MULTILEVEL BEHAVIORAL VERIFICATION FOR VLSI DESIGN

by

Seung Ho Hwang

Memorandum No. UCB/ERL M89/85

18 July 1989

MULTILEVEL BEHAVIORAL VERIFICATION FOR VLSI DESIGN

by

Seung Ho Hwang

Memorandum No. UCB/ERL M89/85

18 July 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

MULTILEVEL BEHAVIORAL VERIFICATION FOR VLSI DESIGN

by

Seung Ho Hwang

Memorandum No. UCB/ERL M89/85

18 July 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Multilevel Behavioral Verification for VLSI Design

by

Seung Ho Hwang

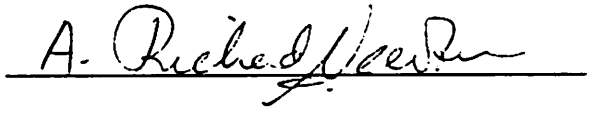
Abstract

To narrow the gap between the theory and its application to real problems, this dissertation addresses the practicality of hardware design verification. This goal is achieved by dividing the verification problem into two subproblems: correctness checking of finite-state machines and equivalence checking of multilevel behavioral descriptions.

An efficient algorithm is presented for the correctness checking of finite-state machines. The algorithm checks to see if an implementation, given by a net-list of gates and latches, satisfies a specification given as a state transition table. When the implementation is incorrect, an input sequence that distinguishes the implementation machine from the specification machine is provided to help the user locate errors. Experimental results show that the method can be applied to fairly large systems.

For the equivalence checking of behavioral descriptions, a formal technique based on theorem proving methods is used. Because verification is performed by a formal technique, complete verification can be achieved without exhaustive simulation. Behavior is specified in a hardware description language that deals with timing and functionality in one paradigm using functional (denotational) semantics. Type definition mechanisms and macros are provided, along with recursive definitions. The behavioral verification system automati-

cally handles type definitions and exploits hierarchy. Hierarchy is exploited when checking functional equivalence by using techniques such as inductive verification of recursive descriptions. Hierarchical timing verification is also supported by the abstraction of timing information by constraint propagation. During the abstraction of timing information, the availability of functional relations between signals eliminates the static-insensitizable-path problem.

Signature 

A. Richard Newton
Committee Chairman

ACKNOWLEDGEMENTS

I greatly acknowledge my research advisor, Professor Richard Newton, for his guidance, encouragement, and support throughout the course of this work. Without his enthusiasm and the inspiration he provided, this work would not have been possible.

I am also grateful to Professors Robert Brayton, Alberto Sangiovanni-Vincentelli and Carlo Sequin for their continuing interest in my research projects. Thanks are due to Professor James Sethian, who served on my qualifying examination from Mathematics Department.

I thank all CAD group members at the University of California, Berkeley. In particular, I wish to express appreciation to Gregg Whitcomb, Brian O'Krafka and Bill Lin for their critical reading of the manuscript of this dissertation. Rick Spickelmier, Tom Quarles, Peter Moore, Jeffrey Burns and Resve Saleh are some of the Berkeley CAD group members who were more than happy to help me whenever I needed it. I enjoyed the discussions with Tom Laidig, Srinivas Devadas and Tony Ma on relevant subjects.

I thank all former and current Korean EECS graduate students who I have been acquainted with for those wonderful time I had. Young Kim, Hyun Shin, Hyunchul Shin, Hong-June Park, Han Koh, Daebum Lee are a few of them.

My wife, Young Soon, my children, Jae-Won and Jae-Sun, have been the source of encouragement. I am also greatly indebted to my other family members and relatives for their continuous support.

I express my gratitude for the financial support that made this research possible. I acknowledge support from the Semiconductor Research Corporation and the Digital Equipment Corporation.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Motivation

The ultimate goal of computer-aided design (CAD) of integrated circuits(IC's) is to build systems which can generate automatically designs of entire circuits from the user-supplied requirements. However, the achievement of this goal is not foreseeable in the near future. Current approaches involve human intervention, and design verification continues to be an important problem.

