text{not}$ as one of its domain operations which is the source of non-monotonicity in functions. Frontiers algorithms are applicable only to monotonic functions. Domain inflation is proposed as a way to overcome this dilemma. By inflating the domain, non-monotonic functions can be represented as monotonic functions making specifications amenable to solution by frontiers algorithms.

Synthesis of hardware is accomplished by mapping sum-of-products expressions to hardware. Two distinct forms of hardware result due to the synthesis procedure:

- Two-level logic in the form of a PLA.
- Multi-level (nested) logic or random logic. Multi-level logic synthesis is a direct consequence of the iterative approach used for recursion elimination.

Multi-level synthesis refers to synthesis techniques that derive circuits with an unlimited number of intermediate signals. The synthesis of multi-level circuits presented here leverages on the naturally nested structure that results from the progression of the frontier set. Multi-level implementations seek to optimize the circuit through the minimization of number of literals in a given function. In this pursuit, this thesis explores the use of frontiers approach to identify common sub-expressions and derive factored forms.

Synthesis of sequential circuits is accomplished using a finite state machine model. Lists or streams are used to model input and output sequences. Using
the finite-state machine model, the task of sequential circuit synthesis is reduced to that of the synthesis of functions that are needed to implement a state machine. This approach enables the algorithms employed for combinational logic synthesis directly applicable for sequential circuit synthesis with little or no modification. Figure 1.1 summarizes the synthesis procedure described here.

1.3 Outline

The rest of this thesis is organized as follows:

- **Chapter 2: High-level Synthesis – An Overview.** This chapter discusses some of the techniques commonly employed in high-level synthesis. Synthesis from structural and behavioral description, using both imperative and applicative languages, are discussed.
• Chapter 3: Functional Languages and Recursion. The reasons for choosing functional languages as a specification language are justified in this chapter. This chapter addresses the problem of recursion removal from functional specification and presents important theoretical results that limit recursion removal in program transformation systems.

• Chapter 4: Fixpoints of Recursion Equations. This chapter introduces the concept of fixpoints. Frontiers and frontiers algorithms are used (with examples) to compute the fixpoints of recursion equations.

• Chapter 5: Specification Languages. The formal semantics of FHDL are presented. FHDL descriptions are transformed into an intermediate language, KL. Specifications in KL can be viewed as specifications in a boolean domain (domain 2) with “and”, “or” and “not” as the domain operations. The formal semantics of KL and a translation scheme from FHDL to KL are presented.

• Chapter 6: Foundations for Synthesis. Frontiers algorithms are applicable for monotonic functions only. But the kernel language, KL, is non-monotonic due to the not operation. This chapter introduces the concept of domain inflation to overcome the problem of non-monotonicity. A new frontiers algorithm, based on computing frontiers from the abstract syntax of the function, is also presented.

• Chapter 7: Circuit Synthesis. This chapter discusses the interface between this tool with the existing back-end systems for circuit generation. The current interface supports a VHDL code generator that forms input to a VHDL compiler, and PLA generation.

• Chapter 8: Multi-Level Logic Synthesis. Multi-level synthesis refers to circuits with an unlimited number of intermediate signals. Synthesis of multi-level circuits which correspond to the Ascending Kleene Chain is discussed.

• Chapter 9: Synthesis of Sequential Circuits. Synthesis of sequential circuits is accomplished by identifying the appropriate control and state infor-
motion elements of the circuits connected via appropriate feedback loops. This chapter presents a methodology for identifying the requisite information and synthesizing sequential hardware based on a finite state machine architecture.

- Chapter 10: Conclusions. A summary of contributions of the work presented in this thesis are discussed. Avenues that can be explored to further this research are presented.
Chapter 2

High-Level Synthesis – An Overview

The complexity of integrated circuits (ICs) has increased by several orders of magnitude over the last decade due to continuing decreases in minimum feature sizes. The need to manage this increasing complexity and the competitive pressure to reduce the design cycle time has led to the development of computer-aided design tools to automate the design process. These tools are focused on synthesizing ICs that meet design objectives which are stated as specifications. These specifications can be stated at different levels of design abstraction. The generally accepted hierarchy, in ascending degree of abstraction is:

- **Physical**: The physical level of abstraction specifies the hardware realization of a design at a geometric level. Behavior is specified only as a consequence of geometry.

- **Structural**: The structural level of abstraction specifies the interconnection architecture of a particular design in terms of its lower level components (e.g. logic gates).

- **Behavioral**: The behavioral level of abstraction specifies the functional description that reflects the response of a circuit to an input stimuli.

*High-level synthesis* is a broad term used to define the process that refines abstract specifications to derive the corresponding logic circuits. The input specifications are analogous to existing programming languages (e.g. C, Pascal) and are
commonly known as hardware description languages (HDLs). The purpose of this chapter is to review the related work on hardware description languages. The objective is to convey the methodologies and techniques commonly used and is not intended as a survey of all languages and systems that are currently available. The languages or methodologies chosen here represent a cross-section of recently developed languages. There are too many languages either in development or in use to permit discussion of all of them.

This chapter begins with an overview of some recent hardware description languages that capture the design at structural and behavioral level. Some common techniques employed in the synthesis of hardware from behavioral specifications are discussed. Finally, circuit descriptions using formal logic are considered.

### 2.1 Hardware Description Languages

A hardware description language allows design capture either from a structural or a behavioral specification. This section outlines the purpose of these specification styles.

#### 2.1.1 Structural Specification

A structural specification defines an interconnection architecture in terms of lower level components which achieves a designated behavior [120]. As an example, at the register transfer level of design, one can define the interconnections of various elements such as arithmetic logic units (ALU), data registers and multiplexors. A structural hardware description language is useful to record and manipulate large amounts of structural information that is involved in the design of a VLSI circuit [36].

These HDLs bear a strong resemblance to structured programming languages. Each component is defined by a function or a procedure. The component inputs are analogous to the arguments of a function; the outputs are returned as a result. Continuing with the analogy, the wires that are internal to a component are the
local variables of a function. The instances of the subcomponent making up the part are generated by making “calls” to the appropriate functions. These concepts are illustrated using a structural description subset of VHDL [113], which was developed in the mid-1980's as a part of U.S. Department of Defense (DOD) Very High Speed Integrated Circuit (VHSIC) program. It has been adopted as IEEE standard 1076.

Consider the 4-to-1 multiplexor shown in Figure 2.1. The VHDL structural description for the multiplexor is given in Figure 2.2.

The process of synthesizing from a structural specification involves flattening the input specifications and instantiating each component layout by calling each component from the component library. Then automatic placement and routing tools are invoked to generate the final layout. This process of transforming the input specifications to a final layout corresponds to compilation in the normal programming environment. The “compilers” that generate layouts from the input specifications using a pre-designed cell library are called silicon compilers.
--Package declaration for the input bus.

```vhdl
package buses is
    type nibble is array (0..3) of bit;
end buses;
```

-- Multiplexor is defined as an entity. This block defines the interface.

```vhdl
entity multiplexor (data, cntrl : in nibble; output : out bit) is
end multiplexor;
```

-- Structural description defines how the components are interconnected.

```vhdl
architectural body muxbody of multiplexor is
    component and (a, b : in bit; z : out bit);
    component or (a, b : in bit; z : out bit);
    signal temp : nibble;
    signal a, b : bit;

    begin
        mux: for i in 0..3 generate
            and (data (i), cntrl (i), temp (i))
        end generate;
        mux 1: or (temp(0), temp(1), a);
        mux 2: or (temp(2), temp(3), b);
        mux 3: or (a, b, output);
    end;
```

Figure 2.2: VHDL description of multiplexor.

2.1.2 Behavioral Specification

When an idea for a design is conceived its desired behavior may be described only in an informal manner using algorithms or timing diagrams. Clearly, it would be quite helpful to have a formal means to describe the behavior of a circuit at this stage. Moreover, descriptions at the behavioral level are much more abstract than structural level descriptions and hence are easier to write, understand, and modify. This makes them less error prone.

A circuit's behavior is characterized by the way in which its outputs depend on its inputs and its initial values. For example, the behavior of a piece of combinational logic could be described simply by a set of boolean equations relating the outputs to its inputs. This is a static view of the behavior and is not sufficiently general.
to describe most types of hardware. A more general approach was considered in CIRCAL [81]. Instead of boolean equations, the behavior was modeled in terms of a finite state machine, where changes in the input cause the circuit to move from one state to another with the outputs being static functions of the inputs and internal states. The language provides a very close correspondence between state transition diagrams and the language representation of the behavior.

Consider for example the behavioral description of an inverter given in Figure 2.3. The state names are mnemonics to convey information. The two digits at the end of the mnemonics represent the values on the input and the output of inverter. There are two stable states (INV01 and INV10) and two unstable states (INV11 and INV00). In the figure, *in* and *out* indicate the input and the output. For example, the second statement of the specification can be interpreted as: “when the inverter is in state INV11, it produces a 0 at the output port if the input is 1 and transitions to the state INV10, or if the input is 0, it transitions to the state INV01”. The corresponding state transition diagram is shown in Figure 2.4.

CIRCAL is quite different from most other hardware description languages, which tend to have much in common with conventional programming languages. It presents a general framework for the representation of circuits, but it is not clear how well it can be applied to more abstract behavioral descriptions.

Modern behavioral description languages raise the level of abstraction to the algorithmic level. The specifications are written as an algorithm in an imperative

![Figure 2.3: CIRCAL description of inverter.](image-url)
Figure 2.4: State transition diagram for inverter.

```
entity gcd is
  port (X, Y: in integer; Z: out integer)
end gcd;
architecture behaviour of gcd is
begin
  process
    variable A, B: integer;
    A := X;
    B := Y;
    while (A != B) loop
      if (A < B) then
        B := B - A;
      else
        A := A - B;
      end if;
    end loop;
    Z := A;
  end process;
end arch;
```

Figure 2.5: A behavioral specification fragment in VHDL programming language such as C or Pascal [7, 76, 115]. Behavioral VHDL [24] is one
such language. VHDL along with Verilog\(^1\), the most widely used commercial HDL, are emerging as industrial standards for behavioral specifications [34]. A sample behavioral description of the greatest common divisor problem using behavioral VHDL is given in Figure 2.5.

### 2.2 Synthesis from Behavioral Specifications

This section deals with the synthesis of hardware structures from behavioral level descriptions. Though, most hardware description languages allow both synchronous and asynchronous design specifications, synthesis from these specifications is still restricted to synchronous hardware [34]. However, there has been growing interest in extending the current techniques and deriving new approaches for the synthesis of asynchronous circuits [8, 32, 38, 72]. This thesis focuses on the synthesis of synchronous hardware.

For a general purpose synthesis system, the main steps involved in the synthesis of a detailed representation from an abstract behavioral specification can be summarized as:

- **Compilation**: This is a one-to-one transformation of source specifications into an internal representation, usually a graph. The graph can be a data flow graph and/or a control flow graph.

- **General purpose transformations**: These transformations include general transformations used for high-level programming languages, such as constant propagation, dead code elimination, identification of common sub-expressions, loop unrolling, and code motion. Some synthesis tools also include specific transformations such as lifetime analysis of variables to decide which variables must be implemented as registers and which might share the same register.

- **Architecture specific transformations**: These transformations are used by synthesis systems targeted for a pre-designated architecture.

\(^1\)© Verilog is a trademark of Cadence Design Systems Inc.
• **Scheduling**: Scheduling is the problem of minimizing the number of control steps while satisfying the hardware resource constraints. A control step is a fundamental sequencing unit in a synchronous digital system and is equivalent to a state in a finite state machine. This step is also sometimes referred to as *control path synthesis*.

• **Resource Allocation**: This is the process of mapping operators onto operators, variables to registers, and edges of the flow graph to buses as interconnections between operators and registers. Allocation is also known as *data path synthesis*.

• **Partitioning**: Partitioning is aimed at generating more modular designs. Partitioning can be performed both at the algorithmic level and the structural level.

Finally the design is converted into hardware. This can be done by passing the results of these transformations to a logic synthesis tool which can perform the optimizations on combinational logic and generate the hardware, or by generating an equivalent structural description which can be used by lower level synthesis tools.

The following subsections detail the above mentioned subtasks of the synthesis process.

### 2.2.1 Compilation

The first step in high level synthesis is to transform the source specifications into an internal representation. As in other programming languages, the usual choice for an internal representation is a graph. The transformation of the graph into a structure constitutes synthesis. The graph can be a control flow graph or a data flow graph.

The control flow graph is a directed graph, which indicates the sequence in which the operations are performed. It is derived from the explicit order given in the original specification. Nodes in the graph represent the operations to be performed and the flow of control can be determined from the edges.
Data flow graphs are also directed graphs which depict the orderings based on the data dependencies in the original specification. The nodes correspond to data and operations and the directed edges correspond to the direction of data flow. The separation of data and control flow are only one of the several ways that a high level synthesis system represents the information. Some systems mix both the graphs to form a dependency graph or use all of them [25, 26, 84, 96]. The dependency graph contains only essential operational dependencies and data dependencies. Figure 2.6 shows the control, data and dependence graphs for the behavioral description given in Figure 2.5. In the example, the dependence graph differs from the control graph at the point of initial assignments, which are independent of each other.

2.2.2 General Purpose Transformations

Compilation is essentially a one-to-one transformation of the source program into an internal representation. Initial optimizations that can be performed are identical
to the compiler optimizations for a high-level language [107]. These include dead code elimination, common sub-expression elimination, strength reduction, and others. Some transformations more specific to hardware synthesis, which preserve the behavior of the specifications, are also performed at this stage. Such transformations are known as behavior preserving transformations. Some transformations that are applied in a synthesis system that are common to high-level language compilers are:

- **Strength reduction**: Strength reduction is an optimization technique which replaces one type of operation by an equivalent operation which is more efficient. For example, multiplication by two can be replaced by a shift left operation and division by two can be replaced by a right shift.

- **Loop unrolling**: A cycle in a graph represents a loop in its behavioral specifications, e.g. a while loop. Loops violate hardware constraints, e.g. writing repeatedly to a port or a register. If the number of iterations is small and fixed, this optimization can be performed. Loop unrolling eliminates cycles but at the expense of replicating the body of the loop which in turn increases the depth of the flow graph.

These optimizations are performed either interactively or automatically in several of the major synthesis systems [28, 95, 108]. These transformations, though general enough, must be applied with care, else the control graphs may grow unwieldily [79].

### 2.2.3 Architecture Specific Transformations

The goal of architecture specific transformations is to optimize the circuit behavior for a target architecture. High performance hardware can be synthesized by reducing the number of levels in the flow graph using the properties of the target architecture. These architecture directed transformations have specific application areas such as digital signal processing (DSP) applications [29, 57, 69]. Based on characteristics of the algorithm such as modularity, signal flow, and inherent parallelism, a predefined architecture is selected which best suits the algorithm.
High performance designs can be obtained by pipelining the computations [31, 85, 92]. In pipelining, each computational task (e.g. an instruction) is partitioned into a sequence of subtasks and each subtask is executed during a clock cycle. Subtasks of consecutive tasks are overlapped in time on different parts of the pipeline circuit. This is done by identifying the multiple disconnected graphs for parallel independent tasks from the data flow graph. Once the pipelined data flow paths are identified, standard logic synthesis techniques can then be applied to yield logic designs.

2.2.4 Scheduling

Scheduling and allocation are perhaps the most important tasks in a synthesis system. These two tasks are closely inter-related and both them are aimed at improving the performance while minimizing the hardware used [101, 102, 114]. This section presents some scheduling algorithms. Allocation is discussed in the next section.

Scheduling assigns each operation to a specific control step. A control step is equivalent to a state in a finite state machine. The goal of a scheduler is to minimize the number of control steps, while taking into consideration both the hardware constraints as well as storage and inter-connection costs. Some of the common hardware constraints that every scheduler should satisfy are [27]: every unit of hardware can be used only once during a control step, data registers can be loaded only once during a control step, evaluation may be performed only once, and buses may carry only one value. Commonly used scheduling algorithms in the synthesis process include As Soon As Possible (ASAP) scheduling, As Late As Possible (ALAP) scheduling, list scheduling, and force directed scheduling. This section highlights these scheduling algorithms.

ASAP and ALAP scheduling algorithms [79] identify data dependencies from the data flow graph. In an ASAP schedule, all the operations are assigned to the earliest possible control step. In an ALAP schedule all the operations are scheduled as late as possible. Both these approaches consist of sorting the data flow graph topologically according to the data dependencies and based on the available hardware resources,
the operations are scheduled in depth-first (last) order.

Another approach to scheduling is list scheduling. A constraint on maximum allowable resources is provided and the algorithm tries to minimize the total execution time within those constraints. A priority function is used to defer the operation when resource conflicts occur. A common priority function is the mobility function, which is computed as the difference of ASAP and ALAP schedules. SLICER [84] and RECALS II [94] use list scheduling to optimize the number of control steps.

Force directed scheduling [90, 91] is probably the most complex scheduling algorithm in use today. It is aimed at minimizing the hardware requirements by ensuring that each structural unit has a high utilization. The force on each operation is used as a guiding factor for choosing which operation to schedule next and where to schedule it. The probability of each operation being in a particular control step is computed using the mobility function. The demand on the hardware is computed by adding the probabilities at each control step. Then the operations are scheduled such that the demand differences among the operations of the control step is minimized.

Scheduling is NP-complete [27]. All of the scheduling algorithms use heuristics [50, 67] to search for a good schedule. Most of these algorithms give reasonably good results, even though they are not guaranteed to be the optimal.

2.2.5 Resource Allocation

In hardware design, various resources can be identified as, for instance, functional units, storage elements, and interconnection elements. Resource allocation is the process of assigning appropriate units to operations. Operations are bound to operators, data values are assigned to registers, and edges in the flow graph form the communication paths between functional units and storage elements. The problem of resource allocation is to minimize the overall hardware used.

The process of binding operations to functional units is also referred to as “module binding” [68]. The modules are selected from a set of modules available. For a
given operation, there may exist a number of different functional units exhibiting the same behavior. The choice of a particular module is based on the user-specified constraints such as performance, area, and power dissipation. For example, an allocator may choose a ripple-carry adder when the area is a constraint and a carry-lookahead adder when performance is a constraint.

For a given schedule, two operations can share the functional unit if the two operations are mutually exclusive, i.e. they are scheduled in different control steps. The problem of allocation is then to form groups of mutually exclusive operations in such a way that the number of groups is minimized. The problem of minimizing the storage and interconnection can be formulated similarly.

The values produced by an operation in a control step and used later must be assigned to a register. This can be avoided only if the value is used immediately in the next control step. This is determined by performing life time analysis [96]. Values can be allocated to the same storage element if their lifetimes do not overlap. Registers may be shared during the overlapping lifespan if they lie on different data paths of a conditional branch [25, 85]. Storage allocation should also consider the interconnection costs and thus produce simple communication paths.

The minimization of functional unit allocation, storage element allocation and inter-connection allocation is done separately. The minimization of all of them together is usually too complex for a reasonable size design. The minimization algorithms for allocation of different resources in the literature can be identified as:

- **Linear programming**: The problem of allocation is formulated as a linear programming problem, where limited resources can be allocated to competing activities in an optimal manner. This approach was used by Hafer and Parker [45].

- **Knowledge based**: These are used by expert system based synthesis approaches [68]. The heuristics used are captured as rules and allocation is based on which rules apply. The rules are aimed at minimizing the overall hardware cost.

- **Graph based**: Given a flow graph, the problem of design improvement is con-
cerned with the minimization of number of functional units, storage elements and interconnections. Each variable is represented as a node and the relationship of shareability is represented as connectivity of the nodes. The minimization of each hardware resource is formulated as the clique partitioning problem [109].

Like scheduling, the problem of resource allocation is also NP-complete [27]. Since scheduling and allocation are highly interdependent, it is not clear which transformation should be done first.

2.2.6 Partitioning

Partitioning is the process of decomposing the design into independent modular pieces that can be processed more or less independently. This is equivalent to hierarchical design at a register-transfer level. At a higher level, partitioning addresses the issue of behavioral and structural partitioning. The Yorktown Silicon Compiler [25, 26] partitions its design at a functional block level. This is used primarily to limit the size of the design being synthesized. The pipelining in Sehwa [85] is the result of partitioning of behavior into different stages. A set of behavioral and structural transformations were developed for Carnegie Mellon University’s System Architect’s Workbench [116] to allow the exploration of algorithmic level design alternatives. The partitioning problem addresses the architectural issues such as [116]:

- Should the design be partitioned such that it can be implemented on a single VLSI chip or more? If it has to be implemented on more than one chip, where should the design be partitioned?

- Can the design be split into independent processes to incorporate concurrency?

- Should the design be pipelined or un-pipelined? If the design is to be pipelined, how many stages should there be in the pipeline?

The development of transformations addressing these issues is just beginning. Current implementations provide an interactive framework to manually explore the
design alternatives.

2.3 New Paradigms for HDLs

A survey of HDLs [5] shows that most languages in use are imperative, i.e. they rely on state mutation by assignment statements and sequential (step-by-step) semantics. This is unfortunate because HDLs based on imperative languages have several deficiencies including the presence of assignments and the lack of formal semantics. Also the sequential nature of the specifications make it difficult to find optimal implementations. The question naturally arises as to what notation or formalism is better suited to expressing the semantics of hardware systems.

On the other hand, some recent programming languages, for example Miranda [111] and Haskell [52], do have a well defined, and relatively simple, semantics. In addition to the relative ease with which specifications can be written in these languages, the properties of the programs written can be formally verified. Adopting this point of view, this section explores the use of formal logic for hardware specification. Yet another emerging specification style, based on functional notation, is fundamental to this thesis. Functional programming is discussed in the following chapters after a brief introduction to the notion and impediments of using this tack for hardware description.

2.3.1 Logic based HDLs

Logic, especially higher-order logic, was one of earliest formalisms considered for the specification of circuits. Higher-order logic allows variables to range over predicates and functions. With higher-order logic, the behavior of digital systems can be rigorously specified, and the correctness of digital circuits can be formally verified [43, 46, 65].

In this section, two examples of an inverter are presented that illustrate the use of predicates. The predicates that are used are higher-order. In the first example, the values on the lines are modeled with functions, and consequently the predicates
are used to specify the behavior of the device. In the second example, the device is built by connecting together the individual components that make up this device. Predicates are used to specify the behavior of the individual components. Then the behavior of the overall system is derived in two steps:

- Conjoining the constraints corresponding to the components.
- Existentially quantifying the variables corresponding to the internal lines (internal lines correspond to signal lines that are not visible to the “outside world” and the external lines might be thought as the interface lines).

Consider the inverter circuit given in Figure 2.7. Input and Output correspond to the inverter input and output, respectively. Any change in the input causes corresponding changes in the values occurring on the lines of the device. This indicates the need for time varying values for the arguments of the predicates used to represent the device behavior. Such values correspond to “waveforms” and can be modeled by functions of time. Thus, the behavior of an inverter with a delay of \( \delta \) units of time can be specified with a predicate \textit{inverter} as:
\[
\text{inverter}(\text{input, output}) \equiv \forall t. \text{output}(t + \delta) = -\text{input}(t)
\]

Here the values on the lines \textit{input} and \textit{output} which are functions of time are mapped to values represented by booleans. These functions are in \textit{inverter} relation if and only if for all times \(t\), the value of the \textit{output} at time \(t + \delta\) equals negated value of the \textit{input} at time \(t\).

The inverter shown in Figure 2.7 can also be viewed as a structure built out of four components: a power source (\(Pwr\)), a ground element (\(Gnd\)), an n-transistor (\(ntrans\)) and a p-transistor (\(ptrans\)). These components can be modeled as:

- The power source can be modeled by a predicate \(pwr\), which constrains the value on the line \(p\) to be true as:

\[
pwr(p) \equiv (p = \text{true})
\]

- The ground line can be modeled by a predicate \(gnd\), which constrains the value on the line \(p\) to be false as:

\[
gnd(p) \equiv (p = \text{false})
\]

- The n-transistor, which acts as a switch, can be modeled as:

\[
ntrans(g, d, s) \equiv (g \Rightarrow (d = s))
\]

where \(d\) and \(s\) represent the drain and source of the transistor and \(g\) represents the gate.

- The p-transistor can be modeled as a switch which conducts when its gate is low and can be defined as:

\[
ptrans(g, d, s) \equiv (\neg g \Rightarrow (d = s))
\]

where \(g\), \(d\) and \(s\) are as defined previously.

Conjoining together the constraints from the four components and existentially quantifying the internal line variables yields the following predicate \textit{inverter}:
\[
\text{inverter}(\text{input, output}) \equiv \exists p_1, p_2. \text{pwr}(p_1) \land \text{gnd}(p_2) \\
\land \text{ptrans}(\text{input, } p_1, \text{output}) \\
\land \text{ntrans}(\text{input, output, } p_2)
\]

The examples presented here demonstrate that higher-order logic can provide a formalism in which both the structure and behavior can be specified and that specifications can be made to look like conventional hardware descriptions while maintaining logical purity. An attempt has also been made to use logic programming languages like Prolog as hardware description languages [74, 106]. The specifications using logic descriptions simplify the verification process. A number of verification techniques based on higher-order logic and the mechanization of these techniques are presented in [43, 46, 82]. The disadvantages are the complexity to which these specifications grow for clocked sequential circuit descriptions and the deductive proof style. Hanna and Daeche [46] present some interesting statistics for deductive proof of a D-flipflop which required one week of concentrated effort by an experienced user of the system, using about 5000 “primitive” inference rules and 1600 “high level” (derived from application of primitive inference rules) inferences.

2.4 Concluding Remarks

Hardware description languages allow the design to be specified at several levels of abstraction, thus providing designers with an environment in which they can rapidly explore various design alternatives. The evolution of HDLs has followed that of programming languages in a sense that there has been growing emphasis on formal semantics. This research develops a method of synthesis which uses functional languages for the specification. The most significant benefit of functional languages is the mathematical basis which provides means for systematic program transformation and formal design verification. Recognizing the potential benefits of functional languages, this research work adopts this framework for hardware specification.
Chapter 3

Functional Languages and Recursion

The eventual goal of the research reported here is to derive digital circuits from high-level functional specifications. Though the choice of language can be partly justified as a matter of preference, justice can only be done by rendering answers to the questions:

- What are functional programming languages?
- Why use functional programming languages?
- How to program in functional languages?