Traditionally, simulation at various levels of description has been used for the verification of hardware design. In order to achieve a complete verification using this approach, exhaustive simulation must be performed. Unfortunately, excluding very simple designs, exhaustive simulation is nearly impossible. To overcome this limitation, formal verification techniques have been proposed. However, in order to apply these techniques to real design problems, more research work is needed.

In this dissertation the practicality of the hardware design verification is addressed in an effort to narrow the gap between the theory and its application to real problems. To achieve the goal of obtaining a more practical verification system, the verification problem is divided into two subproblems: correctness checking of finite-state machines and equivalence checking of multilevel behavioral descriptions.

1.2. Hierarchical Design Methodology

To address the complexity of a very-large-scale-integration (VLSI) design, the concept of hierarchical design has been widely accepted. This idea is similar to the structured programming technique, which has been used by software engineers for many years. A hierarchical methodology allows the designer to specify a design in a modular, multilevel manner, starting from the top level and working to the bottom. In VLSI design, this process involves a decomposition by which a complex entity is partitioned into multiple, less complex subentities. The decomposition of the design into modules is arbitrary, but typically it is organized according to the functional structure of the system under design. The top level of the design is a high-level description of the entire design in terms of modules and the interconnection of those modules. The modules are abstracted to hide unnecessary details from the designer and to allow him to think about only necessary information on a given level.

When a design is performed in a hierarchical environment, the verification of a large system can be split into smaller and simpler problems. This "divide-and-conquer" approach provides a practical approach to solving the complex verification problem. The increasing importance of the logical structure in VLSI design [1, 2], demands methodologies and tools to support hierarchical decomposition at high levels.

1.3. Timing and Functional Verification

In an equivalence checker for multilevel behavior, not only must the functional behavior be verified but it is also necessary to address timing behavior.

Most timing verifiers deal with flattened-down descriptions of a design at a specific level, usually in switch-level or gate-level [3, 4, 5]. However, with hierarchical descriptions it is more desirable to be able to handle the timing information in a hierarchical manner. The

abstraction of timing behavior, as well as functionality, is important. Also, as the timing verification is performed with the known functional relations of signals, the static-insensitizable-path problem, which has been an important problem in timing verification, can be eliminated.

In this work, the functional verification problem is formulated as an equivalence check between two descriptions. The description language and its verifier are presented. The verifier exploits hierarchy and structural regularity, both of which are very important to achieve the goal of practical formal verification.

1.4. Organization of the Dissertation

In Chapter 2, various aspects of design verification are introduced and the relationship between simulation and formal verification is explained. This is followed by a description of the concepts and terms needed in later chapters along with a definition of the verification problem addressed in this work.

Chapter 3 deals with the verification problem of finite-state machines. In this case, the problem is to check whether the implementation satisfies the given specification. The finite-state machine verification problem is defined and an efficient algorithm is presented. Experimental results are also presented.

Previous and related work is reviewed in Chapter 4. Because the study of formal approaches began earlier in the software domain than in hardware design, the most common techniques used in program specification and verification are described first, then existing formal hardware verification techniques are described. Finally, the problems and issues of these existing methods are addressed.

The behavioral description language used in this research is introduced in Chapter 5. The semantics of the language is based on a functional formalism. The primitive constructs are introduced with their semantics. In the language for data abstraction, three kinds of type constructing mechanisms are provided and parameterized designs may be defined as recursive cells. Also, some macro facilities are provided for the convenience of describing hardware.

The timing verification part of the behavioral verifier is presented in Chapter 6. After the timing constraints of synchronous digital systems are reviewed, the static-insensitizable-path problem is addressed. The timing model and the mechanism used for abstracting timing information are described. To illustrate the elimination of the static-insensitizable-path problem, some experimental results are provided.