This chapter tackles the first two issues: what and why. Introductory books on functional programming (e.g. [12, 40, 48]) provide an excellent introduction to programming in the functional style. Section 3.3 considers how to use functional languages for hardware specification and describes a method of synthesis proposed in [59].

One of the facilities inherent in functional languages is recursion. Although many algorithms can be efficiently and succinctly defined in recursive forms, such recursive forms are not most suitable for efficient implementations. The characteristic property of implementation of these languages is that they use and manipulate a stack. From the hardware synthesis point of view, it is difficult to derive a stack for recursive definition, as the size of stack and how the stack is manipulated at run time are not known at compile time. Semantics preserving program transformation
schemes are generally used to eliminate recursive definitions. The chief objective of such program transformation schemes is to produce more efficient programs, though not necessarily a simpler program. Section 3.5 presents some of the research efforts aimed at recursion removal.

3.1 Functional Programming

Functional programming languages are so called because the notion of function is the primary building block for programs. Programs consist entirely of functions. The main program itself is a function which acts on the input argument values to deliver the output result. Typically, the main function is defined in terms of other functions and this process of function definition continues, until the lowest level where functions are defined in terms of language primitives.

In such a paradigm, functions are treated as first class objects: functions can be stored in data structures, passed as arguments, and returned as results. Syntactically, functional languages have an equational look in which functions are defined as expressions. A program consists of:

- A set of objects. The objects can be atomic (e.g. numbers, characters, etc.) or composite (e.g. lists, strings, etc.).

- A set of function definitions. These functions map objects into objects. A function can be a primitive function, a user defined function, or a combination of functions using functional forms.

- A set of functional forms used to combine existing functions or objects to form new functions (e.g. “map” in Miranda, function composition etc.).

Computation in these languages is due to function application. Computation is the evaluation of an expression defining the program, i.e. solving the equations. Such computational style is possible because “pure” functional languages have no side-effects. This “purity” enables one to solve the equations in much the same way
one solves equations in conventional algebra using only substitution and properties of the operators in the defining function. This “purity” in functional languages is also known as referential transparency which is discussed later on in this section.

The functional programming style is often described as expressing what is being computed rather than how it is being computed, as in imperative languages. The differences between functional programming and imperative programming have been discussed in depth in [4, 51]. The main distinguishing features are:

- **Higher-Order**: As a direct consequence of functions as first-class objects, functions can appear anywhere a value can appear. Hughes [53] provides convincing arguments that higher-order functions greatly contribute to modularity in functional languages.

- **Fully Lazy Evaluation**: An expression is evaluated only when its value is needed, and is evaluated only once [12].

- **Strong Typing**: Functional languages can be equipped with type checking and type inferencing mechanisms. A successfully type-checked program never fails to run due to type violation. Typing information not only aids in understanding a program, but also helps in debugging it.

- **Formal Foundation**: Modern functional languages are embellishments of the \( \lambda \)-calculus [6]. The notion of reduction in functional languages is due to the well-known Church-Rosser theorem. The reduction (rewrite) rules of the \( \lambda \)-calculus depend on the substitution of an expression \( e_1 \) for all free occurrences of an identifier \( x \) in an expression \( e_2 \), which is written as \( [e_1/x]e_2 \). A variable \( v \) in a \( \lambda \)-expression is said to be free if it is not within the scope of “\( \lambda v \)”; otherwise it is bound. For example, in the \( \lambda \)-expression:

\[
\lambda x. (+ x y)
\]

\( x \) is a bound variable and \( y \) is a free variable.
The three standard rewrite rules of the $\lambda$-calculus for an expression are defined as:

1. $\alpha$-conversion (renaming):
   \[ \lambda x_i.e \iff \lambda x_j.[x_j/x_i]e \text{ where } x_j \text{ does not occur free in } e. \]

2. $\beta$-conversion (application):
   \[ (\lambda x.e_1)e_2 \iff [e_2/x]e_1 \]

3. $\eta$-conversion (equivalence):
   \[ \lambda x.(e \ x) \iff e \text{ if } x \text{ does not occur free in } e. \]

A $\lambda$-expression is said to be in its normal form if it cannot be further reduced using $\beta$ or $\eta$ conversion rules. The normal form of a $\lambda$-expression is the value of the expression. Thus the notion of computation reduces to reducing an expression to its normal form using the rewrite rules. $\lambda$-calculus can be shown to be consistent as a mathematical system [6].

- **Formal Semantics**: Functional languages are very closely related to denotational semantics [103]. The semantics of these languages is relatively simple. Elaborations arise only due to additional syntactic features that ease the use of the language.

### 3.2 Why Functional Languages?

Advocates of functional programming claim that programs can be written accurately, concisely, quickly and in a style closer to traditional mathematical notation. Turner [110] characterizes this as the descriptive power of functional languages. The other important aspect of any notation is its manipulative power, i.e. suitability towards formal manipulations of descriptions. These properties are highlighted in the following subsection.
3.2.1 Descriptive Power

The descriptive power or expressive power of the functional style of programming is due to freeing the programmer from concerns about control. Thus the general consensus is that functional programs define what needs to be done rather than how it is to be accomplished. The best way to demonstrate the expressive power of functional languages is to take some problem and show how it is possible to develop a working solution within a fraction of effort that would have been necessary in conventional imperative languages.

Figure 3.1 presents a program for binary search in Miranda\(^1\). Binary search is the problem of determining whether a given element \(x\) is present in a set. Using divide-and-conquer approach, the set is organized as a lexically ordered tree. Each node in the tree holds an element of the set. A node is empty when the set is completely exhausted. When binsearch finds an empty node (Nilt), the result is false. For each non-empty node, there exists a left sub-tree and a right sub-tree. When a node is found which contains the goal, the result is true. The binary search algorithm searches only either a left sub-tree or a right sub-tree, depending on the result of comparison of a given element with the current node, starting from the root node of the tree. If \(goal < x\), where \(x\) is the current node value, then only the left sub-tree needs to be searched. If \(goal > x\), then only the right sub-tree is

\(^1\)© Miranda is trademark of Research Software Ltd.
searched. For example, the steps involved in finding a value are shown below:

```plaintext
binsearch 1 (Node 2 (Node 1 Nilt Nilt) (Node 3 Nilt Nilt))
⇒ binsearch 1 (Node 1 Nilt Nilt)
⇒ True
```

A separation of calculation from control is provided by lazy evaluation. This somewhat frees the programmer's mind from efficiency considerations as values are not evaluated unless necessary. Laziness also provides an ability to compute with "infinite" data structures. Turner [110] provides strong arguments for using lazy lists combined with list comprehensions. Hughes [53] presents an argument for facilitating modularity in functional languages using higher-order functions and lazy evaluation as the "glue" to join pieces of program together.

### 3.2.2 Manipulative Power

The manipulative power of functional languages is derived from the principle of referential transparency. Informally, the principle of referential transparency can be expressed as follows: if in a certain mathematical context the variable name \( x \) stands for an expression \( E \), then any occurrence of \( x \) in any expression in that context may be replaced by \( E \) and vice versa. In a functional program, the principle of referential transparency holds because there are no side-effects. Hence expressions can be transformed in a simple fashion using clean equational style reasoning. The only transformation rules are:

- **Substitution** (as determined by the principle of referential transparency).
- **Specific rules**: Properties of the operators (as given by axioms or derived from definitions).

As an example, consider the following set of definitions:

\[
\begin{align*}
a &= 6 + b \\
b &= 33 - c
\end{align*}
\]
\[ c = 25 \]
\[ d = 3a + b \]

The "meaning" of the expression "\(3d + a\)" in the context of the above definitions can be derived using the transformation rules as:

\[
3d + a = 3d + (6 + b) \quad \text{(from } a = 6 + b) \\
= 3(3a + b) + (6 + b) \quad \text{(from } d = 3a + b) \\
= 9a + 3b + b + 6 \quad \text{(commutativity of +)} \\
= 9a + b(3 + 1) + 6 \quad \text{(distributivity of *)} \\
= 9a + 4b + 6 \quad \text{(+ for numbers)} \\
\]

\[ \ldots \]

\[ = 164 \]

Every time a definition appeared as justification, the substitution was used, otherwise some relevant property of the operators was used. The proof style remains essentially the same for more sophisticated definitions.

### 3.3 Functional Hardware Description Languages

From the previous discussion, it is evident that functional programming languages are amenable to formal reasoning and analysis. These languages are gaining acceptance due to their simple and precise semantics, and their strong mathematical foundation. Also, most theory of digital systems is formulated on the basis of mathematical functions. The flow of data in hardware is closely matched by substitution of expressions for function arguments. This establishes the functional specification as a natural notation for the description of the semantics of digital systems [14, 15]. Kanthamanon et. al. [66] present a comprehensive comparison between the use of functional and imperative style for behavioral specification of hardware and show that the functional style of description provides a consistent and a more natural/intuitive semantics for hardware descriptions.
The mathematical basis of these languages provide means for systematic program transformation and formal design verification [97]. Recognizing the potential benefits of functional programming, several researchers have considered these languages as models for high-level behavioral specifications. This section presents some of the research efforts made towards adapting functional languages as hardware description languages. The use of functional specifications for both structural and behavioral specifications will be considered.

3.3.1 Structural Specification

In functional languages, the basic notion of computation is a function. Functions map objects to objects and there exists different functional forms that combine functions. Objects can be atomic (e.g. bits, numbers) or composite (e.g. lists). Atomic objects are represented as “signals” in the hardware. Functions take these signals as arguments and produce signals as a result. Functions can be defined hierarchically, thereby permitting modular decomposition of the problem.

The functions in a hardware specification language fall into two categories:

1. **Elementary functions**: Arithmetic functions (such as “+” and “-”) and logic functions (such as “and”, “or”, and “not”) are taken as primitives in the language.

2. **Manipulator functions**: This category contains basic interconnection specification functions. These functions can be further classified as selector, structure modifying and structure generating functions.

   **Selector functions** include “first”, “last” (to select first and last signal lines) [99] or “select” (to select a particular signal line) [77] and other list processing functions. These functions never create new signals and their effect is independent of the input value. For example, to select the first, second and fourth signal lines, in µFP [99] one writes in the form “[first, second, fourth]”.

   **Structure modifying functions** rearrange atoms (signal lines) within the objects. These include list constructor functions like “apl”, which puts an element at the
front of a list, "apr", which appends an element at the end of the list \[99\], and other functions like "trans", "distl", "distr" and "appndl" which are defined in \(\nu\)FP \[89\] as:

\[
\begin{align*}
\text{trans}: ((1,2),(3,4)) &\rightarrow ((1,3),(2,4)) \\
\text{distl}: (a,(x,y,z)) &\rightarrow ((a,x),(a,y),(a,z)) \\
\text{distr}: ((x,y,z),a) &\rightarrow ((x,a),(y,a),(z,a)) \\
\text{appndl}: (a, (b,c,d)) &\rightarrow (a,b,c,d)
\end{align*}
\]

Higher-order functions such as the triangle function \(\triangleright\)\[99\], fall under the class of structure generating functions. The left and right versions of this function are shown in Figure 3.2. In the example (for a 4 bit argument), \(R \triangleright F\) applies increasing number of F's to each input element. \(L \triangleright F\) works in the reverse manner. From the example, it can be observed that:

\[
\text{reverse} \circ L \triangleright F \circ \text{reverse} = R \triangleright F
\]

where "reverse" is the list reversing function and the functional form \(\circ\) is functional composition.

The functional composition of two functions \(F \circ G\) is normally interpreted as "do G and then F". Other higher-order functions such as "reduce", and "map"
Figure 3.3: Geometrical interpretation of composition function.

```lisp
(defun halfadder (a, b)
    [a and b, a xor b]
enddef

defun fulladder (a, b, Ci)
    [first or second, third] @ appndl @ [first, halfadder @ [second, third]]
    @ appndr @ [halfadder @ [a, b], Ci]
enddef
```

Figure 3.4: Description of a full adder in vFP.

have similar interpretations of their own. All such interpretations can be mapped to
their corresponding geometric interpretation [99], which defines a possible structure
at hardware level. Figure 3.3 shows the geometrical interpretation of “F o G”. As an example, the structural description of a full adder circuit using two half-
adder circuits is given in Figure 3.4. The specifications are in a functional hardware
description language, vFP [89]. The composition is represented in vFP by “@”,
“appndr” is the converse of “appndl” as defined before and “first”, “second” and
“third” are list element selector functions.

Synthesis from structural specifications is accomplished by a series of transforma-
tions. Each transformation produces a small theorem. These theorems can be easily
proven using the formal semantics of the language. Thus the initial specification,
through a series of refinements, produces a description in an intermediate form that
can be mapped directly to hardware. Routing is the result of routing primitives and
combining forms of the function. Since the circuits produced are formally verified at each step of transformation no further verification is needed. Such transformations are known as correctness preserving transformations. Transformational systems for the synthesis of special purpose architectures suited for DSP applications have been reported in [63, 100].

3.3.2 Behavioral Specification

A survey of hardware description languages has shown that most hardware description languages are imperative in their specification style [5]. Most of them amount to coding in "C" and making calls to vendor-specific libraries [37]. Behavioral hardware description languages raise the level of design abstraction to an algorithmic level, thus deriving circuits which correspond to the algorithm being implemented. This is in contrast to the structural specifications where the inter-connection architecture between various components is specified.

If the objective of hardware description languages is to capture the design from the algorithmic description of the circuit, applicative languages have an edge over the conventional imperative languages. Applicative languages, with their expressive power, bridge the gap between intuitive algorithmic description and formal specifications.

A functional program is a system of simultaneous equations. Each equation defines a function with an expression (e.g. $f(x_1, \ldots, x_n) = e$, where $f$ is the defining function symbol and $e$ is an expression). Expressions are built from constants, identifiers, operators and applications. Conditional constructs (e.g. if ... then ... else ...), along with predicates are used to manage the flow of control. Assignments are not allowed. The symbols in an expression can range over the user defined function symbol. Such expressions are known as recursion equations. A system of recursion equations has been used by Johnson [58, 59, 60] as a specification language for deriving synchronous system descriptions. One might write the specification for the greatest common divisor problem in applicative terms as shown in Figure 3.5.
gcd xy = x, if x = y
= gcd x (y-x), if x < y
= gcd (x-y) y, otherwise

Figure 3.5: Functional description of greatest common divisor problem.

The notation used is that of Miranda. Transformation of functional specification to digital circuits is discussed in the next section.

3.4 Synthesis from Functional Specifications

The synthesis process can be viewed as translation of high-level specifications to low-level hardware components and their interconnection structure. The translation process incrementally refines the abstract specifications until a concrete description, that can be mapped directly to hardware, is obtained. Specifications in an applicative language, in addition to normal program constructs, allow liberal use of recursive definitions. This section concentrates on synthesis of digital circuits from recursive specifications.

Recursive definitions provide a clear and concise problem specification. However they pose a major problem in high-level synthesis. They require the use of stack. Although the underlying computational model is powerful enough to support stack operations, the specifications cannot be realized directly as the size of the stack and how the stack is accessed during the run time are unknown. A common solution is to transform these recursive equations into equivalent iterative equations or loops. This results in specifications analogous to flowchart descriptions of the computation. Just as flow chart descriptions are frequently used for digital design, iterative specifications are used as the realization specifications with the loops as feedback elements. Thus, the major tasks involved in synthesis from functional specifications are:

- Recursion removal.
Figure 3.6: Two versions of factorial function. (a) Recursive. (b) Iterative.

- Construction of an equivalent system description in a realization language.
- Optimization of the obtained solution.
- Partitioning the design to generate modular systems.

### 3.4.1 Transformations on Recursion Equations

The first step in the synthesis process is to translate these recursive specifications into an equivalent iterative form. Iterative form is of interest in general because of its correspondence to sequential control algorithms (i.e. programs) [70]. Since digital circuits are sequential in nature, results of research in recursion removal are useful in the synthesis of circuits. The basic approach is to transform the specifications repeatedly according to certain rules, until a satisfactory program is produced. The result is conversion of recursion equations into “flowchart” programs.

For example, consider the following two versions of one program which computes the factorial shown in Figure 3.6, following the notation of Miranda.

One is clear and abstract (Figure 3.6(a)), the other is more obscure, but efficient. The objective is to transform the recursive specifications (as given in Figure 3.6(a)) into an iterative algorithm (as in Figure 3.6(b)). There has been considerable theoretical investigation into how to translate recursive definitions into equivalent iterative ones, which is the topic of discussion for Section 3.5. For the time being, it suffices to assume that automated systems exist which mechanically convert recursive programs into equivalent iterative programs.
3.4.2 The Realization Language

A realization language should convey what parts a circuit contains and how they are connected. In general terms this can be considered as the netlist of a circuit. The netlist is generated from high-level specifications after recursion removal. The parts can be either storage elements (registers), or computational components. The behavior of the storage elements is governed by an external synchronizing agent or clock. The components are assumed to have distinct inputs and outputs. The method of synthesis shown here is due to Johnson [59].

3.4.2.1 Model of Behavior

The realization language is built from signal expressions [60]. A signal expression denotes a signal, or history of values acquired over a period of time. Before defining how the signals are expressed in terms of the components, the conventions used for describing different aspects of the circuit is introduced. The notation is due to [59].

- To avoid confusion between operator symbols in high-level specification and hardware components that correspond to the operation symbol, hardware component symbols are enclosed in boxes.

- The symbol "!" designates an anonymous register.

- Signal names are written in upper case.

3.4.2.2 Circuit Description

A circuit description is a set of signal defining equations. A signal definition is an equation of the form:

\[ X = s \]

where \( X \) is an identifier and \( s \) is a signal expression. A signal expression is one of:

- \([c]\), where \( c \) is a constant.
• \(X\), where \(x\) is an identifier.

• \(c \cdot s\), where \(c\) is a constant and \(s\) is a signal expression. The component \(\cdot\)! is initialized to \(c\) with subsequent values denoted by the signal expression \(s\).

• \(f(s_1, \ldots, s_n)\), where \(f\) is an \(n\)-place operation and each \(s_i\) is a signal expression.

For the factorial function, the signal expressions can be written as:

\[
\begin{align*}
N &= n \cdot \text{decrement} (N) \\
Y &= \text{multiply} (N, Y)
\end{align*}
\]

where \text{decrement} implements the operation \(n - 1\) in the original specification and \text{multiply} implements the operator symbol \(\cdot\). Since the target system is synchronous, all the registers are enabled by the same clock. Multiplexors replace the conditionals. Synthesis is straightforward once the iterative specifications are derived.

### 3.4.2.3 Synthesis

A natural correspondence exists between the operations of the specification and the hardware components. If a similar correlation can be found between iterative specifications and signal expressions, the synthesis of hardware can be accomplished by translating iterative specifications to corresponding signal expressions, which can then be directly mapped to hardware. Such a relation between the iterative specification and signal expression was formalized by Johnson [59] by the following generalization:

The iterative specification:

\[
F(x_1, \ldots, x_n) \leftarrow p \rightarrow r, F(t_1, \ldots, t_n)
\]

(read as a function \(F(x_1, \ldots, x_n)\) is defined as “if \(p\) then \(r\) else \(F(t_1, \ldots, t_n)\)”)

where \(p\), \(r\) and \(t_i\) are trivial expressions (no recursive calls), is realized by the signal equations:
\[ X_1 = x_1 \cdot t_1 \\
  \vdots \\
 X_n = x_n \cdot t_n \]

\[ \text{Ready} = p \]

\[ \text{Value} = r \]

That is, if the registers that produce signals \( X_1, \ldots, X_n \) are initialized with \( x_1, \ldots, x_n \) respectively, \( \text{Value} \) will contain \( F(x_1, \ldots, x_n) \) the first time \( \text{Ready} \) is true. By convention, a circuit description will be called a realization of a specification only when the realization has signals \( \text{Ready} \) and \( \text{Value} \) produce the value specified by the specification.

### 3.4.2.4 An Example

Consider the functional specification of the greatest common divisor problem given in Figure 3.5. Let \([\text{lt?}]\) and \([\text{eq?}]\) correspond to the conditionals "<" and "=" given in the specifications. Let the operation "\( x - y \)" be performed by \([\text{sub}]\). For the interior conditionals (the second and third statements which recursively call \( \text{gcd} \)) introduce the multiplexors, denoted by \([\text{mux}]\). The specification of \( \text{gcd} \) after recursion removal and rewriting in the syntax used to define the iterative specifications results in:

\[
\text{gcd} (x, y) \leftarrow \text{eq?} (x, y) \rightarrow x, \text{gcd} ([\text{mux}] ([\text{lt?}] (x, y), x, y),
\]
\[
[\text{mux}] ([\text{lt?}] (x, y), [\text{sub}] (y, x),
\]
\[
[\text{sub}] (x, y))\]

The conditional \([\text{mux}] ([\text{lt?}] (x, y))\) can be pushed into the call to \([\text{sub}]\) resulting in:

\[
\text{gcd} (x, y) \leftarrow \text{eq?} (x, y) \rightarrow x, \text{gcd} ([\text{mux}] ([\text{lt?}] (x, y), x, y),
\]
\[
[\text{sub}] ([\text{mux}] ([\text{lt?}] (x, y), y, x),
\]
\[
[\text{mux}] ([\text{lt?}] (x, y), x, y))\]

From the formalization of Johnson, the signal expressions from this iterative specification can be derived as:
Figure 3.7: Circuit realization for the greatest common divisor problem.

\[
\begin{align*}
X &= x^0 \cdot \text{mux} (\text{lt?} (X, Y), X, Y) \\
Y &= y^0 \cdot \text{sub} (\text{mux} (\text{lt?} (X, Y), X, Y), \text{mux} (\text{lt?} (X, Y), X, Y)) \\
\text{Ready} &= \text{eq?} (X, Y) \\
\text{Value} &= X
\end{align*}
\]

Identical signals can be shared to optimize the obtained solutions as:

\[
\begin{align*}
V &= \text{lt?} (X, Y) \\
U &= \text{mux} (V, X, Y) \\
X &= x^0 \cdot U \\
Y &= y^0 \cdot \text{sub} (\text{mux} (V, Y, X), U)
\end{align*}
\]

The schematic for the corresponding realization is given in Figure 3.7. By the formalizations presented in [59], this circuit is correct with respect to its specifications. Specifications were modified during the recursion removal process, but the theorem prover associated with the recursion remover asserts the correctness of the circuit.

To institute modularity in the design, a number of transformations have been proposed in [61, 62]. These transformation, called system factorizations, incorporate modularity by eliminating common sub-expressions and encapsulating subsystems. A number of working prototypes have been reported, including a VLSI garbage collector [16]. The approach to system derivation has also been applied to the derivation of control and architecture based on the SECD machine [119].
3.5 Recursion Removal

Recursion equations are of interest because of their abundant use in applicative languages. Recursion must be eliminated prior to hardware synthesis. A number of transformational approaches which aim at, in addition to recursion removal, increasing the efficiency of these specifications by transforming them into more efficient programs have been widely considered [39, 86, 87]. A formal approach to developing efficient programs from initial specifications is known as program transformation or transformational programming. Often the approach used in transformational programming is to employ a strategy to modify clear, and concise program specifications into semantically equivalent and efficient, but less clear programs. This section presents two such strategies for transformation of recursive specifications into equivalent iterative specifications.

Applicative languages support recursion in their computational model, λ-calculus. The λ-calculus captures functions in their fullest generality, providing them with the ability of self-application. At the specification level, recursion is an important conceptual tool in the design of algorithms. Many algorithms from a variety of applications can be defined clearly and succinctly using recursive forms. However this form is not the most suitable one for efficient execution on a computer. As Burstall [22] points out, there is a sharp contrast between programs written for maximum clarity and those written for tolerable efficiency. Program transformation systems provide a formal approach for recursion elimination. Unfortunately, not all recursive specifications have an equivalent iterative form. Some important theoretical results which illustrate this are discussed later on in this section.

3.5.1 Strategies for Recursion Removal

This subsection presents two useful strategies for recursion removal: generalization of the function definition and inverting the flow of computation. The first approach is one of the earliest and most extensively studied. Variations of the key idea presented in this approach are presented elsewhere [2, 3]. Wand [118] presents the idea of use
of continuations for generalizations.

3.5.1.1 Generalization of the Function Definition

The transformation system consists of a set of transformation rules, together with a strategy for applying them. The transformation rules replace one segment of a program description by another equivalent description. The strategy for recursion removal consists of identifying a new function \( g \) or "eureka" [23] for a recursive function \( f \) such that \( g \) "generalizes" \( f \) and \( g \) can be transformed into an iterative program. The generalization step is analogous to an induction hypothesis used in proving a program or theorem by mathematical induction. Thus in program transformation, induction is analogous to recursion. Various rules for transforming recursion equations, such as folding, unfolding, and instantiation, with other algebraic laws for the operations are used to convert a set of recursion equations into an iterative program.

The concepts used in such transformational systems are illustrated using a simple example – the factorial function defined in Figure 3.8(a). The transformation system converts the recursive form to an equivalent iterative form. A function \( f_i \) is said to be in iterative form if for each equation \( f_i(x_1, \ldots, x_n) = E \), either \( E \) does not contain \( f_i \) or if it is of the form \( f_i(E_1, \ldots, E_n) \) and \( E_1, \ldots, E_n \) do not contain \( f_i \).

The translation into an iterative scheme is obtained by following steps:

1. Introduce a new function \( g \) such that:

   \[ g(n-1) \ u = u \times \text{fac}(n-1) \quad \text{"Eureka"} \]

2. \[ g \ 0 \ u = u \]
   instantiate "0" for "(n-1)"
   and unfold using "fac 0"

3. \[ g \ n \ u = u \times \text{fac} \ n \]
   instantiate
   \[ = u \times n \times \text{fac}(n-1) \]
   unfold
   \[ = g(n-1)(u \times n) \]
   fold with (1)

4. \[ \text{fac} \ n = g(n-1) \ n \]
   fold "fac n" with (3)

Figure 3.8(b) gives the final transformed definition of the factorial function. Several automated systems that perform this type of transformation have been reported
fac 0 = 1
fac n = n * fac (n-1)

(a)

fac 0 = 1
fac n = g (n-1) n

g 0 u = u
gn u = g (n-1) (u*n)

(b)

Figure 3.8: Factorial function definition

[23, 35, 71]. Little is said about how the function g can be invented. It may be difficult to find the “eureka”, the right generalization needed by these systems.

3.5.1.2 Inverting the Flow of Computation

“Computational inversion” or “inversion techniques” reverse the order in which the computations are performed. The transformations result in functions that use the same arguments but evaluate them in inverted order [13]. For a function \( f \) which is defined by \( f(k(x)) \), the evaluation of the call \( f(x_0) \) causes a number of recursive calls \( f(x_1), f(x_2), \ldots, f(x_n) \) where \( x_{i+1} = k(x_i) \). The goal of the transformation scheme is to find function \( f' \) such that the sequence of calls of \( f' \) for computing \( f(x_0) \) is \( f'(x_n), f'(x_{n-1}), \ldots, f'(x_0) \).

As an example consider the definition of factorial given in Figure 3.8(a). In the course of evaluation of “fac(N)”, a sequence of arguments \( n_0 = N, n_1 = (N - 1), \ldots, n_N = 0 \) need to be evaluated. In the inverted computation of factorial function for argument \( N \), it is clear that it should start with argument 0. If \( N \) is also 0, then 1 should be returned, else the next argument should be 1. This process should repeat until the sequence reaches \( N \). An inverted factorial function can be rewritten as shown in Figure 3.9.

Computational inversion techniques result in a tail-recursive structure. A function \( f_i \) is said to be in tail recursive form if for each equation \( f_i(x_1, \ldots, x_n) = E \), either \( E \) does not contain \( f_i \) or it contains at most one call to \( f_i \). Such recursion
equations can be easily transformed to equivalent iterative ones. Computational inversion transformations also enable optimizations like finite differencing [83]. However, computational inversion requires strong applicability conditions (e.g. in the factorial example it was implicitly assumed that computation terminates with “fac 0”, the next argument in the inverted computation is determined by inverting the operation “-” to “+”, i.e. operations be “invertible”) and thus may not be applicable in most cases.

In addition to recursion removal strategies, a number of approaches have been considered to improve the efficiency of recursive programs. These include tabulation techniques [10, 11], and sharing the results of computation [9]. A host of other optimizations which aim to improve the recursion equations and applicative programs in general are discussed in [86].

3.5.2 Theoretical Results

There has been considerable theoretical investigation into how to translate a recursive program into an equivalent iterative program (e.g. [41, 104]). These studies apply to translations of a “schema” which preserves its effect for all “interpretations” of the primitives. A “schema” is a purely syntactic object which contains an “abstract” symbol for each of functions, predicates and constants. An “interpretation” defines some data domain and some interpretation of the symbols as mappings on this domain. The recursively defined factorial function in Figure 3.8(a) might be investigated in terms of a “recursion schema” as:

\[ f(x) = \begin{cases} 
    c & \text{if } p(x) \\
    m(n, f(d(x))) & \text{otherwise}
\end{cases} \]

Figure 3.9: Inverted factorial function.
The theme of all program transformation systems is to translate one set of program schemes into an equivalent scheme in another class. This is essentially the translatability theorem: any scheme in one class can be translated into an equivalent scheme in another class. The translation essentially requires the knowledge to replace certain features of programs by others. The translation of a recursive scheme into an equivalent iterative scheme is known as flowchartability, where the recursion is said to be "flowchartable" only if there is an iterative scheme that would be functionally equivalent under all interpretations of the primitive function, predicate and constant symbols [117]. Non-flowchartability means there does not exist an equivalent iterative scheme, thus requiring access to unbounded number of data locations (stack) during the computation of the function.

At this juncture, a natural question to arise is: given a recursive scheme does there exist an equivalent iterative scheme? A summary of results, as described in [44], are presented as a response to the above question:

- All iterative schemes can be translated into recursive schemes.
- There exists certain recursive schemes that are not translatable into iterative schemes.
- It is undecidable whether any particular recursion scheme is translatable into an iterative scheme.

Given a recursion equation the general question of the existence of an equivalent iterative program is undecidable and there may not exist an equivalent iterative program. Thus most transformation systems can derive an iterative program for only a subset of recursion equations, most notably "linear" recursions, which are guaranteed to have an equivalent iterative program. A recursion equation is said to be linear if it has at most one user defined function symbol in each of its defining equations. The factorial function is a typical example of a linear recursion equation.

The approach chosen in this research does not make use of the techniques surveyed, but uses an alternate approach to finding fixed points of recursion equations.
to derive digital circuit descriptions. The following chapter defines the fixed points of a recursion equation and presents the techniques to compute those fixed points.

3.6 Concluding Remarks

Functional languages are based on solid mathematical foundations, with simple and elegant semantics. These features make them attractive for hardware description languages. Recursive specifications in functional languages cannot be mapped directly to hardware. The transformational approach to recursion elimination is constrained by the theoretical limitations of recursion equations and by the need for "eureka". This research instead considers recursion removal from a different perspective – that of domain theory.
Chapter 4

Fixpoints of Recursion Equations

The previous chapter has emphasized the importance of recursive specifications in the functional programming paradigm, and the need for eliminating recursion prior to hardware synthesis. However, the theoretical results of flowchartability of recursion equations tend to deny formally-based efforts to find equivalent iterative specifications for a wide class of recursion equations. Practical concerns force us to ask whether it is possible to find useful, non-recursive implementations. These limitations hinder the use of powerful and elegant functional notations for hardware specification and synthesis.

A new perspective on the utility of such definitions is provided by recursive function theory [47]. A theorem due to Kleene is useful:

**Theorem 4.1** Let \( f : X \to X \), be a recursive function; then there exists an \( x \in X \) such that:

\[
\varphi_x = \varphi_{f(x)}
\]

where \( \varphi_x \) is a partial function and \( x \) is called an index or Gödel number for \( \varphi_x \). \( x \) is called the fixpoint for \( f \).

This theorem provides a new way to interpret recursive definitions – as fixpoint equations. The fixpoints of interest are fixpoints of high-level functionals, which take functions as arguments. The solutions of fixpoint equations result in functions “equal” to the original functions but without the recursion. Domain theory [103] provides a way of computing the solution to fixpoint equations as the least upper
bound of a sequence of approximations to the recursive function. This approach was used in *frontiers algorithms* [30, 64] to compute the fixpoints of recursion equations for use in abstract interpretation [1, 21].

The rest of this chapter considers the following issues:

- how recursion equations can be interpreted as fixpoint equations, and
- solving fixpoint equations.

### 4.1 Recursion Equations as Fixpoint Equations

The recursion theorem enables one to interpret recursion equations as fixpoint equations of the form:

\[ x = \tau(x) \]

As an example, consider a function defined as:

\[ \text{square} = \lambda x. x^2 \]

The fixpoint equation is of the form:

\[ x = \text{square} (x) \]

The solutions to this equation are: \( x = 0 \) and \( x = 1 \). Thus the function “square” has (at least) two fixpoints.

The fixpoints of interest are fixpoints of high-level functions. In finding the meaning of a function \( \tau \) of type \((\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \beta)\) defined as:

\[ \tau = \lambda x_1. \ldots. x_n. \text{body} \]

a new function \( \phi \) of type \((\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \beta) \rightarrow (\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \beta)\) is defined such that:

\[ \phi = \lambda \tau. x_1. \ldots. x_n. \text{body} \]

resulting in a fixpoint equation of the form:
\[ \tau = \phi(\tau) \]

The solution of this fixpoint equation results in a function "equal" to the original function \( \tau \), but without the recursion. The term "equal" here means that they are defined for the same arguments and yield the same values for all arguments. Recalling that the function "square" has two fixpoints, the natural question is: is it possible to have more than one fixpoint for high-level functionals such as \( \phi \)? The answer is yes. The problem now arises as to which of these fixpoints is of actual interest. The least fixpoint theorem [78] states that only the most partial fixpoint is of interest. The "most partial" function is taken as the meaning of the recursive definition. In other words, the "most partial" function equivalent to the recursive definition is the one that does not imply anything that is not defined in the original function.

The solution of the fixpoint equation for \( \phi \) can be computed as a limit of a sequence of approximations to the function \( \tau \) as \( \tau_0, \tau_1, \ldots \), starting with the function that maps every argument to the bottom (\( \bot \)) value of the domain. The series of approximations generated as:

\[
\begin{align*}
\tau_0 & = \bot \\
\tau_1 & = \phi(\tau_0) \\
\tau_2 & = \phi(\tau_1) \\
& \vdots
\end{align*}
\]

until the series converges such that:

\[ \tau_i = \tau_{i+1} \]

\( \tau_i \) is the least fixpoint and the sequence of approximations form the Ascending Kleene Chain (AKC). Each approximation leads a step closer to the solution of the fixpoint equation.
4.2 Inherent Problem in Computing Fixpoints

Though domain theory provides a clean and elegant solution to a difficult problem, it is at the expense of requiring a solution to an NP-hard problem, the test for equivalence of functions. With the AKC-based approach, the convergence of the series of approximations is determined by the equality of functions. To compare two functions for equality it is necessary to check that the functions coincide at every point in their domain. A naive approach to performing such comparisons evaluates both functions at every point in their domain. For a \( p \)-point domain, a function of \( n \) arguments requires \( p^n \) evaluations.

In order to support the domain theoretic approach for recursion removal, one needs schemes for representing functions that are as succinct as possible and at the same time support efficient solutions of the basic tasks in manipulation, especially

- efficient algorithms for deriving the representations of the functions, and

- efficient algorithms to perform equivalence tests or other similar tests in terms of these representations.

4.3 A Solution – Frontiers Approach

The frontiers algorithm finds fixpoints using an iterative approach to find the fix-points of monotonic functions in an abstract domain.

**Definition 4.1** A function \( f \) is said to be monotonic if for an argument domain \( A \):

\[
x \leq y \implies f(x) \leq f(y) \quad \forall (x, y) \in A
\]

The domains are (may be) partially ordered. For a set \( S \), a relation on \( S \) is a subset of \( S \times \ldots \times S \). If \( R \subseteq S \times S \) is a binary relation, then the elements of \( R \) are ordered pairs of the form \( (x, y) \), where \( x \) and \( y \) are elements of \( S \). \( R \) is said to be a partial order relation on \( S \) if three conditions are satisfied:

1. \( (x, x) \in R \) for any \( x \in S \).
2. If \((x, y) \in R\) and \((y, x) \in R\), then \(a = b\).

3. If \((x, y) \in R\) and \((y, z) \in R\), then \((x, z) \in R\).

These three conditions are known as reflexivity, antisymmetry, and transitivity, respectively. It is written as \(x \leq y\) to indicate that \((x, y) \in R\).

**Definition 4.2** For all \(x, y \in S\), if \(x \leq y\) or \(y \leq x\), then \(S\) is a totally ordered set.

The rest of this section outlines the frontiers algorithm of [30, 64].

### 4.3.1 Motivation

Abstract interpretation [1, 21] is a semantics-based approach for compile time analysis of programs using simplified value domains. One of the important applications of abstract interpretation is strictness analysis, which determines if the function is strict in a particular argument, i.e., if the value of an argument is needed for the evaluation of the function. Strictness analysis typically uses domain 2 \((0, 1)\). Arithmetic operators such as +, -, *, and / that are strict in both the arguments, in the abstract domain become boolean and, constants have an abstract value 1. Similarly conditional statements have a mapping to the abstract domain. For example, an “if \(e_1\) then \(e_2\) else \(e_3\)” statement has an abstract interpretation:

\[ e_1 \text{ and } (e_2 \text{ or } e_3) \]

where \(e_1\), \(e_2\), and \(e_3\) are some expressions, and and or are boolean “and” and “or” operations.

As an example, consider the (iterative) definition of a factorial function given as:

\[ \text{fac} = \lambda x. y. \text{if } x = 0 \text{ then } y \text{ else } \text{fac} (x - 1) (x \ast y) \]

This becomes the abstract function \(\text{fac}\#\):

\[ \text{fac}\# = \lambda x. y. (x \text{ and } 1) \text{ and } (y \text{ or } \text{fac}\# (x \text{ and } 1) (x \text{ and } y) \]

which can be simplified to:
fac# = \lambda x.y.x and (y or fac# x (x and y)

In order to determine if function fac is strict in its second argument, it suffices to show that if "fac# 1 0 = 0", then fac is indeed strict in its second argument [64]. Though it is obvious that fac is strict in both its arguments, a straight forward evaluation of its abstract function leads to the computation:

\[
\text{fac# 1 0} = 1 \text{ and } (0 \text{ or } \text{fac# 1 0}) \\
= 1 \text{ and } (0 \text{ or } (1 \text{ and } (0 \text{ or } \text{fac# 1 0}))) \\
= \ldots
\]

Therefore it is essential that the least fixpoint of the abstract function be found, else the abstract interpreter will fail to terminate.

4.3.2 Frontiers

The set of all possible arguments (represented as tuples) to a function in an abstract domain have a pointwise ordering:

\[
< x_1, x_2, \ldots, x_n > \leq < y_1, y_2, \ldots, y_n > \quad \text{iff} \quad x_i \leq y_i \forall i : 1 \leq i \leq n
\]

A set for which the above relationship is defined for all pairs is a totally ordered set. Such a totally ordered set is represented as a chain [103].

Definition 4.3 If \( R \) is a subset of \( L \) and \( u \) is an element of \( L \) such that \( r \leq u \) for any \( r \in R \), then \( u \) is said to the an upper bound for \( R \). Moreover, \( u \) in \( L \) is called the least upper bound of \( R \) if \( u \) is the smallest among all upper bounds for \( R \).

Definition 4.4 \( l \) is a lower bound for a subset of \( R \) of the partially ordered set \( L \) if \( l \leq r \) for any \( r \in R \). It is called greatest lower bound, for any other lower bound \( l' \), \( l \geq l' \).

Definition 4.5 A partially ordered set is called a lattice if every subset \( R \) consisting of two points possesses both a least upper bound and a greatest lower bound.
In general, every totally ordered set is a lattice. Directed graphs are useful in representing these lattices. Each element in the set denotes a node in the graph, and whenever the relation $\rho$ exists such that $a \rho b$, a line is drawn between $a$ and $b$. $a$ is positioned beneath $b$ to indicate the ordering.

As an example, consider a domain $T (0, 1)$. For a function of two arguments, ordering on the domain of possible arguments $T^2 = \{<0, 0>, <0, 1>, <1, 0>, <1, 1>\}$ results in two totally ordered subsets $\{\{<0, 0>, <0, 1>, <1, 1>\}, \{<0, 0>, <1, 0>, <1, 1>\}\}$. Figure 4.1 depicts the graphical representation of the lattice. Such a lattice represents a function if each node is labeled with the value of the function at that point.

Consider the function:

$$f = \lambda x.y. (x \text{ and } y)$$

The labeling with values of $f$ is shown in square brackets in Figure 4.1. All the nodes labeled by a 0 (1) are referred as 0-nodes (1-nodes). A deeper examination of the lattice reveals the fact that for any monotonic function, all 1-nodes will always be above the 0-nodes. A much more compact representation of a function can thus be obtained using this information, by using only those nodes that lie on the boundary between the nodes labeled with different values. The set of nodes that lie on the boundary leads to the idea of a frontier. Obviously, it is possible to have two types of sets, the set of 0-nodes as the boundary to form a 0-frontier and the set of 1-nodes which form the boundary as 1-frontier.
It is desirable that these frontier sets be as small as possible. Minimization of the set is accomplished by eliminating redundant elements in the set. Two elements in a frontier set are said to be redundant if they are comparable to each other, i.e. there exists an edge in the lattice between two nodes. Figure 4.2 illustrates the idea. Though the nodes $<0, 0, 1, 1>$, $<0, 1, 1, 0>$, $<1, 0, 1, 0>$ lie on the frontier, they are redundant with respect to $<0, 0, 1, 0>$. The irredundant 1-frontier set is $<0, 0, 1, 0>$. "Frontier set" is used to refer to either an irredundant 0-frontier set or an irredundant 1-frontier set and is explicitly specified as where required.

A frontier thus provides a compact, canonical representation for a function. The problem of equality of two functions is now reduced to that of comparing two frontiers which immediately leads to the following theorem [64]:

**Theorem 4.2** Two functions with equal frontier sets are equal.

### 4.3.3 Frontiers Algorithm

Frontiers algorithms find fixpoints by iterating to find successive approximations to the function. A fixpoint is found when two successive approximations to the original function are found to be equal, the equality measure based on the comparison of their frontier sets.
For each approximation in the AKC, a frontier set is computed by evaluating the current approximation of the function. To minimize the number of evaluations, only 1-nodes (0-nodes) need to be expanded to compute 0-frontier (1-frontier), starting from the top element in the lattice. This process of exploring the lattice continues until no 1-nodes are found or the lattice has been completely explored. A set, initialized to an empty set, is maintained to which the frontier nodes are added as they become available, e.g. if a 0-node is encountered during the top-down traversal of the lattice, this can be added to the 0-frontier set as all its descendants below it are guaranteed to be zero since the function is monotonic. This process of computing the frontiers continues until two successive frontier sets are found to be equal, at which point the current approximation determines the fixpoint of the defining function.

As an example, consider the abstract function definition of the factorial function given before. The sequence of approximations can be derived as:
\[
\begin{align*}
\text{fac}_0 & = 0 \\
\text{fac}_1 & = x \text{ and } (y \text{ or } \text{fac}_0 \times (x \text{ and } y)) \\
& = x \text{ and } (y \text{ or } 0) \\
& = x \text{ and } y \\
\text{fac}_2 & = x \text{ and } (y \text{ or } \text{fac}_1 \times (x \text{ and } y)) \\
& = x \text{ and } (y \text{ or } (x \text{ and } y)) \\
& = x \text{ and } y
\end{align*}
\]

The frontiers for these expressions are shown in Figure 4.3. The approximation "x and y" is the fixpoint for the abstract factorial function and it indicates the fact that the factorial function is strict in both its arguments.

Since the concept of frontiers has first appeared in the literature [30], it has evolved as one of the standard approaches for computing the fixpoints in abstract interpretation [20]. A number of techniques which aim at optimizing the search are presented in [64, 121]. The algorithm was applicable only to first-order functions over a flat domains. Martin and Hankin [73] extended this basic idea to include higher-order functions. Hunt [54] and later Hunt and Hankin [55] present the underlying theory and generalized the algorithm for a rich-class of finite lattices and higher-order functions. Another approach has been considered by McCrosky and Wang [75]. Their method is based on computing the frontiers from the abstract syntax of the defining function without evaluating the function at all. Unlike other approaches, which are based on expensive search-based strategies, substantial improvements in performance are gained by computing the frontiers from the abstract syntax. The extension of their approach to cope with higher-order functions is being considered.

4.4 Concluding Remarks

Frontiers algorithms provide a constructive way to find non-recursive equivalents for recursive specifications. This can be exploited to synthesize hardware from recursive specifications without a need for "eurekas". 

58
Chapter 5

Specification Languages

High-level synthesis is an effective tool for managing the complexity of hardware specification and design. Synthesis tools use hardware description languages to represent the circuit behavior or structure. These languages permit formal design specification at an algorithmic level without going into the details of the implementation. It is desirable that such formal specifications be lucid so that it is easy for a designer to comprehend the specifications. Chapter 3 has demonstrated how the functional programming paradigm could lend itself to digital system specification. The research described here adopts this point of view. This thesis describes a method based on the functional programming paradigm for the specification of algorithms and their digital hardware synthesis.

This chapter begins with an introduction to FHDL (Functional Hardware Description Language), a functional programming based specification language, for formal specification of digital circuits. It is imperative that any language used for formal specification have its formal semantics defined. In this spirit, the denotational semantics of FHDL are presented.

Specifications in FHDL are abstract – their main purpose is to convey the intent of the designer. The abstract specifications are transformed into an intermediate language called the Kernel Language (KL). KL is defined on domain 2 with and, or and not as its primitive operations. The kernel language is an intermediate step towards the final synthesis. The transformation to KL has the following effects on the original FHDL specifications:
• FHDL specifications are *split* to their bit-level equivalent KL specifications.
  
  – All arithmetic and predicate operations are replaced by their corresponding gate-level descriptions using only boolean *and*, *or* and *not*.
  
  – All constants are replaced by their binary equivalents.
  
  – All function names and variable names are tagged with a bit-number for all their *n* bits.

• The original recursive structure of the specification is preserved.

A function, *f*, in FHDL is *split* into *n* functions in KL, where *n* is number of bits used for representation of the result of FHDL specification. KL also provides a formal means to specify all high level arithmetic and predicate functions at an implementation level. Due to the significance of KL in the synthesis methodology, this chapter also presents the formal semantics of KL. The reasons for splitting the original specifications are discussed informally in this chapter. Recursion needs to be eliminated prior to hardware derivation. Formal reasoning for transformations on FHDL and recursion removal from KL are adjourned until the next chapter. This chapter concludes with a formal translation scheme from FHDL to KL.

### 5.1 Introduction to FHDL

The specification language FHDL is a simple high-level functional language used to specify algorithms intended for hardware implementation. A program in FHDL is an expression that maps objects into objects. The objects can be atomic (e.g. numbers) or sequences (e.g lists). This chapter considers only atomic objects. Lists aid in the specification of sequential circuits (finite state machines). Extensions to the language to include sequences of objects are discussed along with sequential circuit synthesis in the following chapters.

Undefined values in FHDL are denoted by ⊥ (read as 'bottom'). By definition, any expression which contains ⊥ as an element is itself undefined and is equal
to \bot. This means that if any argument to a function is undefined, the result of evaluating the function is also undefined. Thus the following identity is implicit in the denotational definition of the language:

$$\forall f, \ f \bot \equiv \bot$$

where \( f \) is a defined function.

This implies that all the functions defined in FHDL be total and the language has strict semantics. Some functional languages (e.g. Miranda) have non-strict or lazy semantics. Lazy semantics aid the use of incomplete and possibly infinite data structures which can be useful in some computations [53, 110]. Lazy semantics do not, however, seem to be useful for circuit synthesis.

The primitive functions of FHDL consist of arithmetic, logic and relational functions. Figure 5.1 shows the primitive functions of FHDL along with their type information. Type \( \text{Int} \) represents integers and \( \text{Bool} \) represents boolean values.

Conditional expressions are introduced in the form of pattern matching on the input parameters. The standard \( \text{if} \ldots \text{then} \ldots \text{else} \) form of conditional expressions is supported by guarded expressions, where the guard determines the branching control. The FHDL compiler transforms pattern matching expressions into their equivalent guarded expressions during the compilation process. Thus the two styles of definition are considered equal. To illustrate the point, two equivalent definitions of the factorial function are given in Figure 5.2.

A program in FHDL consists of a set of function definitions, together with an expression to be synthesized. The expression can be a simple function name. It is required that all the functions be well-typed. Function definitions are allowed to be recursive and a function can be defined in terms of other functions. If pattern matching is used for case analysis, the different cases are grouped together. The only restrictions that apply for function definitions with parameters are:

- The parameters are simple, i.e. expressions are not allowed as parameters on the left hand of the defining equations. This restriction was chosen to simplify
Arithmetic functions:

+ :: Int -> Int -> Int addition
- :: Int -> Int -> Int subtraction
* :: Int -> Int -> Int multiplication
/ :: Int -> Int -> Int division

Predicate functions:

< :: Int -> Int -> Bool less than
> :: Int -> Int -> Bool greater than
eq :: Int -> Int -> Bool equals
le :: Int -> Int -> Bool less than or equal
ge :: Int -> Int -> Bool greater than or equal
ne :: Int -> Int -> Bool not equal

Logic functions:

and :: Bool -> Bool -> Bool logical and
or :: Bool -> Bool -> Bool logical or
not :: Bool -> Bool -> Bool logical not

Figure 5.1: Primitive functions of FHDL.

factorial 0 = 1
factorial n = n * (n - 1)

(a)

factorial n = 1, n = 0
= n * (n - 1), otherwise

(b)

Figure 5.2: Two equivalent definitions of factorial function in FHDL.

the implementation process. Under this restriction, it is easy to transform a
function definition with parameters into one without parameters.

- In the case of function definitions with pattern matching, the order of equa-
tions is important. Pattern matching proceeds from top to bottom and the
expression corresponding to the first found match is evaluated.

The abstract syntax of the specification language is defined below. The notation
[\alpha]^\beta \text{ represents zero or more occurrences of } \alpha \text{ separated by } \beta. \text{ The tokens } "<\text{ident}>" \text{ and } "<\text{constant}>" \text{ in the abstract syntax denote identifiers and integer constants respectively. The symbols underlined in the abstract syntax are the syntactic symbols that are added to the language.}

\[
<\text{program}> ::= \text{Synthesize} \ <\text{expr}> \ with \\
[ <\text{defn}> ]_{\text{new line}} \tag{5.1}
\]

\[
<\text{defn}> ::= <\text{call-pattern}> \equiv <\text{expr}> \tag{5.2}
\]

\[
<\text{call-pattern}> ::= <\text{ident}> [ <\text{atom}> ]_{\text{blank space}} \tag{5.3}
\]

\[
<\text{expr}> ::= <\text{atom}> | <\text{conditional}> | <\text{application}> | \\
\langle <\text{expr}> \rangle \tag{5.4}
\]

\[
<\text{atom}> ::= <\text{constant}> | <\text{ident}> \tag{5.5}
\]

\[
<\text{conditional}> ::= <\text{expr}> , <\text{expr}> , <\text{expr}> , \text{otherwise} \tag{5.6}
\]

\[
<\text{application}> ::= <\text{expr}> <\text{expr}> \tag{5.7}
\]

Figure 5.3(a) illustrates typical function definitions in FHDL. In the case of synthesis of functions from a program with multiple definitions, the order in which function definitions occur does not effect the synthesized hardware. Dependency analysis, a topological sort of functions based on the function to be synthesized is performed to determine:

- the partial information on the order in which functions are called, and
- the function definitions needed for the hardware synthesis.