Chapter 7 deals with the functional verification aspect of the behavioral verifier. Theorem provers utilized in existing hardware verification systems are reviewed and the theorem prover employed in the verification system implemented as part of this work is explained. Also, to illustrate the performance of the verification system, several verification examples are provided.

Conclusions and future research work in the formal verification field are included in Chapter 8.

CHAPTER 2

THE HARDWARE DESIGN VERIFICATION PROBLEM

2.1. Introduction

When a hardware or software system is designed there is always the problem of determining whether the design behaves as the designer intended. In a software system design, the final product is a program or a package of programs and the reproduction of the final result is usually very easy and inexpensive. However, in hardware design, the final product is not as easy to implement and in general it is very expensive to build the first version. Therefore it is necessary to check the correctness of the design prior to fabrication so as to prevent costly errors and corrections at a later prototype debugging phase. This activity is called *design verification*, *design test* or *design correctness checking* and should be distinguished from *product test*. In a product test, the goal is to see if there has been any physical failure during the fabrication process (which should not occur if the fabrication process is perfect). However, since the manufacturing cycle is not perfect, each of the products should be tested before being brought to market. The number of manufactured devices is generally very large and the evaluation cost of each device is critical for the total cost of the product. Hence it is important to devise a set of test inputs which detect most of the probable errors incurred during fabrication, yet is small enough so that manufacturing costs are within a reasonable limit. On the other hand, the goal of a design test is to check the correctness of the design so that if the manufacturing process was perfect, the final product would be fault free; that is, it is also what the designer intended. The focus of this dissertation is the design test problem.

Early integrated circuit design was performed manually. Since the circuits involved were relatively small and simple, this approach was quite satisfactory. The first digital integrated circuits (IC's) were available commercially in the early 1960's; it was a number of years before computer-aids were applied to the design and verification of these circuits. In retrospect, it is surprising how little the computer has been used in the design of IC's. Early circuits were small enough that mask patterns could be drawn by hand, and then photographically reduced to generate the IC masks directly. For the verification of the function of the circuit, however, simulators proved very useful. Initial work in the mid-1960's thus focussed on the development of device analysis [6,7] and circuit analysis [8,9] techniques. The circuit simulators were originally developed for the analysis of nonlinear and radiation effects in discrete circuits. It was not until the early 1970's that circuit simulators suitable for IC analysis became generally available [10,11,12,13,14,15]. As the digital hardware continued to become integrated into monolithic form and with increasing complexity, industry turned to the computer to store IC layout data and produce the masks required for manufacture. Systems for layout digitization and interactive correction found extensive use by the early 1970's. However, it was not until the mid-1970's that programs for the physical layout rule checking of the circuit began to find widespread use [16,17]. By 1975 it had become clear that computer-aids were a necessity in the design of complex IC's, both for physical and for functional design and verification. Until then, the layout of an IC and its transistor-level schematic diagram had been quite separate. In the late 1970's, computer programs became available for tasks such as connectivity verification [18], extraction of transistor-level schematics from IC artwork data [19,20,21] and even extraction of gate-level schematics from the transistor list [22]. These programs are loosely coupled in general and are often incompatible with one another. It was not until the mid-1980's that much effort was invested in making the computer-aided design (CAD) systems integrated.

In this chapter, various aspects of design verification are introduced, and the relation between simulation and formal verification is explained. Finally, the concepts and terms needed in the sequel of the dissertation are described, along with a definition of the verification problem.

2.2. Various Aspects of Design Verification

Throughout the process of designing VLSI circuits, a variety of different representations or *views* of the design are used. These representations may reflect a particular level of abstraction, such as a functional specification or mask layout, or they may reflect the view required for a certain application, such as the information required for simulation or formal verification. The choice of appropriate representations for each level of the design process is a key factor in determining the effectiveness of computer aids since it is through these representations that both the structure of the design and specific information relating to a particular design level are expressed. The design process involves transformations between these representations, both for design and verification. In this section, a classification of representations is presented. This classification is used in the later part of this section to relate various aspects of design verification.

The major categories of design representations are shown in Figure 2.1 [23]. These representations fall into one of three major categories: behavioral, schematic, and physical. At the behavioral or algorithmic level, the functional intent of a design is described independent of a particular implementation. In some cases, programming languages such as concurrent Pascal [24] have been used to represent the design at this level, as well as to provide simulation capability. Languages specifically designed for this task, called *hardware description languages* (HDL's), have also been developed [25, 26, 27, 28].

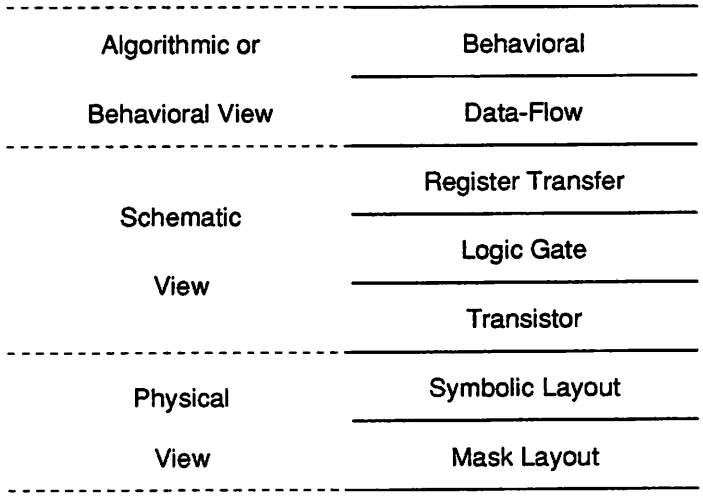


Figure 2.1 Categories of design representation

Once a functional implementation strategy has been determined, a schematic view may be generated. At its most abstract level, this schematic view consists of a *chip plan*, illustrating the loose physical placement of the major components and busses, or a register transfer level (RTL) description, defining the functional relationships between the major components of the design. As the implementation is further refined, logic gate level and finally transistor level schematics may be generated. With each new level of refinement more information concerned with the detailed physical and functional implementation of the circuit is included in the description. The final transformation consists of the generation of detailed, mask-level geometries, from a transistor-level schematic view.

The transition between high-level functional schematic descriptions and lower-level schematic and mask layout may involve the use of additional views. At the higher level, a data-flow description of the circuit may serve this purpose. At the behavioral level, this description may be viewed as the parse tree generated by a compiler operating on the algorithmic description of the intended function. At the RTL level, nodes in the data-flow graph represent an initial configuration of circuit building blocks used to implement the function, while branches indicate data paths between these functions. At the physical level, the symbolic layout forms a bridge between a schematic view of the circuit and its mask-level layout. A symbolic layout contains both explicit connectivity information and the relative placement of circuit components, such as transistors, cells, and building blocks.

A typical, top-down design flow of a contemporary design system is illustrated in Figure 2.2. First, from the system specification the designer chooses an architecture. The architectural design is then refined to the logic and circuit level. Finally, a layout is generated from the schematics of logic and transistors. An ideal approach is to develop the design from the specification by a methodology that ensures that it cannot be incorrect. This approach serves the ultimate goal of CAD researchers, but requires codification of a great deal of knowledge about the design domain, from the most abstract levels of system description down to the most detailed levels of implementation. It potentially faces a astronomically large search space of design alternatives. In software design, the attempt to achieve *correctness-by-construction* is exemplified in the principles of a structured programming methodology [29] and in research into automatic or semi-automatic programming [30]. In hardware design, research has focussed on pieces of the problem that are the most tedious for a human designer and therefore prone to human error, such as wire routing or programmable logic array generation. There have been attempts at automatic design of entire integrated cir-