In Figure 5.3(b), the dependency analysis determines that the only function required for the hardware synthesis is \textit{sum}, in addition to the primitive functions "+" and "-". The results of dependency analysis along with the function definitions form the input to the FHDL to KL translator.
5.1.1 Formal Semantics of FHDL

The semantic domain of FHDL, $V$, contains data as well as functions. If $Int$ represents the domain of all integer numbers and $Bool$ represents the domain of boolean values, then $V$ is:

$$ V = Int + Bool + (V \rightarrow V) $$  \hspace{1cm} (5.8)

where $\rightarrow$ is the function constructor.

The denotational equations for FHDL are given below. $\rho$ denotes the environment which holds the attributes associated with the identifiers and the operator function symbols (grouped as identifiers in the syntactic domain) and $K$ maps the constants to their corresponding values. Identifiers evaluate to their binding in the current environment. For $true, false \in Bool$ and $m, n \in V$ the choice function is defined as:

$$ (* \rightarrow * \square *) :: Bool \rightarrow V \rightarrow V \rightarrow V $$

$$ (true \rightarrow m \square n) = m $$

Figure 5.3: Typical definitions in FHDL.

<table>
<thead>
<tr>
<th>synthesize sum with</th>
<th>synthesize sum with</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sum } 0 = 0$</td>
<td>$\text{sum } 0 = 0$</td>
</tr>
<tr>
<td>$\text{sum } n = n + \text{sum } (n-1)$</td>
<td>$\text{sum } n = n + \text{sum } (n-1)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>synthesize fib with</th>
<th>synthesize fib with</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{fib } n = 0, n = 0$</td>
<td>$\text{fib } n = 0, n = 0$</td>
</tr>
<tr>
<td>$\text{fib } n = 1, n = 1$</td>
<td>$\text{fib } n = 1, n = 1$</td>
</tr>
<tr>
<td>$\text{fib } n = \text{fib } (n-1) + \text{fib } (n-2)$</td>
<td>$\text{fib } n = \text{fib } (n-1) + \text{fib } (n-2)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>synthesize gcd with</th>
<th>synthesize gcd with</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{gcd } a b = \text{gcd } (a-b) b, a &gt; b$</td>
<td>$\text{gcd } a b = \text{gcd } (a-b) b, a &gt; b$</td>
</tr>
<tr>
<td>$\text{gcd } a (b-a), b &gt; a$</td>
<td>$\text{gcd } a (b-a), b &gt; a$</td>
</tr>
<tr>
<td>$\text{gcd } a, \text{otherwise}$</td>
<td>$\text{gcd } a, \text{otherwise}$</td>
</tr>
</tbody>
</table>
\[(\text{false} \rightarrow m \odot n) = n\]

**Syntactic Domain:**

\[
\begin{align*}
\text{id} & \in I = \text{Identifiers} \quad (5.9) \\
\text{c} & \in C = \text{Constants} \quad (5.10) \\
\text{e} & \in E = \text{Expressions} \quad (5.11) \\
\text{d} & \in D = \text{Function definitions} \quad (5.12) \\
\text{p} & \in P = \text{Synthesize } e_i \text{ with } d_0 \ldots d_{n-1} \quad (5.13)
\end{align*}
\]

**Primitive Domain Functions:**

\[
\begin{align*}
\rho & : I \rightarrow V \quad (5.14) \\
\text{K} & : C \rightarrow V \quad (5.15)
\end{align*}
\]

**Valuation Functions:**

\[
\begin{align*}
\text{I} & : I \rightarrow \rho \rightarrow V \quad (5.16) \\
\text{I}[\text{id}] \rho & = \rho[\text{id}] \quad (5.17) \\
\text{C} & : C \rightarrow \rho \rightarrow V \quad (5.18) \\
\text{C}[\text{c}] \rho & = \text{K}[\text{c}] \quad (5.19) \\
\text{D} & : D \rightarrow \rho \rightarrow \rho \quad (5.20) \\
\text{D}[\text{id} \, \text{id}_0 \ldots \text{id}_n = e_1] \rho & = \lambda \rho'.\rho[\lambda \text{id}_0.\ldots.\lambda \text{id}_n.\text{E}[e_1] \rho' / \text{id}] \quad (5.21) \\
\text{E} & : E \rightarrow \rho \rightarrow V \quad (5.22) \\
\text{E}[\text{c}] \rho & = \text{C}[\text{c}] \rho \quad (5.23) \\
\text{E}[\text{id}] \rho & = \text{I}[\text{id}] \rho \quad (5.24) \\
\text{E}[e_0 \, e_1] \rho & = (\text{E}[e_0] \rho) (\text{E}[e_1] \rho) \quad (5.25) \\
\text{E}[e_0 \, e_1] \rho & = \text{E}[e_0 \, e_1] \rho \quad (5.26)
\end{align*}
\]

65
\[
\mathcal{E} \left[ \begin{array}{c}
e_0, \hat{e}_0 \\
\vdots \\
e_{n-1}, \text{otherwise} \end{array} \right] \quad \mathcal{E} [\hat{e}_0] \rho \rightarrow \mathcal{E} [e_0] \rho \quad \Box \tag{5.27}
\]
\[
\mathcal{P} :: P \rightarrow \rho \rightarrow V \quad \Box \tag{5.28}
\]

The function \( \text{fix} \) is fixpoint operation. Computing fixpoints, a crucial step for hardware synthesis, and mapping these fixpoints to hardware are discussed in the next chapter.

### 5.2 The Kernel Language

Most approaches to high-level synthesis from behavioral specifications use an intermediate form to represent behavior. The intermediate form is usually a directed graph. Synthesis consists of mapping this graph into a hardware structure via scheduling and resource allocation. Other approaches (e.g. [59, 63, 88, 97]) choose the transformational approach to high-level synthesis where the specifications are refined step-wise, until they are suitable for mapping to hardware structures.

The approach presented here uses a transformational approach to synthesis. Starting from a high-level specification in FHDL, the descriptions are transformed into more concrete descriptions. "Concrete descriptions" here means descriptions that more closely reflect the implementation. If the gate-level descriptions are chosen as implementation-level primitives, the purpose of a synthesis tool is to transform all high-level descriptions to gate-level descriptions.

The transformation presented here is based on mapping all operations (functions) to their implementation level primitives using only and, or and not operations. The reason for choosing only these operators are: (i) any logic operation can be
expressed in terms of these primitive operations, and (ii) the approach to computing
the fixpoints presented in this thesis, and hence synthesis of hardware, exploit some
of the properties exhibited by these operations to compute the fixpoints of the
functions. These properties are discussed in detail in the next chapter.

Mapping high-level operations to their corresponding implementation-level prim-
itives corresponds to the resource allocation step in other high-level synthesis systems
where a high-level primitive in the intermediate form is mapped to its hardware re-
source. The corresponding “resource” here is a set of function definitions defining
the operation at the lowest level of implementation. KL provides a formal means to
specify these functions. In summary, KL serves two important purposes:

- as an intermediary language between FHDL and the hardware synthesis algo-
  rithms, and

- as a formal means to specify high-level language primitives in terms of their
  implementation-level primitives.

A function \( f \) defined in FHDL is represented in KL as a set of functions:

\[
\langle f_n, \ldots, f_i \rangle
\]

where \( n \) is desired number of bits for input and output data representation and each
\( f_i \) is defined on a domain with \( \text{and}, \text{or} \) and \( \text{not} \) as its operators. Under this scheme, in
addition to splitting the high-level functions of FHDL, all the constants (variables)
in a definition are replaced by their corresponding binary equivalents (equivalent
n-bit representation). Hence, for example, a function \( f \) of type:

\[
f :: \text{Int} \rightarrow \text{Int}
\]

is represented in KL as \( f_i \), corresponding to the \( i^{th} \) bit, as:

\[
f_i :: T^n \rightarrow T_i
\]

\( n \) is as defined before, domain \( T = \{0, 1\} \) and \( T^n \) is a tuple of size \( n \) such that,
for any \( f \), that translation from FHDL to KL denotes:

\[
f a \equiv \langle f_n, \ldots, f_i \rangle < a_n, \ldots, a_1 >
\]
Boolean values assume the values 1 and 0 for the true and false respectively. Figure 5.4 illustrates the geometrical interpretation of this translation.

**Proposition 5.1** Every high-level function $f$ of type:

$$f :: \alpha_0 \to \ldots \to \alpha_n \to \gamma$$

can be defined using only and, or and not logic elements as $(f_n \ a_n, \ldots, a_1), \ldots, (f_1 \ a_n, \ldots, a_1)$ for a defined value $n$. There exists $f_i$ such that:

$$\forall f_i \land 0 \leq i \leq n \implies f_i :: T^n \to \ldots \to T^n \to T_i$$

This proposition suggests that for any high-level function defined in FHDL, there exists an equivalent KL definition, and hence the translation always succeeds. The proof is trivially based on the universality of KL domain operations.
KL library functions:

\[
\text{exor } a \ b = (\ or (\text{and} (\text{not} a) b) (\text{and} a (\text{not} b)))
\]
\[
+_0 <a b> <c d> = (\text{exor } b d)
\]
\[
+_1 <a b> <c d> = (\text{exor} (\text{exor } a c) (c_1 b d))
\]
\[
c_1 a b = (\text{and } a b)
\]
\[
-0 <a b> <c d> = (\text{not} (\text{exor} (\text{not} b) d))
\]
\[
-1 <a b> <c d> = (\text{exor} (\text{exor } a c) (b_1 b d))
\]
\[
b_1 a b = (\text{and} (\text{not} a) b)
\]
\[
=1 <a b> <c d> = (\text{and} (\text{not} (\text{exor } a c)) (\text{not} (\text{exor } b d)))
\]
\[
=0 <a b> <c d> = (\text{and} (\text{not} (\text{exor } a c)) (\text{not} (\text{exor } b d)))
\]

FHDL specifications translated to KL:

\[
\text{sum}_1 <n_1 n_0>
\]
\[
= 0, (=_1 <n_1 n_0> <0 0>)
\]
\[
= (+1 <n_1 n_0> < (\text{sum}_1 <(-1 <n_1 n_0> <0 1>)
\]
\[
(-0 <n_1 n_0> <0 1>)) (\text{sum}_0 <(-1 <n_1 n_0> <0 1>)
\]
\[
(-0 <n_1 n_0> <0 1>)), \text{ otherwise}
\]

\[
\text{sum}_0 <n_1 n_0>
\]
\[
= 0, (=_0 <n_1 n_0> <0 0>)
\]
\[
= (+0 <n_1 n_0> < (\text{sum}_1 <(-1 <n_1 n_0> <0 1>)
\]
\[
(-0 <n_1 n_0> <0 1>)) (\text{sum}_0 <(-1 <n_1 n_0> <0 1>)
\]
\[
(-0 <n_1 n_0> <0 1>)), \text{ otherwise}
\]

Figure 5.5: An example of KL library and a translated definition.

At this point, it is important to note that KL functions cannot be mapped directly to hardware. The translation to KL does not effect the original structure of the definition. This means that if the original specification in FHDL is recursive, its equivalent definition in KL level is also recursive. Hardware structures can be derived only after the recursion has been eliminated from the definitions. Figure 5.5 shows a portion of translation of \textit{sum} function defined in Figure 5.3 and required KL function library.

Following the conventions used in the definition of FHDL syntax, the abstract syntax of the kernel language is defined below:
\[
\begin{align*}
< \text{program} > & ::= \text{Synthesize } \{ < \text{ident} > \}_i \text{ with } \\
& \text{[< defn >] new line} \\
< \text{defn} > & ::= < \text{call pattern} > = < \text{expression} > \\
< \text{call pattern} > & ::= < \text{ident} > [< \text{tuples} >]_{\text{blank space}} \\
< \text{expression} > & ::= < \text{tuples} > | < \text{application} > \mid < \text{conditional} > | < \text{ident} > | < \text{atom} > \\
< \text{tuples} > & ::= \leq [ < \text{atom} >]_{\text{blank space}} \geq \\
& \mid \leq [ < \text{ident} >]_{\text{blank space}} \geq \\
< \text{atom} > & ::= 0 \mid 1 \\
< \text{application} > & ::= < \text{expression} > < \text{expression} > \\
< \text{conditional} > & ::= < \text{expression} > \_ < \text{expression} > \mid < \text{expression} > , \text{ otherwise} \\
\end{align*}
\]

### 5.2.1 Formal Semantics of KL

The semantic domain of KL consists of boolean values, tuples of booleans, and functions. If \( \text{Bool} \) represents the domain of boolean values and \( \times \) the domain constructor for tuples, then semantic domain \( V \) for the kernel language is:

\[
V = \text{Bool} + (\text{Bool} \rightarrow \text{Bool}) + BT + FT
\]

\[
BT = \langle \text{Bool} \times \ldots \times \text{Bool} \rangle
\]

\[
FT = \langle (\text{Bool} \rightarrow \text{Bool}) \times \ldots \times (\text{Bool} \rightarrow \text{Bool}) \rangle
\]

\( BT \) and \( FT \) represent the domain of boolean tuples and function tuples.

The denotational equations are defined using the previous conventions used for FHDL. Identifiers evaluate to their binding in the current environment defined by \( \rho \).
Syntactic Domain:

\[
\begin{align*}
  id &\in I = \text{Identifiers} \\
  c &\in C = \text{Constants} \\
  it &\in IT = \text{Identifier tuples} \\
  ct &\in CT = \text{Constant tuples} \\
  ot &\in OT = \text{function symbol tuples} \\
  e &\in E = \text{Expressions} \\
  d &\in D = \text{Function definitions} \\
  p &\in P = \text{Synthesize } e_i \text{ with } d_0 \ldots d_{n-1}
\end{align*}
\]

Primitive Domain Functions:

\[
\begin{align*}
  \rho &:: I \to V \\
  \mathcal{K} &:: P \to \rho \to V
\end{align*}
\]

Valuations:

\[
\begin{align*}
  I &:: I \to \rho \to V \\
  I [id] \rho &= \rho [id] \\
  \mathcal{C} &:: C \to \rho \to V \\
  \mathcal{C} [0] \rho &= 0 \\
  \mathcal{C} [1] \rho &= 1 \\
  IT &:: IT \to \rho \to V \\
  IT [< it_n \ldots it_1 >] \rho &= < (I [it_n] \rho) \ldots (I [it_1] \rho) > \\
  CT &:: CT \to \rho \to V \\
  CT [< ct_n \ldots ct_1 >] \rho &= < (C [ct_n] \rho) \ldots (C [ct_1] \rho) > \\
  OT &:: OT \to \rho \to V \\
  OT [(ot_n \ldots ot_1)] \rho &= < IT (ot_n) \rho \ldots IT (ot_1) \rho >
\end{align*}
\]
\[ D :: D \to \rho \to \rho \]  

\[ D [id \ t_0 \ t_1 \ ... \ t_n = e_1] \rho = \lambda \rho'.\rho[\lambda \mathcal{E}[t_0].\lambda \mathcal{E}[t_1]. \ldots \lambda \mathcal{E}[t_n]. \mathcal{E}[e_1] \rho' / id] \]  

\[ E :: E \to \rho \to V \]  

\[ \mathcal{E} [it] \rho = \mathcal{IT} [it] \rho \]  

\[ \mathcal{E} [ct] \rho = \mathcal{CT} [ct] \rho \]  

\[ \mathcal{E} [id] \rho = \mathcal{I} [id] \rho \]  

\[ \mathcal{E} [c] \rho = \mathcal{C} [c] \rho \]  

\[ \mathcal{E} [e_0 \ e_1] \rho = (\mathcal{E} [e_0] \rho) (\mathcal{E} [e_1] \rho) \]  

\[ \mathcal{E} [\lambda \it e] \rho = \lambda v.\mathcal{E} [e] \rho[ v_i / it_i ] \]  

\[ \mathcal{E} \left[ \begin{array}{c} e_0, \hat{e}_0 \\ \vdots \\ e_{n-1}, \text{otherwise} \end{array} \right] \rho = \begin{cases} \mathcal{E} [\hat{e}_0] \rho \to \mathcal{E} [e_0] \rho & \square \\ \ldots & \square \end{cases} \]  

\[ \mathcal{E} \left[ \begin{array}{c} e_0, \hat{e}_0 \\ \vdots \\ e_{n-1}, \text{otherwise} \end{array} \right] \rho = \begin{cases} \mathcal{E} [\hat{e}_0] \rho \to \mathcal{E} [e_0] \rho & \square \\ \ldots & \square \\ \text{true} \to \mathcal{E} [e_{n-1}] \rho & \end{cases} \]  

\[ P :: P \to \rho \to V \]  

\[ \mathcal{P} \left[ \begin{array}{c} \text{Synthesize ot with} \\ d_0 \\ \vdots \\ d_{n-1} \end{array} \right] \rho = \begin{cases} \mathcal{OT} [ot] \rho_e \text{ where} \\ \rho_e = \text{fix } \lambda \rho_e.\rho[ (\mathcal{D} [d_0] \rho_e), \ldots, (\mathcal{D} [d_{n-1}] \rho_e)] & \end{cases} \]  

The notation used in Equation 5.70 follows the standard convention extended to tuples which means the expression \( e \) with \( v_i \) substituted for free occurrences of \( it_i \) where \( i \) is the index variable of the tuple.

### 5.3 Translation Scheme

The translation from FHDL to the kernel language is accomplished with a set of straightforward transformation rules. Let \( TS \) denote the translation scheme. The following equations illustrate the effect of \( TS \) on FHDL. Let \( e, i, c, o, \) and \( d \) denote an expression, an identifier, a constant, an operator, and a definition, respectively.
Identifiers denote a user defined function symbol or a variable name. They can be distinguished from each other by the initial environment generated at the parse time. Using this information, all identifier symbols that denote a user defined function are treated as operator symbols.

$TS$ can be defined as a function:

$$TS :: FHDL \rightarrow KL \quad (5.74)$$

$$TS \left[ \begin{array}{l}
\text{Synthesize } i \text{ with } \\
\text{Synthesize } TE \ [e] \ c \text{ with } \\
d_0 \\
\vdots \\
d_{n-1}
\end{array} \right] ::= \left[ \begin{array}{l}
TD \ [d_0] \ c \\
\vdots \\
TD \ [d_{n-1}] \ c
\end{array} \right] \quad (5.75)$$

$TD$ is the translation scheme for FHDL lambda abstractions. $TE$ maps a FHDL expression to its corresponding KL expression. $c$ is the number of bits to which the function is split. Let $FHDL-DEFN$ and $KL-DEFN$ denote a definition in FHDL and KL respectively. The environment variable $\rho$ is not shown but is implicit in all definitions. The formal scheme for translating the definitions is given as:

$$TD :: FHDL-DEFN \rightarrow num \rightarrow [KL-DEFN] \quad (5.76)$$

$$TD \ [i \ e_1 \ldots e_n = e] \ c = STS[TW [(TE \ [i] \ c) \\
= \lambda (TE \ e_1 \ c) \ldots (TE \ [e_n] \ c). \\
(TE \ [e] \ c)] \ c] \quad (5.77)$$

$TE$ is the translation scheme for expressions. It is defined as:

$$TE :: FHDL-EXPR \rightarrow num \rightarrow KL-EXPR \quad (5.78)$$
The translation for various cases of expressions is effected by the following translation rules:

\[
TE [c] c = BT [c] c \\
TE [i] c = IT [i] c \\
TE [o] c = OT [o] c \\
\]

\[
TE [e_1 e_2] c = (TE [e_1] c) (TE [e_2] c)
\]

Let \text{BOOL-TUPLE}, \text{ID-TUPLE} and \text{OT-TUPLE} represent boolean tuples, identifier tuples and function tuples. The functions \text{BT}, \text{IT} and \text{OT} are tuple generator functions and have the following effect:

\[
BT :: num \rightarrow num \rightarrow BOOL-TUPLE
\]

\[
IT :: ident \rightarrow num \rightarrow ID-TUPLE
\]

\[
OT :: operator \rightarrow num \rightarrow OP-TUPLE
\]

For example, a function call to \text{BT} can be illustrated as:

\[
BT 5 4 = <0 1 0 1>
\]

and is written as:

\[
BT [c] c = <b_0 \ldots b_1> where \ b_0, \ldots, b_1 \in \{0, 1\}
\]

An example of \text{IT} can be given as:

\[
IT x 4 = <x_4 x_3 x_2 x_1>
\]

where \text{IT} is defined as:

\[
IT [i] c = <i_0 \ldots i_1>
\]
Operator symbols, which include user defined function symbols and primitive operation symbols follow a similar strategy. The function $OT$ is defined as:

$$OT\, [o]\, c = \langle o_c \ldots o_1 \rangle$$

(5.88)

For example, let $test$ and "+" be the operation symbols encountered by the translator. The call to $OT$ translator has the following result on these symbols:

$$OT\, test\, 4 = \langle test_4\, test_3\, test_2\, test_1 \rangle$$

$$OT\, +\, 4 = \langle +_4\, +_3\, +_2\, +_1 \rangle$$

At this point in the translation to KL, the definition of $sum$ given in Figure 5.3 assumes the form shown in Figure 5.6.
For a \(< b_n \ldots b_1 >\) which form a binary tuple or an identifier tuple, \(<o_n \ldots o_1>\) form an operator or defined function symbol tuples, and \(e\) denoting an expression, an indexing function \(\uparrow\) is defined such that for any \(1 \leq c \leq n:\)

\[
\begin{align*}
\uparrow [< b_n \ldots b_1 >] c &= < b_n, \ldots, b_1 > \\
\uparrow [(o_n \ldots o_1)] c &= o_c \\
\uparrow [e_1 e_2] c &= (\uparrow [e_1] c) (\uparrow [e_2] c) \\
\uparrow [e_1, e_2] c &= (\uparrow [e_1] c), (\uparrow [e_2] c) \\
\uparrow [e_1, otherwise] c &= (\uparrow [e_1] c), otherwise
\end{align*}
\]

TW translation scheme uses the projection function. The effect of TW translation defined below on sum is shown in Figure 5.7.

\[
TW [(f_c, \ldots, f_1) < t_1 \ldots t_n > = E] c =
\begin{cases}
(i, \uparrow (f_c, \ldots, f_1)) < t_1 \ldots t_n > i \\
(\uparrow [E] i))
\end{cases}
\]

The next step in the translation process is to identify simple expressions in the given definition and apply STS transformation.

**Definition 5.1** A simple expression is defined as:

- if an expression contains only an identifier tuple, or
- if an expression contains only a boolean tuple.

For example, KL definitions of the form:

\[
\begin{align*}
f_i < x_4 x_3 x_2 x_1 > &= < 1 \, 0 \, 0 \, 1 > \\
f_i < x_4 x_3 x_2 x_1 > &= < x_4 x_3 x_2 x_1 > \\
f_i < x_4 x_3 x_2 x_1 > &= < 1 \, 0 \, 0 \, 1 >, =i < x_4 x_3 x_2 x_1 > < 0 \, 0 \, 0 \, 0 >
\end{align*}
\]
are all simple expressions. Following the previous convention of \(< b_n \ldots b_1 >\) which forms a binary tuple or an identifier tuple, a new indexing function \(\downarrow\) is defined as:

\[
\downarrow [< b_n \ldots b_1 >] c = b_c
\]  

STS transformation uses the newly defined \(\downarrow\) function as given below:

\[
STS [(i, fi < t_1 \ldots t_n >= E)] = 
\]

\[
f_i < t_1 \ldots t_n > = (\downarrow [E] i), \quad \text{if } E \text{ is a simple expression}
\]

\[
f_i < t_1 \ldots t_n > = E, \quad \text{otherwise}
\]  

Application of STS transformation on sum results in the form shown in Figure 5.5. This forms input to the frontiers algorithm for eliminating recursion.

### 5.4 Concluding Remarks

FHDL is a high-level specification language for behavioral description of algorithms, based on applicative notation. The language allows the specification of both combinational and sequential circuits. Extensions to the language to include sequential circuits is discussed in the chapter on sequential circuit synthesis. KL, the kernel language, provides another level of abstraction at which the design can be specified. KL specifications can be viewed as specifications in the Boolean domain (domain 2) with and, or, and not as domain operations, to describe a circuit. Any FHDL specification can be translated to KL, and there exists a translation scheme to do so. KL specifications are not amenable to hardware realization because the specifications in KL can be recursive. The following chapter presents the algorithms for recursion removal from KL specifications and synthesis of combinational circuits.
Chapter 6

Foundations for Synthesis

The goal of the research reported here is to derive digital circuits from high-level behavioral specifications. Towards this end, a high-level functional language FHDL was defined in the previous chapter. The kernel language KL was introduced as an intermediate step towards the final synthesis. The translation scheme, presented in Chapter 5, transforms an FHDL definition to its equivalent KL definition with only and, or and not operations. Such operations could normally be mapped to hardware, but for recursion in KL. Hardware synthesis can be accomplished only after recursion elimination.

Fixpoints are used as the means for eliminating recursion. An iterative approach is used for computing the fixpoints. This approach necessitates a test for the equality of two functions. To compare two functions for equality it is necessary to check that functions coincide at every point in their domain. A naive approach to performing such comparisons evaluates both functions at every point in their domain; this is prohibitively expensive. Frontiers of functions are used to test the convergence of the sequence of approximations. However, the frontiers approach is feasible for monotonic functions only, and KL is non-monotonic due to the not operation. This chapter introduces the concept of domain inflation as a solution to overcome the problem of non-monotonicity. In an inflated domain, all KL functions are monotonic.

Overall, this chapter serves the following purposes:

- To provide the basis for translation to KL.
- Explain how the non-monotonicity of functions can be dealt with.
Present an algorithm for computing the frontiers in an inflated domain.

6.1 Boolean Function Domain

This section begins with a review of some basic mathematics necessary for the synthesis approach presented in this thesis.

6.1.1 Background

Given any set $S$, the elements of the set in the domain can be linked by an order relation. An ordering $\rho$ on $S$ is said to be total if for any $x, y \in S$ either $x \rho y$ or $y \rho x$ (or both). The set $S$ is referred to as a partially ordered set if $\exists a, b \in S$ such that $a \not\rho b$ and $b \not\rho a$. $a$ and $b$ are said to be incomparable elements. A partially ordered set can be represented as a union of totally ordered subsets.

Given $n$ sets $S_1, \ldots, S_n$, the set of all tuples $< x_1, \ldots, x_n >$ such that $x_i \in S_i$ is called the Cartesian product of $S_1$ to $S_n$ denoted by $S_1 \times \ldots \times S_n$. The notation used for tuples indicates that the order in which the elements are written is important.

A function $f : S_1 \to \ldots \to S_n \to R$ defines a mapping from $S = S_1 \times \ldots \times S_n$ to $R$. The sets $S_1, \ldots, S_n$ form the domain of $f$. The set $S$, also called the product domain, defines the set of all possible arguments to the function $f$.

A function $f$ is said to be monotonic if for an argument domain $S$:

$$x \leq y \implies f(x) \leq f(y) \quad \forall (x, y) \in S$$

If $x$ and $y$ are themselves tuples $< x_1, \ldots, x_n >$ and $< y_1, \ldots, y_n >$, respectively. Then $x \leq y$ is defined as:

$$< x_1, x_2, \ldots, x_n > \leq < y_1, y_2, \ldots, y_n > \quad \text{iff} \quad \forall i : 1 \leq i \leq n \quad x_i \leq y_i$$

For a finite domain, the ordering "$\leq$" of the product domain forms a lattice [103]. Each node in the lattice corresponds to a possible argument of the function. For example, consider a domain $T = \{0, 1\}$ where $0 \leq 1$. Figure 6.1 shows the lattice of values for $T^3$. The notation $T^n$ means a function of $n$ arguments over a domain $T$. 

79
6.1.2 Boolean Algebra

Boolean algebra is a well established mathematical subject. For the sake of completeness, boolean algebra is defined:

**Definition 6.1** A boolean algebra is a set $B$ containing:

- A value domain consisting of two distinct elements, written 0 (false) and 1 (true).
- Three operations: $\land$ (and), $\lor$ (or), and $\lnot$ (not). $\land$ and $\lor$ are binary operators and $\lnot$ is a unary operator.

**Definition 6.2** A logic function $f$ is a function of boolean variables, where the value domain for each variable is $T \in \{0, 1\}$; and the function $f$ is written $f : T^n \to T$, where $n$ is the number of logic variables.

Though it is clear from Definition 6.1, the universality of the domain operations can be formally stated by the following theorem:

**Theorem 6.1** Any mapping $T^n \to T$ can be written as an expression containing only $\land$, $\lor$, and $\lnot$ operators.
The mapping \( f : T^n \rightarrow T \) expressed in terms of variables \( v_1, \ldots, v_n \) can be fully defined by a truth table having \( 2^n \) rows. Generally, a logic function can be thought of as the set of \( m \) rows of the table having the result 1; let these be \( B_1, \ldots, B_m \), each \( B_i \) corresponds to an ordered tuple \( < B_{i1}, \ldots, B_{in} > \), where each \( B_{ij} \) is either \( v_j \) or \( \overline{v_j} \) depending on whether \( v_j \) is 1 or 0, respectively, for that row of the table. The function \( f \) can be written concisely as:

\[
f = \bigvee_{i=1}^{m} \left( \bigwedge_{j=1}^{n} B_{ij} \right)
\]

The form shown above is called the disjunctive normal form ("or"s of "and"s) or sum-of-products. A sum-of-products expression for a function is not unique due to redundant terms. It is the task of two-level logic minimization to find an irredundant cover of the sum-of-products expression. A function \( f \) is said to cover another function \( g \), if \( f \) assumes the value 1 whenever \( g \) does.

Following the discussion on product domains as lattices, the set of inputs to a boolean function can be represented as a lattice, each node in the lattice corresponding to a row of input values of a truth table. Such a lattice represents a function if each node is labeled with the value of the function corresponding to that particular combination of the input arguments. Thus any boolean function can be represented as a finite lattice.

This observation provides the clue for translation from FHDL to KL. Translation to KL has the following desired effects:

- provides the ability to represent functions as a finite lattice which is crucial for the use of frontiers algorithms, and

- offers a formal method for rewriting a function represented as a finite lattice to other equivalent forms, for example as a sum-of-products representation.

This is quite useful in an iterative approach to computing the fixpoints. The following subsection discusses this issue. Unfortunately the frontiers algorithm discussed
in the previous chapters cannot be applied directly because a boolean function do-
main is non-monotonic. Section 6.2 presents a solution to this problem.

6.1.3 Boolean Functions

Consider a function \( f : T^n \rightarrow T \) in KL, where

\[
\begin{align*}
  f_{a_n \ldots a_1} &= \ldots f \\
  f_{a_1} &= \ldots f_0 \\
  f_{a_1} &= \ldots f_1 \\
  \vdots
\end{align*}
\]

\( f \) is a recursively defined function. Domain theory provides a way of computing the solution to such functions as a limit of a sequence of approximations to the function \( f, f_0, f_1, \ldots \), starting with the function that maps every argument to the bottom (\( \bot \)) value of the domain. In each approximation, the previous approximation of the function is used for any recursive calls. The least fixpoint has been found when two successive approximations are found to be equal. A series of approximations is generated by substitutions in the original definition of \( f \):

\[
\begin{align*}
  f_0 a_n \ldots a_1 &= 0 \\
  f_1 a_n \ldots a_1 &= \ldots f_0 \\
  f_2 a_n \ldots a_1 &= \ldots f_1 \\
  \vdots
\end{align*}
\]

until the series converges when:

\[
\begin{align*}
  f_{i+1} a_n \ldots a_1 &= f_i a_n \ldots a_1
\end{align*}
\]

\( f_i \) is the least fixpoint – the function which is to be synthesized. In this manner, the recursive specifications can be replaced with their non-recursive fixpoints.

Each approximation in the ascending Kleene chain is non-recursive. The iterative approach to computing fixpoints requires substituting the non-recursive approximation of the original definition for each recursive call of the function to derive the next approximation. This process of substitution continues until two successive approximations are found to be equal.

Using frontiers approach to computing fixpoints, each approximation of a function results in a frontier set. A frontier set defines the boundary between 0-nodes
and 1-nodes. Efficient formulations are required for translating these frontiers sets for substitutions in the original function definition. The boolean function domain readily offers a methodology where a frontier set can be expressed as a canonical sum-of-product (product-of-sums) for a 1-frontier (0-frontier). Thus, any boolean expression in the AKC can be expressed as a KL function because the domain operations and value domain of boolean functions is contained in KL function domain. In other words, KL is an embellishment of boolean algebra, with additional features to ease the translation process. Thus any result of boolean algebra can be extended to KL functions in an AKC.

The boolean function domain, in addition to being a well established mathematical subject, is directly related to the function implementation. All the domain operations can be mapped directly to corresponding hardware structures.

6.2 Domain Inflation

Recall the procedure used to compute frontiers, which requires the evaluation of a function with a particular combination of arguments, and labelling the node in the lattice corresponding to the arguments with the result. In the process, the number of evaluations of the function over an entire argument domain is minimized by expanding only a 1-node (0-node) to compute the 1-frontier (0-frontier) starting from the top (bottom) node of the argument lattice. A frontier set cuts the lattice in two, with all the 1-nodes above the 0-frontier and vice versa.

It is tempting to apply this procedure directly to the boolean domain, where each product term of the sum-of-product expression represents a 1-node in the lattice. This is wrong: frontiers are applicable only to monotonic functions. To illustrate the point, consider the function $\text{xnor}$ defined as:

$$\text{xnor} = ab + \overline{a}\overline{b}$$

Figure 6.2 shows the corresponding lattice of values. It can be seen that no frontier exists. This is because the function $\text{xnor}$, and boolean functions in general, are non-
monotonic due to the not operation. The literal $\overline{v}_j$ in a product term $B_i$ requires a not operation.

A solution to overcome the non-monotonicity of functions due to the not operation is introduced in the following proposition:

**Proposition 6.1** Every boolean function, $f$, has a related monotonic inflated function $f'$ in a lattice of $2^{2n}$ nodes, each node corresponding to an ordered tuple $< B_1, \ldots, B_{2n} >$, where $B_j$ is $v_j$ for $j = 1, 3, \ldots$ and $\overline{v}_j$ for $j = 2, 4, \ldots$.

The above proposition suggests that by representing the literal $\overline{v}_j$ as a primary input in the argument domain, any boolean function can be represented as a monotonic function. This process is called domain inflation. Following the previous conventions discussed in Section 6.1.2, a sum-of-product expression of the boolean function in an inflated domain can be expressed as:

$$f = \bigvee_{1=1}^{m} \left( \bigwedge_{j=1}^{2n} B_{ij} \right)$$

For example, function $f$ with two arguments $a$ and $b$ in an inflated domain applies to the tuple $< a, \overline{a}, b, \overline{b} >$. If the truth table for $f$ at $< 0, 1, 0, 1 >$ is 1, the sum-of-product expression is written as $\overline{a}b$. Figure 6.3 shows the inflated lattice corresponding to the function xnor. The 1-frontier set for this function are the nodes $<0, 1, 0, 1>$ and $<1, 0, 1, 0>$. The sum of product for this expression corresponds exactly to its original definition.
Figure 6.3: Inflated lattice of xnor.

Note, that no restriction is placed on the input arguments in spite of the existence of logical connectivites within the argument tuple of an inflated domain. The result is the evaluation of a function with meaningless inputs, for example $<1, 1, 0, 1>$. Such inputs produce an output result in an inflated domain, but as a sum-of-products expression it is invalid. The following subsection elucidates this point.

6.2.1 Frontiers to the Rescue

To summarize the progress so far, the least fixpoint of a recursive function is an equivalent non-recursive function. Iterative computation of non-recursive equivalents requires testing for equality of functions. Frontiers were proposed as a concise canonical representation of a monotonic function. Boolean functions are non-monotonic on the ordered domain $T (0 < 1)$ due to the not operation. Non-monotonic boolean functions can be represented as a monotonic function by inflating the argument domain. These functions on inflated domains are then amenable to the frontiers algorithms.

**Proposition 6.2** A 1-frontier set of a boolean expression (in an inflated domain) is its minimal sum-of-products expression.

**Proof:** Follows from Proposition 6.3.
The minimality of a sum-of-product expression comes from two sources:

- Minimum number of product terms

- Minimum number of literals in each product term, *per se* an expression which is irredundant and prime.

An *implicant* of a function $f$ is a product term $h$ when $h$ is covered by $f$. A *prime implicant* $p$ of a function $f$ is a product term which is covered by $f$, and deletion of any literal from $p$ results in a new product which is not covered by $f$. Thus prime $p$ is a product term that is not covered by any other implicant of the function.

The similarity of the truth table approach and the frontiers approach is apparent. Instead of using a tabular form, the input values are represented as nodes in the lattice, where each node in the lattice corresponds to a tuple of input values. Though the two approaches might be considered entirely equivalent, there are other advantages to be gained.

**Proposition 6.3** Each node in the 1-frontier set is a prime implicant.

**Proof:** Let a product term $t$ be derived from a 1-frontier set corresponding to a node $< \ldots, 1, \ldots >$ in the lattice. The deletion of a literal from a product term (recall how a sum-of-product expression is written in an inflated domain) of a frontier set results in crossing the frontier, i.e. the new product term includes terms that are not implicants of the function. This is because the node corresponding to a product term of the deleted literal $< \ldots, 0, \ldots >$ lies below $< \ldots, 1,\ldots >$, due to the ordering on the lattice$^1$. □

The proof for Proposition 6.2 follows from the above. The 1-frontier is the union of primes and hence the sum-of-product expression is irredundant. These propositions provide a solution to the problem of "meaningless" terms that can possibly occur in a sum-of-products expression, as mentioned earlier. The solution is based on the argument that such a case can never occur on a 1-frontier, i.e. the term is not a prime.

---

$^1$For an intuitive understanding, see Figure 6.4.
Consider a 1-node in the lattice \(<\ldots, 1, 1, \ldots>\) corresponding to the literals \(x\) and \(\overline{x}\) of a function \(f\). Let the notation \(f_x\) denote the logic function obtained from \(f\) by replacing \(x\) by 1 and \(\overline{x}\) by 0, and \(f_{\overline{x}}\) from \(f\) by replacing \(\overline{x}\) by 1 and \(x\) by 0. A function \(f\) is independent of \(x\) if and only if \(f \equiv f_x\). If \(f\) is independent of \(x\), then there always exists a 1-node \(<\ldots, 0, 1, \ldots>\) below the node \(<\ldots, 1, 1, \ldots>\). A similar argument holds for \(f_{\overline{x}}\). A function \(f\) (defined in a normal domain) cannot be dependent on both \(x\) and \(\overline{x}\) (because \(x\) and \(\overline{x}\) are complements of each other). Such a product term is said to “overly define” the function. Such nodes can never occur in a 1-frontier set.

### 6.2.2 An Example

Consider a function \(f\) defined as \(f = ab + \overline{ab}\). Figure 6.4 shows the lattice of values (in square braces) of the function where each tuple represents the value assumed corresponding to \(<a, \overline{a}, b, \overline{b}>\) of the inflated domain. The dotted line corresponds to the frontier, the boundary between 1-nodes and 0-nodes. Though the nodes \(<0, 0, 1, 1>, <0, 1, 1, 0>, <1, 0, 1, 0>\) lie on the frontier, they are redundant with respect to \(<0, 0, 1, 0>\). The corresponding 1-frontier is the set \{\(<0, 0, 1, 0>\}\). The sum-of-product expression can be obtained from this 1-frontier set as \(b\). The function is independent of \(a\). This don’t care condition with respect to \(a\) results in
a zero value for $a$ and $\bar{a}$ in the 1-frontier set.

6.3 Computing Frontiers

A frontier defines the boundary between a set of 0-nodes and 1-nodes in a finite lattice. If the function is monotonic and $0 \leq 1$, 0-nodes cannot lie above the 1-nodes of the function. The restriction to monotonic functions can be overcome by domain inflation, as discussed in this chapter. Hence a frontier can always be found. The frontiers algorithm discussed in the previous chapters is based on searching the lattice top down (or bottom up or both) to locate the frontier. Such search-based algorithms are expensive, as the domain is exponential in the number of arguments to the function. This section presents a new algorithm for computing the frontiers. The approach is to compute the frontiers directly from a KL definition, avoiding the search-process. McCrosky and Wang [75] also present algorithms for deriving frontier sets from the abstract syntax. The algorithms were proposed for monotonic functions only. This thesis extends this work to include non-monotonic functions, i.e functions that have not operation in their expressions. The algorithms are optimized to improve the performance metrics.

6.3.1 Algorithm for Computing Frontiers

An expression in AKC, of type $T^n \rightarrow T$, can be represented in KL as:

\[
<\text{expr}> ::= 0 \mid 1 \mid <\text{id}> \mid <\text{expr}> + <\text{expr}>
\mid <\text{expr}> \cdot <\text{expr}> \mid <\text{neg_expr}>
\]

\[
<\text{neg_expr}> ::= \neg <\text{expr}>
\]

where $+$, $\cdot$, and $\neg$ represent \textit{or}, \textit{and} and \textit{not} operations respectively, $<\text{expr}>$ is an expression and $<\text{id}>$ corresponds to an identifier.

To represent these expressions as a finite lattice, in general, the expressions are evaluated by application to tuples consisting of $2 \times n$ values, due to domain inflation,
in \( T, (t_1, t_T, \ldots, t_n, t_R), \) where \( n \) is the exponent of the lattice in an uninflated domain. A 1-frontier set is built of such tuples, the irredundant set of 1-nodes. The approach presented here constructs this 1-frontier set by examining expressions and replacing the expressions with a 1-frontier set corresponding to the expression. Thus, for each syntactic structure of the expression there is a corresponding re-write rule. Duals of these re-write rules exist for deriving corresponding 0-frontiers.

The 1-frontier for 0 and 1 are obvious:

\[
\mathcal{O}(0) = \{\} \quad (6.3)
\]

\[
\mathcal{O}(1) = \{<0,0,\ldots,0>\} \quad (6.4)
\]

The 1-frontier of an identifier is the element that is 1 for that identifier and 0 everywhere else:

\[
\mathcal{O}(<id>_j) = <0_1, \ldots, 1_{2j-1}, 0_{2j}, \ldots, 0_{2n}>
\]

(6.5)

where the subscript indicates the position of the variable in a tuple.

To better understand the derivation of the 1-frontier for a sum operation, consider the 1-node sets. The 1-node set of a sum is the union of 1-sets of the subsidiary expressions:

\[
\mathcal{O}(id_a + id_b) = \{x \mid x_{id_a} = 1 \vee x_{id_b} = 1\} \quad (6.6)
\]

where the notation \( x_{id_a} = 1 \) is used to designate a tuple where \( id_a \) is 1 and all other identifiers are either 0 or 1. From the Equation 6.6:

\[
\{x \mid x_{id_a} = 1 \vee x_{id_b} = 1\} = \{x \mid x_{id_a} = 1\} \cup \{x \mid x_{id_b} = 1\} = \mathcal{O}(id_a) \cup \mathcal{O}(id_b)
\]

(6.7)

The 1-frontier set can be computed from 1-nodes by eliminating the redundant nodes.
such that:

\[ \mathcal{O}(id_a) = \min(\overline{O}(id_a)) \]  \hspace{1cm} (6.8)

Let the 1-frontier for sums be defined in terms of a new operation, \( \cup \):

\[ \mathcal{O}(id_a + id_b) = \mathcal{O}(id_a) \cup \mathcal{O}(id_b) \]  \hspace{1cm} (6.9)

This operation is defined now. Applying Equation 6.8 to the above equation:

\[ \min(\overline{O}(id_a + id_b)) = \min(\overline{O}(id_a)) \cup \min(\overline{O}(id_b)) \]  \hspace{1cm} (6.10)

Using Equation 6.7, the above equation can be written as:

\[ \min(\overline{O}(id_a) \cup \overline{O}(id_b)) = \min(\overline{O}(id_a)) \cup \min(\overline{O}(id_b)) \]  \hspace{1cm} (6.11)

Let \( \mathcal{X} \) and \( \mathcal{Y} \) denote \( \overline{O}(id_a) \) and \( \overline{O}(id_b) \) respectively. Renaming the terms in the above equation yields:

\[ \min(\mathcal{X} \cup \mathcal{Y}) = (\min \mathcal{X}) \cup (\min \mathcal{Y}) \]  \hspace{1cm} (6.12)

Recall that \( \min \) is a function that eliminates redundant elements in a set. \( \min \) has an inverse \( \min^{-1} \) that reconstructs the set such that \( \min^{-1}(\min \mathcal{X}) = \mathcal{X} \). Applying this inverse operator in the above equation:

\[ \mathcal{X} \cup \mathcal{Y} = \min ((\min^{-1} \mathcal{X}) \cup (\min^{-1} \mathcal{Y})) \]  \hspace{1cm} (6.13)

\( \min^{-1} \mathcal{X} \) contains all elements greater than or equal to any element in \( \mathcal{X} \). \( (\min^{-1} \mathcal{X}) \cup (\min^{-1} \mathcal{Y}) \) contains all elements greater than or equal to any element in \( \mathcal{X} \) or \( \mathcal{Y} \) and contains no elements not in either \( \mathcal{X} \) or \( \mathcal{Y} \). The elements of \( \min ((\min^{-1} \mathcal{X}) \cup \mathcal{Y}) \) must be...
\[(\min^{-1} \mathcal{Y})\] must be in \(\mathcal{X}\) or \(\mathcal{Y}\). Hence,

\[
\min ((\min^{-1} \mathcal{X}) \cup (\min^{-1} \mathcal{Y})) \equiv \min(\mathcal{X} \cup \mathcal{Y}) \quad (6.14)
\]

Using this relation, Equation 6.13 reduces to:

\[
\mathcal{X} \cup \mathcal{Y} = \min (\mathcal{X} \cup \mathcal{Y}) \quad (6.15)
\]

Using Equation 6.15, Equation 6.9 can be written as:

\[
\mathcal{O}(id_a \cdot id_b) = \min(\mathcal{O}(id_a) \cup \mathcal{O}(id_b)) \quad (6.16)
\]

Following the similar conventions, the 1-frontier for a product expression from 1-node set can be derived as:

\[
\overline{\mathcal{O}}(id_a \cdot id_b) = \{x \mid x_{id_a} = 1 \land x_{id_b} = 1\}
\]

\[
= \{x \mid x_{id_a} = 1\} \cap \{x \mid x_{id_b} = 1\}
\]

\[
= \overline{\mathcal{O}}(id_a) \cap \overline{\mathcal{O}}(id_b) \quad (6.17)
\]

Let the 1-frontier for products be defined as: \(\overline{\mathcal{O}}\):

\[
\mathcal{O}(id_a \cdot id_b) = \mathcal{O}(id_a) \cap \mathcal{O}(id_b) \quad (6.18)
\]

The meaning of \(\overline{\mathcal{O}}\) can be derived using the similar approach used for \(\cup\). Following the previous derivation for \(\cup\) and using Equation 6.18, the definition of \(\overline{\mathcal{O}}\) can be written as:

\[
\mathcal{X} \overline{\cap} \mathcal{Y} = \min ((\min^{-1} \mathcal{X}) \cap (\min^{-1} \mathcal{Y})) \quad (6.19)
\]

The elements of \((\min^{-1} \mathcal{X}) \cap (\min^{-1} \mathcal{Y})\) contain only those elements that have 1's
where some members of \( \mathcal{X} \) or some members of \( \mathcal{Y} \) have ones and 0’s elsewhere.

\[
\mathcal{X} \cap \mathcal{Y} = \min\{\forall p \in \mathcal{X}, \forall q \in \mathcal{Y}, < p_0 + q_0, \ldots, p_{n-1} + q_{n-1}>\} \quad (6.20)
\]

Using Equation 6.20, the expression for computing the frontiers for a product expression given by Equation 6.18 reduces to:

\[
\mathcal{O}(id_a \cdot id_b) = \min\{\forall p \in \mathcal{O}(id_a), \forall q \in \mathcal{O}(id_b), < p_0 + q_0, \ldots, p_{n-1} + q_{n-1}>\} \quad (6.21)
\]

Extending the approach to the negated expressions, the 1-frontier sets for a negated 0 and 1 can be written as:

\[
\mathcal{O}(\text{not}(0)) = \mathcal{O}(1) \quad \text{(by the application of not operation)}
\]
\[
= \{<0, 0, \ldots, 0>\} \quad (6.22)
\]
\[
\mathcal{O}(\text{not}(1)) = \mathcal{O}(0) \quad \text{(by the application of not operation)}
\]
\[
= \{\} \quad (6.23)
\]

The 1-frontier of a negated identifier is the element that is 1 for that location of the identifier corresponding the negated element, contained in the tuple due to domain inflation, and 0 everywhere else:

\[
\mathcal{O}(\text{not}(<id>_i)) = <0_1, \ldots, 0_{2j-1}, 1_{2j}, \ldots, 0_{2n}> \quad (6.24)
\]

A negated product expression has the 1-frontier set:

\[
\mathcal{O}(\text{not}(a \cdot b)) = \mathcal{O}((\text{not}(a)) + (\text{not}(b))) \quad \text{(by laws of boolean algebra)}
\]
\[
= \mathcal{O}(\text{not}(a)) \cup \mathcal{O}(\text{not}(b)) \quad \text{(by Eq.6.9)} \quad (6.25)
\]
Similarly, the 1-frontier set for a negated sum expression can be written as:

\[ O(\text{not}(a + b)) = O((\text{not}(a) \bullet (\text{not}(b))) \text{ (by laws of boolean algebra)} \]
\[ = O(\text{not}(a)) \cap O(\text{not}(b)) \text{ (by Eq.6.18)} \quad (6.26) \]

The above equations allows to compute 1-frontiers for any expression in KL. The algorithm performs two passes over the abstract syntax tree; the top-down pass "pushing" the not operation deeper into the expression tree applying DeMorgan's laws, until each successor node of it is an identifier. The bottom-up pass applies the above rules to compute the 1-frontier sets. Thus, this algorithm neither performs any evaluations of the expression nor searches the domain for the frontier set. Although this algorithm is potentially exponential in the worst-case, its practical performance is excellent, as will be demonstrated in the next chapter where the performance of this algorithm is compared with the previous frontiers algorithm on a practical problem in the context of circuit synthesis.

### 6.4 Concluding Remarks

The fixpoint of a KL specification, using the frontier representation of a function, corresponds to its irredundant sum-of-product expression, i.e all FHDL specifications have a corresponding sum-of-product expression, which are identified to derive hardware structures. The advantages of this approach are:

- The structure of recursion is immaterial, i.e. there exists a sum-of-product representation for all recursive specifications.

- The approach has strong theoretical foundations.

- The frontiers algorithm discussed in this chapter makes it practically feasible to apply the above theoretically sound concepts for circuit synthesis.
Chapter 7

Circuit Synthesis

High-level synthesis is the process of transforming behavioral descriptions to their equivalent structural descriptions that can be mapped directly to hardware. In the previous chapters, FHDL, a high-level behavioral specification language based on functional notation was presented. The functional languages basis of FHDL permit the specifications to be recursive. Recursion should be eliminated prior to hardware synthesis. Earlier chapters have demonstrated the approach adopted here for recursion removal. This chapter describes the process of generating descriptions suitable for a back-end system to generate hardware from the frontier sets.

This chapter begins with a summary of steps involved in the synthesis process. Section 7.2 presents a discussion on the advantage of the approach presented in this thesis over conventional high-level synthesis systems. Section 7.3 describes the interface to some of the existing tools. The current interface supports a VHDL and PLA code generator. This chapter concludes with a discussion on the performance of the synthesis tool. The performance is measured in time and compares the two frontiers algorithms discussed in this thesis. The most commonly cited benchmark example in the literature, the greatest common divisor problem, was used as a standard for tabulating the results.
7.1 Overview of Synthesis Process

This section provides a summary of the synthesis process using FHDL. The synthesis process consists of the following steps:

1. Specify the algorithm in terms of the behavior using FHDL.

2. The specifications are debugged using test data until the specifications match the intent.

3. Generate structural descriptions from the FHDL specifications. This is an automated process and essentially constitutes the following steps:
   
   - Transform the FHDL descriptions to KL. The required translation rules and the need for translation was presented in Chapter 5.
   - Using the algorithms described in this thesis for recursion removal, apply frontiers algorithms on KL to generate the frontier sets.
7.2 Advantages of A Semantics Based Approach

Frontiers provide a sound approach to computing fixpoints of a recursion equation. Previous chapters have illustrated the use of frontiers for hardware synthesis from behavioral specifications. This section illustrates the use of the denotational approach for hardware synthesis by rendering a new dimension to the sense of optimality of a circuit realization from a high-level specification.

To illustrate the point, consider the two definitions, \( \text{mult} \) and \( \text{loop} \), shown in Figure 7.2. Conventional synthesis from these specifications is accomplished without realization of the fact that both the functions are in fact equivalent, resulting in two entirely different architectures. A possible realization of one would require the use of a multiplier and the other a feedback loop. Thus, in currently available synthesis
systems the specification style plays a crucial role in determining the synthesized hardware.

But this is not the case for the approach presented here. The synthesis of hardware from the specifications is accomplished by first computing their frontiers. Hardware realization is due to these frontiers. Frontier sets of two functions are equal if they denote the same function. Computing the frontiers for both these functions results in the same frontier set, hence the same hardware realization for both these functions.

### 7.3 Interface to the Layout Tools

Hardware synthesis is based on deriving frontier sets. The preceding chapters have shown how combinational circuit descriptions in the form of its frontier set is generated from the corresponding FHDL specification. The correspondence between the frontier sets and the sum-of-products expression was unequivocally established. This section describes the method of converting the frontier sets to descriptions acceptable to layout tools. The tools of interest are the PLA generator and the VHDL compiler in the Electric [98] VLSI layout system.

#### 7.3.1 PLA Generation

A Programmable Logic Array (PLA) provides a regular structure for implementing combinational logic functions. A PLA typically uses an AND-OR structure which is ideally suited for a sum-of-product realization. The code generation scheme implemented supports both NMOS and CMOS PLA generator tools [56] supported by the Electric design system.

A PLA with $n$ inputs and $m$ outputs can realize $m$ functions of $n$ variables. Figure 7.3 shows a typical structure of PLA. The input to the AND plane are the input signal lines representing the logic variables. The AND plane combines these to produce a product term. The OR plane generates the output of PLA in the form
Figure 7.3: Programmable logic array structure.

of signal lines, one for each of the output logic variable in the logic design, which is the sum of appropriate product terms from the AND plane.

Typically, the contents of a PLA are specified as a truth table. The input side of the table specifies the product terms. The symbols 0, 1, and x indicate whether a variable is complemented, uncomplimented or not present in the corresponding product term. The output side of the table specifies which product terms appear in each output function. A 1 or 0 indicates whether a given product term is present or not present in the corresponding output function.

From the above discussion on the methodology for specifying a PLA structure, the AND plane can be derived as the union of all frontier sets of each individual split function. This is possible because each node in a frontier set corresponds to a product term. The number of output lines for the PLA generated from the frontier set is identified from the number of lines to which the function is split. The OR plane is generated using a trivial algorithm. The algorithm maintains an array of $m \times n$ elements, where $m$ is the cardinality of the union of all frontier sets such that each row in the array corresponds to a product term and $n$ is the number of bits to which the function is split. Each column corresponds to an output function. The algorithm traverses through frontier set for each bit-wise function marking along
set name = GCD
set inputs = 4
set outputs = 2
set pterms = 9
begin
  1 0 1 0 1 0
  0 1 1 0 0 1
  1 1 1 0 0 1
  1 0 0 1 0 1
  0 1 0 1 0 1
  1 1 0 1 0 1
  1 0 1 1 0 1
  0 1 1 1 0 1
  1 1 1 1 1 1
end

Figure 7.4: NMOS PLA code for 2-bit greatest common divisor problem.

the column corresponding to the function under consideration with a 1 for each matched product term. Code generation for the NMOS and CMOS PLA generator tool is accomplished using these AND and OR plane definitions and adding the necessary syntactic structures required by PLA generator tool. Figure 7.4 illustrates the NMOS PLA code generated for an example 2-bit greatest common divisor PLA circuit\(^1\). Figure 7.5 depicts the corresponding CMOS PLA code. Figures 7.6 and 7.7 show the NMOS and CMOS PLAs generated from the corresponding code generated.

\(^1\)The examples presented here are restricted on purpose; the intent is to illustrate the concepts clearly and in no way imply or indicate any limitation on the system.
Figure 7.5: CMOS PLA code for 2-bit greatest common divisor problem.

Figure 7.6: NMOS PLA for the 2-bit greatest common divisor.
Figure 7.7: CMOS PLA for the 2-bit greatest common divisor.
7.4 VHDL Code Generation

VHDL is an emerging standard in hardware description languages. It has been adopted as IEEE standard 1076. The language supports the design description both at structural and behavioral level. This section concentrates on generating the structural descriptions in VHDL (behavior is specified in FHDL). The primary interest in generating VHDL code is the wide range of platforms that support this language.

Structural specification in VHDL entails specifying the interconnection of components, usually at the gate level, such that the designated behavior is achieved. In order to map the obtained frontier sets to VHDL structural specification, it is necessary to specify:

1. The name of the top-level component (in terms of VHDL, the entity).
2. The interface to the entity identifying the inputs and outputs.
3. Architectural description consisting of:
   
   (a) gate level components used in the circuit,
   (b) all the signal lines that connect the components of the circuit, and
   (c) connectivity of the components.

The entity name can be drawn from the synthesize statement in FHDL. The input ports are exactly the input arguments to the bit-wise split function and the output ports correspond to the function after splitting transformation is applied. Figure 7.8 illustrates the entity declarative part of VHDL generated assuming that a 2-bit gcd circuit (for the sake of simplicity and ease of display of the corresponding figures) is being generated.

The gate level components in the architectural description of an entity are the domain elements of KL, i.e and, or and not. For the sake of simplicity, only two-input and and or gates are used. The process of identifying the signal lines and the circuit connectivity is illustrated with an example.
For the sake of brevity, consider only the nodes \(<0, 1, 1, 0>, <1, 1, 1, 1>\) of the \(gcd\) example. The sum-of-product expression can be written using only the gate-level components described above as:

\[
\text{or} \ (\text{and} \ (\text{and} \ (\neg a_2) \ a_1) \ (\text{and} \ b_2 \ (\neg b_1))) \ (\text{and} \ (a_2 \ a_1) \ (\text{and} \ b_2 \ b_1))
\]

A circuit implementing this expression is depicted in Figure 7.9. The process of VHDL code generation is two step, consisting of:

- Generating an abstract syntax expression consisting of \textit{and}, \textit{or} and \textit{not} expression for each frontier set, as shown above.

- Traversing this abstract syntax tree bottom-up, replacing each application node with the corresponding operation with its successor nodes as the input arguments until the root node is reached. The algorithm assigns a name to all these intermediate nodes and this collection is exactly the signal lines in the circuit. Generation of VHDL code is a simple matter of rewriting this information with the appropriate syntactic structure.

Figure 7.10 shows the corresponding abstract syntax tree. The nodes tagged with
Figure 7.9: Architecture of the circuit implementing SOP expression.

Figure 7.10: Abstract syntax tree generated for VHDL code generation.

"@" symbol are the intermediate nodes. Figure 7.11 shows the code generated. The corresponding VLSI layout is shown in Figure 7.12.
entity gcd is port
    (A_2, A_1, B_2, B_1 : in bit;
    GCD_1 : out bit);
end gcd;

architecture gcd_body of gcd is
    component and2 port (a, b : in bit; z : out bit);
    end component;
    component or2 port (a, b : in bit; z : out bit);
    end component;
    component inverter port ( a : in bit; z : out bit);
    end component;
signal G0002, G0004, G0006, G0008, G0010, G0012,
    G0014, G0016 : bit ;

begin
    G0001: inverter port map (A_2, G0002);
    G0003: inverter port map (B_2, G0004);
    G0005: and2 port map (G0002, A_1, G0006);
    G0007: and2 port map (B_1, G0006, G0008);
    G0009: and2 port map (G0004, G0008, G0010);
    G0011: and2 port map (A_2, A_1, G0012);
    G0013: and2 port map (B_2, B_1, G0014);
    G0015: and2 port map (G0012, G0014, G0016);
    G0017: or2 port map (G0010, G0016, GCD_1);
end;

Figure 7.11: VHDL code generated.
Figure 7.12: Layout generated from the VHDL code.
7.5 Runtime Performance of Algorithms

Frontiers algorithms form the basis for hardware synthesis from behavioral specifications. All frontiers algorithms are exponential in the worst case, the exponent being twice the number of bits in arguments to the function. The fundamental reason for optimism in the use of these algorithms is their average case behavior. It is possible to find frontiers in acceptable times and are easily computable for realistic size problems. Also, substantial gains in performance are accomplished by using standard compilation techniques such as dependency analysis, a topological sort of functions based on dependency relationship, and memoization, reducing the computational cost of computing the frontiers to a one-time cost.

This section presents run-time analysis for the gcd implementation. This problem has been chosen because of its ubiquitous reference in the behavioral synthesis systems literature and is a benchmark circuit \(^2\). However, there are published benchmarks neither for the performance analysis of the algorithms employed nor for the frontiers algorithms, which constitutes the main synthesis algorithm for the approach discussed in this thesis. Thus no direct comparison could be made. Table 7.1 refers to the run-times obtained for the two synthesis algorithms, frontiers by computation (and searching the domain) and from the abstract syntax. The times reported are in seconds, based on a Lisp implementation running on a Sun Sparc Station 2.

\(^2\)From High-Level Synthesis '93 workshop and are available by ftp from mcnc.mcnc.org.

<table>
<thead>
<tr>
<th>No. of bits to which the Function is Split</th>
<th>Domain Size</th>
<th>Search Based</th>
<th>From Abstract Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>0.965</td>
<td>0.933</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>363.067</td>
<td>4.2633</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>10058.050</td>
<td>12.843</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>Runs out of memory</td>
<td>53.756</td>
</tr>
<tr>
<td>32</td>
<td>128</td>
<td>–</td>
<td>396.909</td>
</tr>
</tbody>
</table>

Table 7.1: Run-time performance of the algorithms.
7.6 Concluding Remarks

The fixpoint of a KL specification, using the frontier representation of a function, corresponds to its irredundant SOP expression, i.e. all FHDL specifications have a corresponding SOP expression, which are identified to derive hardware structures. The advantages of this approach are:

- The structure of recursion is immaterial, i.e. there exists a sum-of-product representation for all recursive specifications.

- In spite of different nature of specifications in FHDL, the same two-level hardware is realized.

- The approach has strong theoretical foundations.
Chapter 8

Multi-level Logic Synthesis

Up to this point in this thesis, circuit optimization has meant minimizing the number of product terms in two-level sum-of-product expressions. Two-level circuits can be further minimized by abandoning the two-level restriction and sharing common sub-expressions. Such optimizations fall in the realm of multi-level logic synthesis and optimization. This chapter presents a new approach to deriving multi-level circuits.

Multi-level synthesis refers to synthesis techniques that derive circuits with an unlimited number of intermediate signals. Using two-level synthesis, the only intermediate signals are the result of product terms formed from the inputs. In multi-level synthesis, an intermediate signal may be an output of a multi-level function whose inputs may also be outputs of another multi-level function. Such implementations are sometimes referred to as random logic.

This chapter begins with a brief introduction to multi-level logic synthesis. The distinction between multi-level circuits in the context of this thesis and its general usage in the literature is then discussed. A new approach to multi-level circuit synthesis using frontiers follows. The synthesis process includes two-levels of optimization. The first step of optimization identifies the common sub-expressions. An algorithm for identifying common sub-expressions among multiple functions is presented. The common sub-expressions identified are subject to further optimization via factoring. Using lattices to represent functions provides interesting clues for finding factored forms. The properties of the lattice are exploited to derive factored forms.
8.1 Multi-level Logic Synthesis

Generally, multi-level synthesis systems [19] have two goals:

- minimize the total area of the fabricated chip, and
- control the path delays in the circuit.

Area minimization is achieved exclusively by reusing logic. The generally accepted approach is to identify and eliminate redundant elements of circuits and increase the fanout of the common sub-expressions. This is in contrast with two-level minimization methods which focus on minimizing the number of product terms, which in turn minimizes the PLA area.

The techniques adopted for multi-level synthesis seek behaviorally equivalent implementations of a given function that are optimal with respect to a cost function involving area and delay. The cost function is directly related to the number of literals in a given function; the objective is to minimize their number. A number of standard approaches [80] exist to achieve this goal, including function decomposition, identification of common sub-expressions among different logic functions, and factoring to derive a factored form from a sum-of-products form of a function. All these operations result in a factored form.

Factored forms are isomorphic to tree structures, where each internal node is an and, or or not operator and each leaf node is a literal. Factored forms are the natural representation for multi-level synthesis. Unfortunately, all the optimization techniques leading to factored forms are computationally expensive and are not considered practical for large circuits [33, 80]. Another unanswered question in the use of factored forms for multi-level synthesis is the notion of optimality. No test for optimality exists for factored forms, unlike two-level minimization where the Quine-McClusky tabulation method can provide an effective answer [19].

From the discussion so far, it is clear that factored forms are key to multi-level synthesis. In the approach to synthesis presented in the following section, multi-level circuits are synthesized by noting the progression of frontiers. The monotonic
nature of approximations results in a frontier for each approximation moving from the top element in the lattice towards the bottom element until the fixpoint is found. Each frontier set results in a sum-of-products expression. As the frontier progresses towards the bottom element from the top element in the lattice, the progression of the frontiers results in a sum-of-product expression with an enlarged cube or product terms, i.e. the number of minterms is increased (see Figure 8.1). These enlarged product terms include all the product terms of the previous approximations, resulting in a naturally nested structure. Using this approach, two levels of optimization are performed:

- **Global optimization**: sharing product terms across the function being synthesized for each approximation, i.e. identifying the common sub-expressions within the split functions for each frontier generated, and

- **Local optimization**: minimize the count of literals within each common sub-expression obtained from the above step, i.e. derive a factored form.

True circuit optimization is achieved only if the derived frontier set is optimal in the original domain. The frontier set derived using the algorithms presented in the earlier chapters is the set of prime implicants of the original function. The prime implicant set can be minimized by identifying the minimal cover of the function. Due to the existence of several excellent tools for two-level logic minimization, this thesis will not explore this topic. Instead, this thesis extends the use of the frontiers approach to identify common sub-expressions and factored forms and rely on existing logic minimization tools to arrive at truly optimal forms that form the input to the algorithms discussed in this chapter. The previous chapter has identified the interaction between logic minimization tools and the synthesis tool presented in this thesis. Use of logic minimization tools results in elimination of nonessential primes and hence a smaller frontier set, a subset of the original frontier set derived.

The synthesis of multi-level circuits presented here leverages on the naturally nested structure that results from the progression of the frontier set. For each frontier approximation, additional product terms that result from the progression of the
frontier with respect to the previous approximation are identified. Nested circuits are synthesized by deriving the combinational circuit only for the additional product terms identified for the current approximation. Combinational logic synthesized for the previous approximation along with the newly derived combinational block that implements the additional product terms identified for the current approximation with respect to the previous approximation are ored to complete the realization of the current approximation. Thus, the sum-of-products terms required for the current approximation are incrementally synthesized with a clearly nested structure. Though, at first glance it might appear that the term multi-level has been abused, the spirit in which it is used in the context of general multi-level synthesis systems is captured – minimizing the fabricated area by increasing the fan out of a combinational circuit block and the ability to explore design space to minimize time delays. Exploration of the design space is accomplished by the ability to synthesize different forms of realization (e.g. a PLA or a combinational block) for each combinational block.

8.2 Frontiers for Multi-level Synthesis

This section begins with the synthesis of nested forms. Factored forms are derived in the following subsection.

8.2.1 Synthesis of Nested Forms

Recall the procedure used to compute the fixpoints of recursion equations, where a recursively defined function $f$ was replaced by its equivalent non-recursive specification through a series of approximations to the original function $f$ as:

$$f_0 \ a_n \ \ldots \ a_1 = 0$$
$$f_1 \ a_n \ \ldots \ a_1 = \ldots \ f_0 \ \ldots$$
$$f_2 \ a_n \ \ldots \ a_1 = \ldots \ f_1 \ \ldots$$
$$\vdots$$

112
Figure 8.1: Monotonicity between approximations.

until the series converges when:

\[ f_i = f_{i+1} \]

\( f_i \) is the least fixpoint. By using frontier representation of the function it was shown that \( f_i \) represents an irredundant sum-of-product expression of the original function. Two-level synthesis is accomplished by mapping this sum-of-product expression to a PLA.

Figure 8.1 shows the progression of a 1-frontier from one approximation to the next. The figure also illustrates the monotonicity between the approximations. The sequence of approximations increase monotonically, never repeating until the fixpoint is reached. Each approximation \( f_i \) may be viewed as a successive approximation to the solution of \( f \), whose function space encompasses the previous approximation \( f_{i-1} \). \( f_i \) is defined for more values, and yields the same result for any value for which they are both defined. Figure 8.2 illustrates the idea of progressive recognition of 1's of a function until the fixpoint is reached, where each square block represents the function space (product terms) for the corresponding approximation.

The nested nature of circuit synthesis is due to this property of the AKC. Figure 8.3 depicts the strategy used for multi-level synthesis. Let \( F_1 \) represent the combinational block for the circuit due to the approximation \( f_1 \) of \( f \). From the monotonicity between approximations, the next approximation \( f_2 \) is:
Function space covered by the previous approximation
Function space for the current approximation

Figure 8.2: Function space in AKC.

\[ f_2 \equiv f_1 \cup f'_2 \]

where \( f'_2 \) is the shaded area of \( f_2 \) enclosing \( f_1 \) in Figure 8.2. \( f'_2 \) is the result of additional product terms in the frontier set of \( f_2 \) with respect to \( f_1 \):

\[ f'_2 = f_2 - f_1 \]

where "-" operation is read as set-difference. Extending this process to the complete chain of approximations, the function \( f \) can be computed as:

\[ f \equiv (\ldots (f_1 \cup f'_2) \ldots \cup f'_l) \]

where \( f_l \) is the fixpoint of \( f \). The above formulation maps to a hardware structure with each combinational block feeding the next block and the union operation mapped to a logical or operation resulting in a nested structure as shown in Figure 8.3.

Let a function \( f \) be split to \( n \) bits. \( f^1_l, \ldots, f^n_l \) correspond to the \( i^{th} \) approximation for each of the \( n \)-functions. For the first approximation, these would correspond to
Global logic reduction is achieved through the identification of the common sub-expressions within the first approximations, then as the computation of the frontier expressions progresses the subsequent common sub-expressions are identified in the expanded frontier set across all the functions corresponding to each bit. A common sub-expression encapsulates identical product terms within the given set of frontier approximations.

The idea is illustrated with a simple example. For the sake of simplicity, the functions are denoted in lower case and each cube or product term in an approximation is represented in upper case alphabets. Using this notation, the sum-of-products expression could be graphically represented as shown in Figure 8.4. Each edge originating at a node denoting a function represents a logical or operation. From the figure, it is apparent that the subset of the nodes \{A, B\} can be shared.
Table 8.1: Common sub-expressions for the example.

<table>
<thead>
<tr>
<th>Function</th>
<th>Influence set</th>
<th>Optimized influence set</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>A, B, C</td>
<td>{A, B}, {C}</td>
</tr>
<tr>
<td>g</td>
<td>A, B</td>
<td>{A, B}</td>
</tr>
<tr>
<td>h</td>
<td>C, D, E</td>
<td>{C}, {D, E}</td>
</tr>
<tr>
<td>i</td>
<td>A, B</td>
<td>{A, B}</td>
</tr>
<tr>
<td>j</td>
<td>D, E</td>
<td>{D, E}</td>
</tr>
</tbody>
</table>

by the functions f, g, and i. This relation can be succinctly captured along with
the other common sub-expressions found in a tabular form as given in Table 8.1.
Nodes identified in the braces are the common sub-expressions. The objective is to
identify these subsets of nodes from the approximations derived, such that they can
be shared between different functions.

The algorithm is an iterative two-step process. In the first step, a dependency
graph, as shown in Figure 8.4 is generated. Using this dependency graph, a weighted
adjacency matrix of the nodes that correspond to cubes is generated. Adjacency
matrix identifies the frequency of occurrence of a cube across all the functions under
consideration, and is useful in identifying the possible candidate cubes of common
sub-expressions. In this context, the two vertices are adjacent if there is a single-
hop path between two nodes, where the hop node denotes a function. For each
path identified, the matrix is updated with a unit weight of 1 added to the corre-
sponding entry in the matrix. In effect, we are computing the frequency of each
sub-expression in the sum-of-products expressions generated for an approximation.
Figure 8.5 illustrates the adjacency matrix generated for example shown in Figure
8.4.

Common sub-expressions are identified from the adjacency matrix. A pair of
nodes are candidates for a common sub-expression if their weighted adjacency sum
is greater than 1. In the example, nodes \{A, B\} and \{D, E\} are common sub-
expressions. Let the nodes \{A, B\} and \{D, E\} be identified as $\alpha_{\{a,b\}}$ and $\alpha_{\{d,e\}}$
respectively. It is important to note that the law of commutativity allows one to
choose either of the nodes \{A, B\} or \{B, A\} and one may choose not to compute
the matrix on the opposite side of the diagonal. To aid in redundancy elimination
and ease of computation, all the nodes are alphabetically ordered.

The second step of the algorithm replaces the common sub-expressions identified
in the frontier expression. If \( \alpha_x \) represents the identified common sub-expression
corresponding to a set of nodes \( x \), then \( \alpha_x \) can be substituted in a frontier expression
corresponding to a bit \( i \) as \( f_i \) if:

\[
f_i \cap x = x
\]

to generate a new frontier set given by:

\[
f_i - x \cup \alpha_x
\]

These new sets of frontier expressions are used to iterate until no more common
sub-expressions can be found. The nodes corresponding to an identified common
sub-expression are treated as any other node designated by a literal while generating
the new dependency graph. Figure 8.6 illustrates the updated relation graph.

The algorithm for computing the common sub-expressions for a given circuit
definition can be summarized as follows:

1. For each frontier approximation computed:
   (a) Identify the new product terms due to the progression of frontiers.
(b) Identify the common sub-expressions and remainder terms among all the frontier expressions for all the signal lines using the iterative algorithm described in this section.

(c) Update the association list to correlate the common sub-expressions identified with the frontier set that contains the sub-expressions.

2. Repeat Step 1 until the final approximation is reached.

Figure 8.3, after deriving the common sub-expressions can be represented as shown in Figure 8.7. The figure depicts the situation:

\[
f_1^1 \equiv CS_1 \cup r_1^1 \\
f_2^1 \equiv CS_2 \cup r_2^1
\]

where \( f_1^1, f_2^1 \) are the first and second approximations of \( f^1 \) respectively, \( CS_1, CS_2 \) are the common sub-expressions identified and \( r_1^1, r_2^1 \) are remainder terms. The common sub-expressions identified are subject to further optimization via factoring.

### 8.2.2 Factoring

The number of literals in a function being synthesized is generally correlated to the complexity of the function. The number of literals in a function directly translates
Figure 8.7: Nested circuit realization after identifying common sub-expressions.

into the number of transistors required in an implementation and provides a good indication of the total area of the layout. Factoring is aimed at simplifying a switching function to obtain a function equivalent to the original function with a minimum number of appearances of literals [18, 19].

To aid the presentation of the algorithm for factoring, it is important to define the terms:

**Definition 8.1** Given function $f$, let $p$, $q$, and $r$ be functions such that:

$$f = pq + r$$

- $p$ is called the divisor of $f$ if $r$ is not null and a factor if $r$ is null.
- $q$ is called the quotient of $p$.
- $r$ is the remainder.

In the approach to computing frontiers presented here, lattices are used as a means for representing the function. The properties of this lattice are exploited to derive the factored forms. More specifically, the product terms in a frontier set are partitioned such that nodes with the same parent node are grouped together. Using
the properties exhibited by the nodes derived from the same parent node, divisor, quotient and remainder terms and hence the factored forms can be easily derived. The approach provides two advantages, it:

- localizes the sub-expression which can be factored, and
- identifies if a factored form can be derived.

**Proposition 8.1** Given a frontier set, a factored form of the frontier expression exists between nodes that have at least one common parent node.

In the previous discussion, it was mentioned that a set of possible arguments to a function can be ordered to form a lattice. The arguments to a function are generally represented as cubes or a set of literals. A cube is said to be enlarged if some literals are dropped to expand the function space to include more minterms. As one moves down from the top element in the lattice towards the bottom, the parent-child relationship between the nodes in a lattice is exactly analogous to the operation of cube enlargement. Also, for any node in the lattice, the structural organization of the lattice succinctly captures all possible parent-child relationships. The foundation for the above proposition is that if two nodes were enlarged from the same cube then it is possible to have some literals in common. Identification of common literals within the product terms is key to factorization.

To fully appreciate the idea, consider the frontier set shown in Figure 8.8. The nodes in the frontier set are <1, 0, 1, 0> and <0, 1, 0, 1>. The parent node set for the node <0, 1, 0, 1> is \{<0, 1, 1, 1>, <1, 1, 0, 1>\} and \{<1, 0, 1, 1>, <1, 1, 1, 0>\} forms the parent set for the node <1, 0, 1, 0>. The frontier set corresponds to the expression \(ab + \overline{a}\overline{b}\) for an input vector given by \(<a, \overline{a}, b, \overline{b}>\) and factorization of this expression is not possible. Following the argument along the similar lines, if the node <0, 1, 1, 0>, which has a common parent node <0, 1, 1, 1> with <0, 1, 0, 1> were to exist in the frontier set, the proposition suggests that it is possible to find a factored form between nodes <0, 1, 0, 1> and <0, 1, 0, 1>. These nodes correspond to the expressions \(\overline{a}b\) and \(ab\). \(\overline{a}\) is the divisor term and, \(b\) and \(\overline{b}\) are the quotient terms.
Figure 8.8: An example to illustrate the localization of sub-expressions in a function.

Let \(<x_1, \ldots, x_n>, \ldots, <y_1, \ldots, y_n>\) be the m product terms that can be derived from the same parent node. Let \(S = S_1, \ldots, S_M\) be the set of cubes corresponding to each product term in the expression (recall that a cube can be viewed as a set of literals).

**Definition 8.2** A factor, \(F\), for a sum-of-products expression with \(S_1, \ldots, S_M\) as its cubes is the cube:

\[ F = S_1 \cap \ldots \cap S_M \]

\(F\) is empty when there exists no common literal. If a factor exists, then the expression can written as the product of the cubes:

\[ F \cdot \{S_1 - F, \ldots, S_M - F\} \]

where \(\cdot\) is a logical and with the resulting sum-of-products.

**Definition 8.3** If a divisor, \(D\), for a sum-of-product expression with \(S = S_1, \ldots, S_M\) as its cubes partitions the set \(S\) into sets \(S_q\) and \(S_r\) such that:

\[ \overline{D} \cdot S_q = \emptyset \]
$D \cdot S_r \neq \emptyset$

$s_q$ is the quotient and $S_r$ is the remainder and $D$ is the complement of $D$.

From the above definition, it is evident that the only distinguishing difference between a divisor and a factor is that a divisor results in a remainder set. This distinction is kept to allow for further optimizations where possible, as presented later in this section. If $D$ is the divisor of set $S$ resulting in $S_q$ and $S_r$ as the quotient and remainder sets, by definition, the expression can be written as:

$$D \cdot \{S_i \cap D, \forall S_i \in S_q\} + S_r$$

For a given expression, the divisor set is not necessarily unique. Each possible divisor of an expression results in different quotient and remainder sets. This poses a problem in choosing the best divisor that optimizes the given expression for logic synthesis. The approach is to identify the cube $D$ such that $S_q$ is maximal.

In a lattice, a parent node with $n + 1$ literals in its cube has $n + 1$ successor nodes. Each successor node has a cube of $n$ and its cube is contained by its parent. All the successor nodes differ from each other by only 1 literal. Thus each literal can exist in $n - 1$ cubes. Also, a combination of $(n - 1)$ literal pairs occurs at least $\frac{(n+1)}{2}$ if $n$ is odd and $\frac{n}{2}$ for even $n$. Thus, by choosing the size of the cube $D$ to be $n - 1$, the cube of the divisor and number of product terms that form the quotient set is maximized. If all the successor nodes in the given expression are true, there are $n - 1$ cubes which have the same literal. By choosing only 1 literal, $S_q$ is maximized such that all the $n - 1$ cubes containing the literal are included. Further optimization on the $S_q$ is discussed when the algorithm is presented.

The algorithm for deriving the factored forms consists of the following steps:

1. For a given set of nodes, for each element in the set, identify its parent node set. From this set, derive a minimal set of parent nodes that cover all the given nodes.
2. Derive the divisors, quotients and remainders as discussed before.

3. Recursively perform steps 1 and 2 on quotient and remainder sets until no more divisors can be found.

4. Iterate the steps 1, 2 and 3 for each element in the parent set.

The algorithm for deriving the factored forms is illustrated with an example. Consider a set of nodes \{<0, 1, 0, 1>, <0, 1, 1, 0>, <1, 0, 0, 1>, <1, 0, 1, 0>\} corresponding to the input vector <a, b, c, d>. These nodes result in the sum-of-product expression:

\[bd + bc + ad + ac\]

Figure 8.9 depicts the given nodes relative to their predecessor nodes. It is apparent that the minimal set of predecessor nodes that cover the given set of nodes are <0, 1, 1, 1> and <1, 0, 1, 1>.

The parent node <0, 1, 1, 1> has as its successors the nodes <0, 1, 0, 1> and <0, 1, 1, 0>. The cubes for these nodes are \{b, d\} and \{b, c\} respectively. Both these nodes share the common literal, b, which is extracted as a factor resulting in the expression:

\[\{b\} \cdot \{\{d\}, \{c\}\}\]

or in the nodal representation as:

\[<0, 1, 0, 0> \cdot \{<0, 0, 0, 1>, <0, 0, 1, 0>\}\]
Table 8.2: Summary of results obtained after factorization.

<table>
<thead>
<tr>
<th>Parent Node</th>
<th>Divisor</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0, 1, 1, 1&gt;</td>
<td>&lt;0, 1, 0, 0&gt;</td>
<td>{&lt;0, 0, 0, 1&gt;, &lt;0, 0, 1, 0&gt;}</td>
<td>Nil</td>
</tr>
<tr>
<td>&lt;1, 0, 1, 1&gt;</td>
<td>&lt;1, 0, 0, 0&gt;</td>
<td>{&lt;0, 0, 0, 1&gt;, &lt;0, 0, 1, 0&gt;}</td>
<td>Nil</td>
</tr>
</tbody>
</table>

Similarly, for the nodes <1, 0, 0, 1> and <1, 0, 1, 0> factored forms are derived as:

\[
\{a\} \cdot \{\{d\}, \{c\}\}
\]

and is notationally equivalent to:

\[
< 1,0,0,0 > \cdot \{< 0,0,0,1 >, < 0,0,1,0 >\}
\]

Table 8.2 summarizes the results obtained so far. As can be observed by noting the quotient terms, further optimization is possible. To formalize this observation, an algebraic factor is defined as:

**Definition 8.4** If \( F \) and \( G \) are factors of \( E \) and have the same quotient.

This definition leads to the final optimization performed which is presented in the following proposition.

**Proposition 8.2** Let \( \{P_1, \ldots, P_n\} \) be a minimal set of parent nodes that provide the cover for a sum-of-products expression \( E \). If the triplet \( (D_i, Q_i, R_i) \), of divisor, quotient and remainder for \( i = 1 \ldots n \) be the resultant of factoring such that:

\[
E = D_1 \cdot Q_1 + \ldots + D_n \cdot Q_n + R_1 + \ldots + R_n
\]

For any \( D_i = D_j, Q_i \) and \( Q_j \) are algebraic factors and the expression \( E \) can be minimized as:

\[
E = D_1 \cdot Q_1 + \ldots + D_i \cdot (Q_i + Q_j) + D_n \cdot Q_n + R_1 + \ldots + R_n
\]
and, for any $Q_i = Q_j$, $D_i$ and $D_j$ are algebraic factors and the expression $E$ can be optimized as:

$$E = D_1 \cdot Q_1 + \ldots + (D_i + D_j) \cdot Q_i + D_n \cdot Q_n + R_1 + \ldots + R_n$$

If $D_i = D_j$ and $Q_i = Q_j$, the product $D_i \cdot Q_i$ is redundant. Similarly, if $R_i = R_j$ then $R_j$ is redundant. Such redundant elements can be eliminated by the law of idempotency.

In the example, the final optimization of abstracting the divisors as algebraic factors results in:

$$\{<0,1,0,0>, <1,0,0,0>\} \cdot \{<0,0,0,1>, <0,0,1,0>\}$$

which is equivalent to:

$$(b + a) (d + c)$$

which is the factored form for the original expression:

$$bd + bc + ad + ac$$

### 8.3 Concluding Remarks

Multi-level logic synthesis is a powerful technique for combinational circuit optimization. The synthesis process consists of a sequence of transformations aimed at identifying sharable sub-expressions. This chapter has presented the transformations that are used to generate multi-level circuits. The approach utilizes the algorithms and data structures that were used in the synthesis of the original expression. Also, the use of lattice structure for representing a function and its use for factoring provides a systematic approach to identify and generate factored forms. This is in contrast to the existing factoring algorithms which are based on heuris-
tics and methods for which efficient algorithms are still the subject of investigation. 
This strengthens the view that the frontiers approach provides a natural and viable 
solution for circuit synthesis and their optimization.
Chapter 9

Synthesis of Sequential Circuits

Circuits considered so far in this thesis have all been combinational circuits — their output is a function of only their present inputs. This chapter considers sequential circuits in which the output depends not only on the present input but also on the past sequence of inputs. In effect, they are history sensitive, i.e. they must be able to “remember” something about the past history of the inputs to produce the present output. The synthesis technique described so far is inadequate to handle sequential circuits; no means has been provided to handle history information.

Finite-state machines provide a convenient way to model sequential circuits by encapsulating history sensitivity as state information. Based on the state information, finite-state machines respond to sequences of events that occur at discrete instants to produce an output. Finite-state machines are used as the model for the synthesis of sequential circuits. The task of sequential circuit synthesis is reduced to that of deriving the corresponding finite-state machine. The attention is restricted to synchronous models; sequencing is achieved through global timing. This chapter presents the synthesis of synchronous finite-state machines from the behavioral specification of sequential circuits.

Whereas the behavior of combinational circuits can be captured by logic functions, the behavior of sequential circuits is expressed by input and output sequences. FHDL, as defined in Chapter 5, is inadequate for modeling sequential circuits; no constructs are available to model sequences. Other high-level functional programming languages, such as Miranda [111] and Haskell [52], provide constructs with
well-defined semantics to model sequences in the form of lists or streams. This model is used to extend the semantics of FHDL to stream semantics. Instead of each function operating on a single set of arguments, functions operate on a sequence (stream) of input arguments. From the synthesized hardware perspective, the successive elements of the stream occupy the same spatial location, but arrive at that location at different instants of time [99].

This chapter extends the semantics of FHDL to include streams. A general model for implementation of a finite-state machine is presented. This forms the basis for understanding the synthesis process in terms of various functions or computations that need to be derived. A set of simple program transformations are presented, to derive various computational elements of a finite state machine. Using this scheme, that is unique to this thesis, the algorithms for the synthesis of combinational circuits and multi-level circuits described in the earlier chapters can be used for the synthesis of sequential circuits. The finite-state machine model is then extended to cover different classes of sequential circuits.

9.1 Model of Behavior

In general, an input of $p$ elements to a sequential circuit is given as an ordered sequence:

$$I = \{I_1, I_2, \ldots, I_p\}$$

Each $I_i$ is the input and consists of $x_1, x_2, \ldots, x_m$ as input variables. Similarly, the output sequence consisting of $z_1, z_2, \ldots, z_n$ as output variables is an output $O_i$ and is given by:

$$O = \{O_1, O_2, \ldots, O_q\}$$

Due to the non-terminating nature of circuits, input and output sequences are infinite. A formal language for specifying the sequential circuits, in addition to data
structures for specifying sequences, must support non-finite data structures.

Existing functional languages, for example Haskell [52] and Miranda [111], provide data structures for specifying ordered sequences. The sequences are unbounded. Data structures for defining such ordered sequences are called lists or streams; there is no difference between the two. The following subsection introduces the concept.

9.1.1 List Notation

Lists or streams provide the means to specify a sequence of elements. Unlike a set, the elements may occur more than once in a given list, and the order of elements is significant. This section introduces the notation used in this thesis.

Sequences are specified using the operator “:” (read as “cons”). Using the cons operator, a list can be constructed as:

\[ x : xs \]

\( x \) is the head of the list and \( xs \) is the tail which is a list containing the values of the type of \( x \). head and tail can be formally defined by the equations:

\[
\begin{align*}
\text{head} (x : xs) &= x \\
\text{tail} (x : xs) &= xs
\end{align*}
\]

A function \( f \) in FHDL, defined as:

\[ f (x : xs) = g x : f xs \]

produces an output corresponding to the value of \( g x \) and continues to operate on the tail generating a stream of output. The list is of arbitrary length. This is in line with the non-terminating nature of circuits [59, 99].
9.1.2 Extension to the Formal Semantics of FHDL

The semantic domain, V, containing data and functions was defined in Chapter 5 as:

\[ V = \text{Int} + \text{Bool} + (V \to V) \quad (9.1) \]

To extend the semantics of FHDL to stream semantics, product domains are included in the value domain.

**Definition 9.1** Let \( D_1, D_2, \ldots, D_n \) be domains of values, each domain consisting of values of the same type, then the product, \( D_1 \times D_2 \times \ldots \times D_n \), is the aggregation of elements to form an \( n \)-tuple of the form \((d_1, d_2, \ldots, d_n)\), where \( d_i \in D_i \) for all \( 1 \leq i \leq n \).

From the above definition, it is clear that the type of a tuple is determined by the number of its coordinates and their types. In contrast with the coordinates of a tuple, all entries of a list must be of the same type. Also, unlike a tuple, the type of a list is independent of the number of its coordinates. Thus, it is possible to have infinite lists. In this thesis, \( D_i \times D_i \) is used to denote an infinite list. This notation is used to extend the domain of FHDL to include streams.

An extension to the value domain of FHDL to include streams results in:

\[ P = \text{Int} + \text{Bool} \quad (9.2) \]

\[ V = P + (V \to V) + (P \times P) \quad (9.3) \]

With the inclusion of lists, following the conventions presented in Chapter 5, the complete abstract syntax for FHDL has the form:

\[
< \text{program} > ::= \text{Synthesize} < \text{expr} > \text{ with } \]

\[
[< \text{defn} >]_{\text{new line}} \quad (9.4)
\]

\[
< \text{defn} > ::= < \text{call-pattern} > \equiv < \text{expr} > \quad (9.5)
\]

130
The Valuation function for lists is given as:

\[
\mathcal{E} :: E \to \rho \to V
\]

\[
\mathcal{E}[e_h : e_t] \rho = (\mathcal{E}[e_h] \rho, \mathcal{E}[e_t] \rho)
\]

9.2 Model of Finite-State Machines

A synchronous finite-state machine is often modeled by a combinational circuit component and a memory component. The memory component provides the state information and the combinational logic realizes the input functions for the memory component and the output functions. The output of the memory component defines the present state of the machine. The inputs and the present state information are used to produce the output. Figure 9.1 illustrates the concept for a finite-state machine which accepts \(m\) inputs to produce \(n\) outputs, and uses \(k\) clocked D flip-flops as state memory. Drawing the model in this form emphasizes the presence of feedback – the flip-flop outputs are fed back as inputs to the combinational network. The combinational network realizes the \(n\) output functions and the \(k\) state functions, which serve as inputs to the D flip flops:

\[
z_1 = f_1(x_1, x_2, \ldots, x_m, q_1, q_2, \ldots, q_k)
\]

\[
z_2 = f_2(x_1, x_2, \ldots, x_m, q_1, q_2, \ldots, q_k)
\]
When a set of inputs is applied to the circuit, the combinational network generates the outputs $z_1, z_2, \ldots, z_n$ and the inputs to the memory component $q_1^+, q_2^+, \ldots, q_k^+$. When a clock pulse is applied, the flip-flops change to the proper next state. This process is repeated for each set of inputs. The functions $q_1^+, q_2^+, \ldots, q_k^+$ are called state assignment functions. The synthesis of state assignment functions is dependent on the choice of register type used for storage (e.g., D, T, JK). Only D-type registers are considered here because of their abundant usage.
The synthesis of sequential circuits presented here consists of identifying the functions $f_1, f_2, \ldots, f_n, g_1, g_2, \ldots, g_k$. Hardware is derived for these functions by computing their fixpoints. The synthesized hardware is then instantiated appropriately in the general model for the finite-state machine.

9.3 Semantics of Sequential Machines

Sequential machines can be described as state machines whose input assume values from the input set $I$ and state set $Q$ to produce output values from the set $O$ and next-state from the set $Q$. The behavior can be described in terms of sequences belonging to $\langle I \rangle$ and $\langle O \rangle$. Each state machine $M$ can be concisely described as a triple (next-state function, output function, initial-state). The type of machine can be given as:

$$M :: (Q \rightarrow I \rightarrow Q, Q \rightarrow I \rightarrow O, Q)$$

With these conventions, a typical state machine can be described as:

$$\text{machine} = (\text{next}, \text{out}, \text{start})$$

where

$$\text{next} q i = \langle \text{expression} \rangle$$

$$\text{out} q i = \langle \text{expression} \rangle$$

$$\text{start} = \langle \text{expression} \rangle$$

$$\text{next\_state\ machine} q i = (\text{first\ machine}) q i$$

$$\text{output\ machine} q i = (\text{second\ machine}) q i$$

$$\text{start\_state\ machine} q = (\text{third\ machine}) q$$

$$\text{first} (x, y, z) = x$$

$$\text{second} (x, y, z) = y$$

$$\text{third} (x, y, z) = z$$

133
From the above definitions, the behavior of an arbitrary machine $f$ started in state $q$ can be obtained by means of a general function $\text{gen_seq_machine}$ as:

$$\text{gen_seq_machine} :: M \rightarrow Q \rightarrow \langle I \rangle \rightarrow \langle O \rangle$$

$$\text{gen_seq_machine}\ f\ q\ (h:\ t) = \text{output}\ f\ q\ h : (\text{gen_seq_machine}\ f\ (\text{next_state}\ f\ q\ h)\ t)$$

The definition of $\text{gen_seq_machine}$ constitutes a complete definition for behavioral specification for sequential machines. This definition is used to derive the definitions for $\text{output}$ and $\text{next_state}$, which are output functions and next-state functions, from FHDL definitions. The hardware model of a general finite-state machine is used to derive the interconnects for hardware synthesis.

### 9.4 Synthesis of Function Definitions

The approach to the synthesis of sequential circuits was identified in the previous section. This section outlines the procedure to synthesize the definitions for $\text{output}$ and $\text{next_state}$ functions. In the following sections, the general model of the finite-state machines is extended to handle wider classes of sequential circuits.

#### 9.4.1 Synthesis of Output functions

In order to formally describe an output expression, the following terms are defined:

**Definition 9.2** Terminal terms are defined inductively by:

- $c \in C$, where $C$ is the set of constants,
- $x \in X$, where $X$ is the set of identifiers,
- if $f$ is an $n$-place operator and $t_1, t_2, \ldots, t_n$ are terminal terms, then $f(t_1, t_2, \ldots, t_n)$ is a terminal term, and
• if $F$ is a function variable symbol of dimension $n$ and $t_1, t_2, \ldots, t_n$ are terminal terms, then $F(t_1, t_2, \ldots, t_n)$ is a terminal term.

**Definition 9.3** An expression is defined inductively by:

- all terminal terms are expressions,
- if $f$ is an $n$-place operator and $e_1, e_2, \ldots, e_n$ are expressions, then $f(e_1, e_2, \ldots, e_n)$ is an expression, and
- if $F$ is function variable symbol of dimension $n$ and $e_1, e_2, \ldots, e_n$ are expressions, then $F(e_1, e_2, \ldots, e_n)$ is an expression.

**Definition 9.4** An expression over the identifiers $x_1, x_2, \ldots, x_n$ is one that contains no other identifiers other than $x_1, x_2, \ldots, x_n$.

**Definition 9.5** An expression is trivial if it contains no identifiers.

**Definition 9.6** An expression is called serious if it contains the function variable symbol.

The behavioral definition of the abstract sequential machine forms the basis for a sequential definition in FHDL.

**Definition 9.7** A sequential definition in FHDL assumes the form:

$$F \ x_1 \ldots \ x_n \ (h : t) = e : F \ e'_1 \ldots \ e'_n \ t \quad (9.14)$$

where $F$ is the function variable symbol, $x_1, \ldots, x_n$ are the state variables, $h$ and $t$ correspond to the head and tail of input stream, $e$ is the output expression and $e'_i$ is defined over $g, x_1, \ldots, x_n$ and $h$, for any defined $g$ and $1 \leq i \leq n$.

**Definition 9.8** Given a sequential specification of the form given by Equation 9.14, the immediate output is the result of evaluation of $e$, iff:

- $e$ is an expression defined over $x_1, \ldots, x_n$ and $h$,
• \( e \) is trivial, and

• \( e \) is not a serious expression.

**Proposition 9.1** Given a sequential function \( F \), as defined by Equation 9.14, there exists a function \( F' \) defined over \( x_1, \ldots, x_n \) and \( h \):

\[
F' \ x_1 \ldots \ x_n \ h = e
\]

such that

\[
F' \ x_1 \ldots \ x_n \ h : F \ x'_1 \ldots \ x'_n \ t = e : F \ x'_1 \ldots \ x'_n \ t
\]

\( F' \) can be derived from \( F \) in a straightforward manner; the specification of input stream in \( F \) is recasted to the desired form. The output expression is the rewrite of the immediate output component of the given specification for the output stream such that it satisfies Definition 9.8.

**Proposition 9.2** The function \( F' \ x_1 \ldots \ x_n \ h = e \) synthesized from \( F \) realizes the output function for the sequential machine.

The concepts discussed so far are illustrated by considering an example. The example presented here is the parity detector discussed in [42]. The circuit computes the parity check for every 10 bits and outputs the result. The specification for the circuit and the transcription of the specification to synthesize the output function are shown in Figure 9.2. In the example, following the proposition discussed earlier in this section for the synthesis of an output function definition, only a single case is applicable for specifying the immediate output as shown in Figure 9.2 (b). The purpose of the rest of the definitions and their usefulness in sequential circuit synthesis is discussed in the following sub-section.
Figure 9.2: An example of deriving the output function specification.

9.4.2 Synthesis of Next-State Functions

A definition for a sequential machine in FHDL, given by Equation 9.14, can be expressed in words as: "a function $F$ over the state variables $x_1, \ldots, x_n$ and a stream with $h$ as its head and following input defined by $t$ produces an output which is the result of evaluation of $e$ and continues with $t$ in its modified state". The sequencing of control implied by the word "continues" can be formalized in programming languages as continuations [103]. Continuations define the future course of computation. Continuations inherit values from the present and record obligations for the future. In effect, they record state transformations.

Given a recursive function definition as:

$$F(x) = K(x), \text{ if } p(x)$$
$$= F(G(x)), \text{ otherwise}$$

for any defined functions $K$ and $G$, and predicate $p$, then $K$ is the continuation if $p(x)$ is true. Otherwise, $F$ computes $G$ applied to $x$ with $F$ as its continuation. Given such a general recursive function definition, Johnson’s thesis [59] provides a schematic interpretation for such definitions as shown in Figure 9.3. $x^0$ stands for the initial value and $DR$ stands for a data register. The feedback loop is due to the recursive call in the function definition. The signal at the output of $F$ is the desired result when the output signal of $p$ is true. The output value of the signal measured at "•" is $G(x)$, which is a continuation of $F$ when $p(x)$ is false. The portion of the
Figure 9.3: Schematic interpretation of a recursive definition.

circuit within the dotted lines implement the function:

\[ F(x) = F(G(x)) \]

This is exactly the model proposed for the implementation of sequential machines, shown in Figure 9.1. This forms the basis for deriving the next-state functions for the synthesis of sequential machines.

Based on the concept of continuations, the interpretation of a sequential definition can be refined as: "\(F\) sends the result of evaluation of \(e\) to its output. It computes \(t\) with its continuation whose inputs are defined by the expressions \(x'_1, \ldots, x'_n\)." In other words, the continuation function records the need for future computation and the input arguments to the continuation function define the state in which the function is evaluated, i.e. the expressions that form the arguments to a continuation function define the state transitions.

**Definition 9.9** Given a recursive definition \(F\), if:

\[ F \ e_1 \ e_2 \ldots \ e_n \]
**Definition 9.10** Given a sequential function definition, as defined by Equation 9.14, for the continuation $F e'_1, e'_2, \ldots, e'_n$, $t$ is called a trivial state expression and the expressions $e'_1, e'_2, \ldots, e'_n$ are non-trivial state expressions.

From the previous discussion, it is apparent that non-trivial state expressions of a sequential function definition provide the necessary means to specify the state information and manipulation. These non-trivial state expressions are used to derive the next-state functions of a sequential circuit.

**Proposition 9.3** Given a sequential definition (Equation 9.14), the functions $F_{x_1}, \ldots, F_{x_n}$ defining the next-state functions can be derived from $F$ as:

\[
\begin{align*}
F_{x_1} x_1 \ldots x_n h &= e'_1 \\
F_{x_2} x_1 \ldots x_n h &= e'_2 \\
& \vdots \\
F_{x_n} x_1 \ldots x_n h &= e'_n
\end{align*}
\]

Figure 9.4 illustrates the two next-state functions, $f_y$ and $f_z$, derived from the original definition for the parity detector. $f_y$ counts the number of values read from the input stream and is reset to zero when the count reaches 10. $f_z$ tracks the parity computed so far. The output is 1 for odd parity and 0 for even parity. It is set to zero when the count has reached 10.
9.4.3 Handling Initial State

A sequential circuit is a state machine that traverses through a succession of states producing an output in response to an input. So far, the question of the initial state of the machine prior to the consumption of the input sequence was not addressed.

The initial state of the registers after the hardware is generated is unknown and is generally assumed to be in don’t care condition. The user must provide the initial state of the data registers through an initialization sequence. This is consistent with the denotational description for sequential machines, where initial state \( q \) is passed as an argument to the sequential definition. More strictly, taking into account the initial state, the behavior of an arbitrary machine \( m \) is given using the definition of \( \text{gen}_\text{seq}_\text{machine} \) as:

\[
\text{Behav } m = \text{gen}_\text{seq}_\text{machine} m (\text{start } m)
\]

The initial state of the data registers can be deduced from the specifications and initialized accordingly. Such a step would imply a circuit that is self-starting. A self-starting circuit is one that can be forced into a predetermined state. To maintain a consistent perspective between the semantics and the synthesis hardware, it is
necessary to provide the initialization of the circuit through a finite sequence of inputs.

Additional hardware accompanies the data registers to facilitate proper initialization. Figure 9.5 illustrates the hardware generated per data register. By setting init to true, one can force the state of registers. During the normal operation, init is false; the registers are updated by the output of the next-state functions.

9.5 Extensions to Sequential Circuits

In the model for sequential circuits presented here, an implicit assumption was made that the input and output sequences are continuous, i.e. every element of the input sequence produces a valid output. Examples abound in the real world that do not fulfill this assumption, including the example presented earlier in this chapter. Given an input sequence and the specification for a sequential circuit, the output sequence can be:

- Continuous, i.e. for every element in input sequence produces a valid output.
- Periodic, i.e. an output is produced after processing (consumed or ignored) a finite number of input elements.
- Intermittent, i.e. an output is valid only in certain states of the state machine.

This implies that the synthesized circuit must contain a signal that indicates when the output value of the sequential machine is valid. This is accomplished by the circuit Ready?, as shown in the extended sequential abstract machine shown in Figure 9.6. Synthesis of specification for the Ready? circuit follows.

**Definition 9.11** A circuit description realizes a specification if and only if it has a signal Ready, that states when the output is valid, and the output contains the desired value when Ready is true.
The signal Ready assumes the values true or false. The definition for the circuit Ready? can be synthesized in a straightforward manner from the definition of $F'$ based on the following proposition:

**Proposition 9.4** Let $F_{\text{ready}}$ be the function definition for a circuit that implements the Ready? component of the sequential abstract machine. Given a sequential function $F$, let $F'$ define output function of $F$ as defined before. $F_{\text{ready}}$ is true for all the conditions $F'$ is defined and is false otherwise.

The definition for $F_{\text{ready}}$ is derived by rewriting the output definitions $F'$; $F_{\text{ready}}$ returns true wherever the output expression $e$ is defined for the function $F'$.

Figure 9.7 illustrates the definition synthesized for $F_{\text{ready}}$ for the parity detector example discussed earlier. The output is valid only when the value of $y$ has reached 10 and is invalid otherwise. This is consistent with the initial specification, where
Figure 9.7: Function definition synthesized for Ready? circuit.

the parity is computed for every 10 bits.

9.6 Synthesis of Sequential Circuits

An abstract sequential machine model consists of circuits to:

- produce an output,
- generate the next-state of the state-machine, and
- validate the output.

The approach to the synthesis of sequential circuits consists of deriving the function definitions for each of the corresponding functional entities following the transformations of the original behavioral specification for sequential circuits discussed in this chapter. The following steps summarize the process for deriving sequential circuits from the behavioral specifications:

- derive the behavioral definition for output functions, next-state functions and Ready function,
- generate the hardware for each of the function definitions by computing the fix points. The hardware descriptions are generated following the conventions described in the previous chapters for combinational circuits, and
- instantiate the derived hardware components appropriately in the model for abstract sequential machine.
9.6.1 An Example

This section presents an example to illustrate the concepts discussed in the previous section. The problem modeled here is a part of a wireless basestation receiver; its functionality is described as:

A diversity switch is hooked up to two antennas, primary and diversity antennas. The switch forwards the input from an antenna to signal processors if the confidence level of the signal at the receiving antenna is greater than a prescribed level. Also, the received data is forwarded to signal processors only if a synchronization word is detected. Synchronization is lost if the confidence level of the signal at both the receiving antennas falls below the prescribed level.

This section illustrates the synthesis of specifications for the combinational circuit output, derivation of the next-state functions, and the specification for the Ready? function described earlier in this chapter.

For the sake of simplicity, assume 01010101 to be the synchronization word and the minimum confidence level is assumed to be 5 for each channel. Figure 9.8 shows the FHDL specification for the diversity switch problem. Four streams form the inputs to the diversity switch, two streams of inputs from the antenna and two streams of inputs corresponding to the confidence level of the signal for each of the input stream. A boolean state variable tracks the synchronization status.

The output of the diversity switch is a single stream, an input from either antenna based on the confidence level. The first step in the synthesis of sequential circuits is to derive the specification of the combinational block that generates the output. From the presentation of the synthesis of output functions in Section 9.4.1, the specification for the output function can be synthesized from the given definition by identifying the components of the given specification that generate an immediate output. The resultant definition for the output component block of the sequential circuit synthesized from the original specification is shown in Figure 9.9. This is consistent with the diversity switch specification, where an output is generated by
the diversity switch only if the signal confidence level for an input is met and the synchronization word has been detected.

The next step in the synthesis of sequential circuits is the identification of next-state functions from the given specifications. The underlying theory and the approach to the synthesis of next-state functions was presented in Section 9.4.2. This forms the basis for deriving the next-state functions. The synthesized definition for the next-state function based on the original problem specification is shown in Figure 9.10. The only state variable in the definition of the diversity switch is the synchronization status. The synchronization status is set to \textit{true} if the synchronization word is detected with an appropriate confidence metric value on either of the channels and is unchanged until it is reset to \textit{false} when signal confidence level falls below the defined level on both the input streams. This matches the functional description of the diversity switch.

The final step in the synthesis of definitions for generating hardware for sequential circuits is to validate the output through the \textit{Ready} signal. From the concepts
Figure 9.10: Next-state function definition for the diversity switch.

\[
\text{diversity}_{\text{syn}}(x:x, x_{\text{cnf}}:tx, y:y, y_{\text{cnf}}:ty) \begin{cases} 
\text{syn} = \text{syn}, & \text{if } ((x \neq 85) \text{ and } (y \neq 85) \text{ and } (\text{syn} = \text{False})) \\
= \text{True}, & \text{if } (((x = 85) \text{ and } (x_{\text{cnf}} \geq 5)) \text{ or } ((y = 85) \text{ and } (y_{\text{cnf}} \geq 5)) \text{ and } (\text{syn} = \text{False})) \\
= \text{False}, & \text{if } ((x_{\text{cnf}} < 5) \text{ and } (y_{\text{cnf}} < 5)) \\
= \text{syn}, & \text{if } ((x_{\text{cnf}} \geq 5) \text{ and } (\text{syn} = \text{True})) \\
= \text{syn}, & \text{if } ((y_{\text{cnf}} \geq 5) \text{ and } (\text{syn} = \text{True}))
\end{cases}
\]

Figure 9.11: Ready? function definition for the diversity switch.

\[
\text{diversity}_{\text{ready}}(x:x, x_{\text{cnf}}:tx, y:y, y_{\text{cnf}}:ty) \begin{cases} 
\text{syn} = \text{True}, & \text{if } ((x_{\text{cnf}} \geq 5) \text{ and } (\text{syn} = \text{True})) \\
= \text{True}, & \text{if } ((y_{\text{cnf}} \geq 5) \text{ and } (\text{syn} = \text{True})) \\
= \text{False}, & \text{otherwise}
\end{cases}
\]

presented in Section 9.5, the definition for the Ready? circuit is synthesized as shown in Figure 9.11. The output is valid only if the signal received on one of two channels meets the confidence level metrics and the receiver has acquired synchronization. This corresponds with the synthesized definition.

Hardware generation for the diversity switch is accomplished by computing the fixpoints for the derived definitions, \textit{diversity}_{\text{out}}, \textit{diversity}_{\text{syn}}, and \textit{diversity}_{\text{ready}}, as described in the earlier chapters of this thesis. The synthesis process is complete when the hardware generated is instantiated in the model for sequential circuits, i.e. all the required interconnections are generated. Figure 9.12 illustrates various components synthesized and their placement in the abstract model. Hardware is initialized at runtime using the Init signal that is generated during the synthesis process. The initialization step involves clearing the data register (set to False). This circuit implements the specification of the diversity switch as described earlier in this section.
Figure 9.12: Inter-connection for the diversity switch.

9.7 Concluding Remarks

A finite state machine is an abstract model describing sequential machines. The abstract model for sequential sequential machine can be categorized into the functional components: output circuit, next-state circuit, memory component and output validation circuit. The behavioral specification of a sequential machine encapsulates the behavior of all these components. The approach to the synthesis of sequential circuit consists of extracting the definition for each of the corresponding component from the given specification individually and deriving the hardware descriptions by computing the fix points as described in this thesis. Sequential circuits are generated by instantiating these derived components in the proposed abstract model.
Chapter 10

Conclusions

The objective of the research presented in this thesis is to illustrate the viability of functional programming languages for specification and synthesis of digital circuits. A high-level specification language, FHDL, based on the functional programming paradigm was developed for this purpose. Contrary to the approaches developed so far, which are based on either deriving control and/or data flow graphs in the case of imperative specification languages or transformational based approaches in the case of applicative specification languages, a new method based on denotational semantics is presented for the synthesis of digital circuits from FHDL specifications. The techniques presented are extended to the specification and synthesis of multi-level and sequential circuits. This chapter summarizes the work presented in this thesis and explores avenues for further research.

10.1 Review

10.1.1 Specification Language

Functional programming is a rapidly advancing programming paradigm where the notion of function is the primary building block for programs. Programs consist entirely of functions; the main function is defined in terms of other functions and this process of function definition continues until at the lowest level where functions are defined in terms of language primitives. The simple and precise nature of semantics for these languages can be attributed to their referential transparency, i.e. they have
FHDL is a simple high-level functional language used to specify algorithms intended for hardware implementation. A program in FHDL is an expression that maps objects to objects. The objects can be atomic (e.g., numbers) or sequences of objects (e.g., lists). FHDL has a set of primitive functions defined, consisting of arithmetic, logic and relational functions. Programs in FHDL are built on these primitive functions and conditional constructs. Two forms of conditional constructs are available; the standard if ... then ... else form of conditional expression is supported as guarded expressions where the guard determines the branching control, and pattern matching expressions. A program in FHDL consists of a set of function definitions, together with an expression to be synthesized. Figure 10.1 illustrates a typical definition in FHDL for a traffic light controller problem.

The traffic light controller problem is a standard benchmarking example for high-level synthesis tools\textsuperscript{1}. The stated functionality (as obtained from the source in verbatim) is as follows:

Highway light is green normally. When there is a car on the side road and the highway light has been on for 'timeoutL' time, the highway light turns yellow. 'timeoutS' time after it has turned yellow, the highway light turns red and the side road light turns green.

If there are no cars on the side road or the side road light has been on for 'timeoutL' time the side road light turns yellow. 'timeoutS' time after it has turned yellow, the side road light turns red and the highway road light turns green.

I guess the above timer signals would be connected to some external timer circuit. The startIimer would start the external timer circuitry every time the state changes. The TimeoutL and TimeoutS should originate

\textsuperscript{1}These benchmarks were developed by MCNC and are publicly available from http://www.mcnc.org/
The circuit generates two outputs to control the lights on a highway and a side road (highway_light_state, sideroad_light_state). The state variables are red = 0, yellow = 1 and green = 2.

\[
\begin{align*}
\text{TLC} & \quad \text{htL:ttL htS:ttS hPr:tPr Hst Fst} \\
& = (1, 0) : \text{TLC t}\text{tL t}\text{tS tPr }1 0, \\
& \quad \text{if (Hst }= 2) \text{ and (Fst }= 0) \text{ and (htL }= 1) \text{ and (hPr }= 1) \\
& = (0, 2) : \text{TLC t}\text{tL t}\text{tS tPr }0 2, \\
& \quad \text{if (Hst }= 1) \text{ and (Fst }= 0) \text{ and (htS }= 1) \\
& = (0, 2) : \text{TLC t}\text{tL t}\text{tS tPr }0 2, \\
& \quad \text{if (Hst }= 0) \text{ and (Fst }= 2) \text{ and (htL }= 1) \text{ and (hPr }= 1) \\
& = (0, 1) : \text{TLC t}\text{tL t}\text{tS tPr }0 1, \\
& \quad \text{if (Hst }= 0) \text{ and (Fst }= 2) \text{ and (htL }= 1) \\
& = (2, 0) : \text{TLC t}\text{tL t}\text{tS tPr }2 0, \\
& \quad \text{if (Hst }= 0) \text{ and (Fst }= 1) \text{ and (htS }= 1) \\
& = \text{TLC t}\text{tL t}\text{tS tPr Hst Fst, otherwise}
\end{align*}
\]

Figure 10.1: FHDL description of traffic light controller.

from the external timer. Apparently, the model does not describe how this works.

In this model, the highway light is green usually. When there is a car on the side road, the side road light is green till max 'timeoutL' time and then goes back to red irrespective of whether there are cars on the highway. In fact, the model never checks for cars on the highway. Thus, the highway has priority over the side road.

Appendix A presents the corresponding VHDL code obtained from MCNC for the traffic light controller. The example clearly demonstrates the expressive power of FHDL, and functional programming languages in general, over the conventional imperative languages.

10.1.2 Recursion Removal and Fixpoints

Functional programming languages support recursion on the basis of their computational model, the \(\lambda\)-calculus. At the specification level, recursion is an important
conceptual tool in the design of algorithms. Recursion needs to be eliminated during hardware synthesis in order to avoid the necessity of a stack. Recursion removal is accomplished by computing the fixpoints of the recursion equation using an iterative approach. This iterative approach, based on domain theory, replaces recursive specifications with their equivalent non-recursive fixpoints. Given a recursive function \( r \), the fixpoint \( \phi \) can be computed as a limit of a sequence of approximations to the function \( r \) as \( r_0, r_1, \ldots \), starting with the function that maps every argument to the bottom (\( \bot \)) value of the domain. The series of approximations generated as:

\[
\begin{align*}
\tau_0 &= \bot \\
\tau_1 &= \phi(\tau_0) \\
\tau_2 &= \phi(\tau_1) \\
\vdots
\end{align*}
\]

until the series converges such that:

\[
\tau_i = \tau_{i+1}
\]

\( \tau_i \) is the least fixpoint and the sequence of approximations form the *Ascending Kleene Chain* (AKC). Each approximation leads a step closer to the solution of the fixpoint equation. This approach has two limitations:

- equality on functions must be defined, and
- to compare two functions for equality it is necessary to check that the functions coincide at every point in their domain. A naive approach to performing such comparisons evaluates both functions at every point in their domain. For a \( p \)-point domain, a function of \( n \) arguments requires \( p^n \) evaluations.

This thesis uses *frontiers*, a canonical representation of a function, to determine the equality of functions. In a domain \( T (0, 1) \), the set of all possible arguments to function form a finite lattice. Figure 10.2 depicts the graphical representation of the lattice for \( T^2 \). Such a lattice represents a function if each node is labeled with the value of the function at that point. If the function is monotonic, 1-nodes will never
lie below 0-nodes. The set of 1-nodes that lie on the boundary form the 1-frontier set and vice versa. Using this canonical representation the equality of two functions reduces to the comparison of two frontier sets.

10.1.3 Frontiers Algorithms and Hardware Synthesis

The synthesis method presented in this thesis applies frontiers to boolean function domain $2 (0, 1)$. The frontiers approach originally proposed in the literature is not directly applicable to boolean function domain. Boolean functions can be non-monotonic if they contain a negation (not) operation. This problem is circumvented by domain inflation. By domain inflation, the negated literals of a boolean function are included in the lattice nodes and input vectors. For example, a function $f$ with two arguments $a$ and $b$ in an inflated domain has $\langle a, \overline{a}, b, \overline{b} \rangle$ as its nodes and input vector.

The original algorithms for computing the frontiers are search based. For each approximation in the AKC, a frontier set is computed by evaluating the current approximation of the function. A number of techniques which aim at optimizing the search are available. Still, all search based algorithms are computationally expensive and in the worst case can degenerate to the second limitation for determining the equality of functions that was stated earlier. The work presented here overcomes this restriction by computing the frontiers from the abstract syntax of the defining function without evaluating the function at all, resulting in substantial improvements in the performance of algorithms for computing the fixpoints. The algorithm presented
in this thesis constructs 1-frontier set by examining expressions and replacing the expressions with a 1-frontier set corresponding to the expression using a set of rewrite rules. The algorithm neither performs any evaluations of the expression nor searches the domain for the frontier set which is the source for performance gains.

The main motivation for the use of frontiers algorithms is two fold. In addition to allowing the computation of the equality of functions needed for identifying the fixpoint of a recursive function, the 1-frontier set of an expression directly corresponds to its minimal sum-of-products expression in an inflated domain. The minimality of a sum-of-products expression comes from two sources: (i) minimum number of product terms and, (ii) minimum number of literals in each product term, per se an expression which is irredundant and prime. It is shown in this research work that each node in the frontier set is a prime implicant. The sum-of-products expression, derived using frontiers, can be further minimized in the original domain using the existing logic minimization tools by rewriting the frontier set into a form suitable as an input to a logic minimization tool. The canonicity of representation allows the transformation of optimized sum-of-products to frontier representation in a straightforward manner.

The frontiers algorithms presented in this thesis work are applicable in boolean function domain 2 \((0, 1)\). In order to use the frontiers algorithms, the specifications in FHDL are transformed into an intermediate language called the Kernel Language (KL), an embellishment of boolean function domain. KL is defined on domain 2 with and, or and not as its primitive operations. The structure preserving transformations of the translation scheme map all high-level operations (functions) to primitive operations using only and, or and not operators. In addition, a function, \(f :: I \to O\), in FHDL is split into \(n\) functions in KL, \(f_i :: I^n \to O_i\) for \(1 \leq i \leq n\), where \(I^n\) is the \(n\)-bit-wise representation of the input and \(O_i\) corresponds to the \(i^{th}\) output bit. The frontiers algorithm is then applied to each bit-wise function to compute the fixpoints.
10.1.4 Back-end Tool Support

The fixpoints computed using the frontiers algorithms are the SOP expressions for the given specification. This thesis work presents two tools for converting the frontier sets to descriptions acceptable to layout tools: the PLA generator and the structural VHDL code generator for use with Electric VLSI layout system.

PLA provides a regular structure for implementing combinational logic functions using AND-OR structure which is ideally suited for a sum-of-product realization. The code generation scheme implemented supports both NMOS and CMOS PLA generator supported by the Electric design system.

Structural specification in VHDL entails specifying the interconnection of components at the gate level. The primary interest in generating VHDL code is the wide range of platforms that support this language. The algorithms for generating the VHDL code along with implementation were detailed in this work that transform SOP expressions to corresponding VHDL description. The support for VHDL code generation is extended to multi-level and sequential circuit synthesis forms.

10.1.5 Circuit Optimizations

In this thesis work, in addition to two-level synthesis using sum-of-product expressions, the properties of the progression of frontiers and AKC are utilized to derive naturally nested structures. The resultant circuits can have unlimited number of intermediate signals. In general literature these circuits are called multi-level logic circuits. The synthesis process includes two-levels of optimization. The first step of optimization identifies the common sub-expressions among the multiple fixpoints synthesized, one for each bit-wise split function. The common sub-expressions identified are subject to further optimization via factoring. The properties of the lattice are exploited to identify the factored forms. The algorithm presented can identify if a factored form can be derived and if a factored form exists it is identified. Unlike the algorithms presented in the literature, the search process can be abandoned if no factored form exists and, in general, can lead to enhanced performance metrics.
10.1.6 Sequential Circuits

Unlike combinational circuits that accept an input to produce an output, sequential circuits work on a sequence of inputs or streams. The output is not guaranteed to be immediate (though an output is generated, it may not be the desired output). Sequential circuits also have the property that the output depends not only on the present input but also on the past sequence of inputs. Circuits must maintain the history information to produce correct outputs.

FHDL presents a way to model streams using list notation. This model forms the basis for modeling input and output sequences. Finite state machines are used as a general scheme for implementing sequential circuits. History information is maintained through state transitions. A finite state machine contains 3 components: (i) combinational circuit to produce the current output, (ii) memory component to store the current state information, and (iii) logic necessary for generating the next state information. A specification for sequential circuit in FHDL encapsulates the information required for synthesis of current output and next-state functions. A set of transformational rules are presented to isolate the required functionality for the synthesis of individual components of finite-state machine. D-types flip-flops are used as memory components. The synthesis process consists of computing the fixpoints for the current output function and next-state function, and generating appropriate interconnections with the D-type flip-flops. To handle the cases where the output stream is not synchronous with the input, an output qualifier signal, Ready?, is introduced. The specification for Ready? circuit is synthesized from the given specification following the transformational rules and circuit generation is based on computing fixpoints using the frontiers algorithm. The output of a sequential circuit is valid only if Ready? is true. The synthesized finite-state machine implements the given FHDL sequential circuit specification.
10.2 Prospects for Research

This thesis has addressed the fundamental question of applicability of functional notation for the specification and synthesis of digital circuits. In this section, extensions which could benefit the work presented here are discussed.

10.2.1 Communication and Asynchronous Circuits

In this thesis, all the examples presented assumed certain degree of autonomocity, requiring no interaction with other hardware systems or subsystems. In reality, if each subsystem is treated as a process, realization of an entire system results in communicating hardware processes (CHP) along the lines of programming model presented by C. A. R. Hoare [49]. If the realization is a network of CHP, there is a need to add means for inter-process communication to the specification language. The stream based I/O model presented here is easily extensible for message passing scheme.

Introduction of communication primitives in a hardware specification language, in addition to opening the realm of network architecture synthesis, protocol synthesis, and dealing with non-determinism, to the hardware synthesis area, it will also open avenues for modeling asynchronous circuits.

In the past few years, there has been resurgence of interest in asynchronous circuits [72, 112]. Though synchronous designs have enabled great strides to be taken in the design and performance of modern computers, there is a fundamental limitation – all the subsystem have to be synchronized to a global clock. Clocks are electrical signal and are subject to delays due to propagation. With clock speeds approaching 500Mhz, these delays become more crucial to the behavior of the synchronous circuits. Due to their potential for speed, low power consumption and ease of design, there is growing interest in academic and industrial research houses in the design of asynchronous circuits.

There are many flavors of asynchronous logic, each distinguishing itself by its tolerances to gate delays (speed independent), propagation delays (self-timed) or both
At the implementation level, asynchronous circuits use signalling mechanisms to signal the availability of data. For example, Ivan Sutherland, in his Turing award lecture, presented an approach called “Micropipelines”, using two control wires: Request from the sender to receiver carries a transition when data is valid; Acknowledge from the receiver to the sender carries a transition when the data has been accepted [105]. This protocol defines only the sequence in which the events must occur – there is no upper bound on delays between consecutive events. By extending FHDL to include the communication primitives, it would then be possible to model asynchronous circuits at behavioral level of abstraction.

10.2.2 Partitioning of Control

The approach to the synthesis described here accepts the given set of specifications, performs a dependency analysis to identify the components that are needed to synthesize a given function, and computes the fixpoints based on the dependency graph replacing the occurrence of the function name with the derived fixpoint until the final function that need to be synthesized is realized. Though the approach is shown to realize correct circuits with respect to the specification, as the complexity of circuit specification increases it may not yield most efficient realization with respect to the timing performance. This can be attributed to the lack of any constructs at the specification level to impose structural constraints at the realization level and, absence of any attempt to partition the problem into smaller, manageable subsystems.

FHDL is a first-order specification language. By extending the language to include higher-order functions along the lines of the work explored by Sheeran [99], it is possible to implement structural constraints at the realization level. Under the proposed scheme, all higher-order functions are given a geometric interpretation the realization maps to. For example, a higher-order FHDL specification:
synthesize \( f \circ g \) with
\[
\begin{align*}
f & \rightarrow g \\
g & \rightarrow f
\end{align*}
\]
has an interpretation "do \( g \) and then \( f \)" and its geometric interpretation is shown in Figure 10.3. The dependency algorithm would then synthesize \( f \) and \( g \) as two separate functions and the realization of these functions would then be mapped to the structure shown in Figure 10.3. This scheme is easily extensible to include linear, multi-dimensional and systolic array structures.

Johnson [61] presents another approach to this problem. His approach, based on functional algebra, applies a series of behavior preserving transformations performing a system level factorizations resulting in a target expression closer to a physical realization by decomposing the original specification to isolate subsystems. Rath [93] has extended this work to treat the subsystems as interacting sequential components of a sequential system. Another approach to partition the control would be to adopt the work done by [61, 93] and extend it to FHDL synthesis paradigm.

10.2.3 Other Topics

This thesis motivates continued research into:

- recursion removal techniques,
- efficient frontier algorithms,
• multi-level circuit synthesis using frontiers algorithms,

• use of the frontiers approach for generating test vectors,

• possibilities of synthesizing timing models, and

• synthesis of multi-valued functions using frontiers.
References


165


166


Appendix A

VHDL Specification of Traffic Light Controller

entity TLC is
port(
  Cars : in BIT;
  TimeoutL : in BIT;
  TimeoutS : in BIT;
  StartTimer : out BIT;
  HiWay : out BIT_VECTOR(2 downto 0);
  FarmL : out BIT_VECTOR(2 downto 0);
  state : out BIT_VECTOR(2 downto 0) := "111"
);
end TLC;
architecture TLC of TLC is

begin

-----------------------------------------------
traffic:process

variable newstate, current_state : BIT_VECTOR(2 downto 0) := "111";
variable newHL, newFL : BIT_VECTOR(2 downto 0);
variable newST : BIT;

begin

current_state := newstate;

-- combinational logic to determine nextstate

case current_state is

when "000" =>
    newHL := "100"; newFL := "110";
    if (Cars = '1') and (TimeoutL = '1') then
        newstate := "100"; newST := '1';
    else
        newstate := "000"; newST := '0';
    end if;

when "100" =>
    newHL := "010"; newFL := "110";
    if (TimeoutS = '1') then
        newstate := "010"; newST := '1';
    else
        newstate := "110"; newST := '0';
    end if;

when "010" =>
    newHL := "110"; newFL := "100";
    if (Cars = '0') or (TimeoutL = '1') then
        newstate := "110"; newST := '1';
    else
        newstate := "010"; newST := '0';
    end if;

when "110" =>
    newHL := "110"; newFL := "010";
    if (TimeoutS = '1') then
        newstate := "110"; newST := '1';
    else
        newstate := "010"; newST := '0';
    end if;

end case;

end process traffic;

end TLC;
newstate := "000"; newST := '1';
else
  newstate := "110"; newST := '0';
end if;

when "111" =>
  newstate := "000";
  newHL := "000";
  newFL := "000";
  newST := '0';

when others =>

end case;

state <= newstate;
HiWay <= newHL;
FarmL <= newFL;
StartTimer <= newST;
wait for 10 ns;

end process traffic;

end TLC;