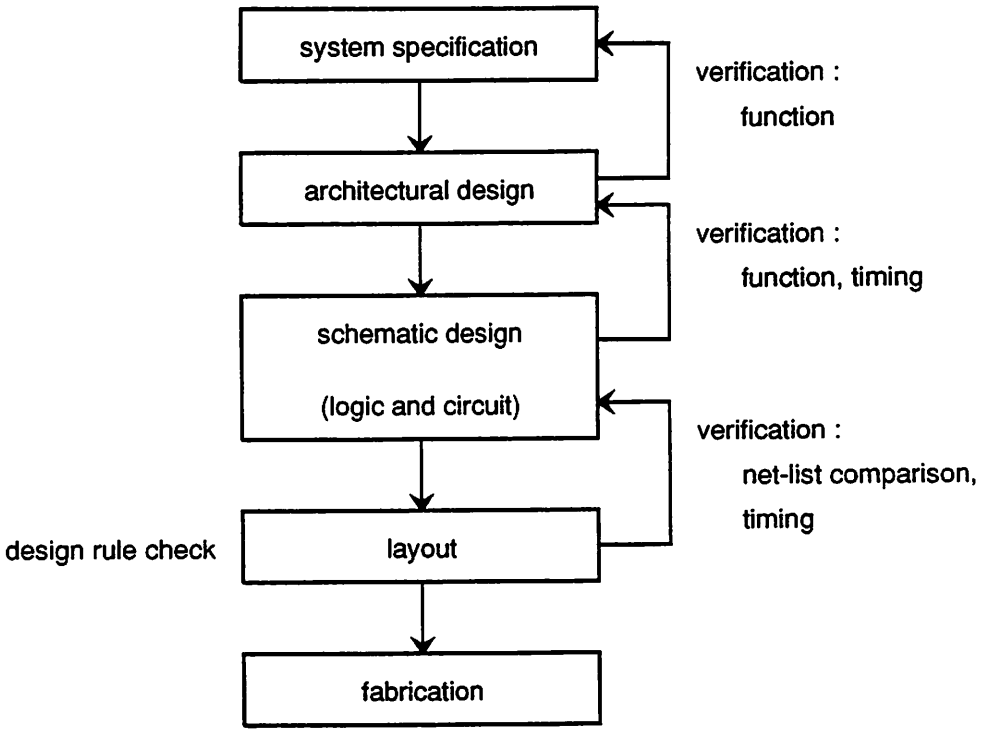


Figure 2.2 A typical VLSI design flow

cuits, including microprocessors that have met with varying degrees of success [32,33]. However, these attempts are still in their infancy and truly general-purpose design synthesis systems will take many years to perfect. Until all the design processes are performed by automatic and error-free procedures, the verification of design work created by humans will continue to be an important problem. Different aspects of verification are required at different levels during the VLSI design process. The functionality is checked at the architectural level; functional and timing behavior are checked at the logic and circuit level; and

physical design rules and detailed timing behavior are checked at the layout level. Finally, as a functionality check, a net-list comparison can be performed.

The problems associated with "verification" can be classified into three categories: functional verification, timing verification, and physical verification. In functional verification, the goal is to show that a design correctly implements the specified function. The goal of timing verification is to show that a proposed design will not malfunction due to hazards, races, excessive delays, or other timing problems. Timing verification may not prove the functional correctness of a design, but it can show whether the design will operate within the timing specifications imposed on it. The problem in physical level verification is in showing that a proposed design conforms to a set of rules or restrictions. These rules may reflect physical limitations of the technology being used, or they may be intended to improve product reliability and maintainability.

2.3. Simulation and Formal Verification

Simulation at various levels has been the most common verification method. The various levels of simulation used for integrated circuit simulation, their use, and examples of simulators for particular levels [34, 35] are listed in Table 2.1. The designer can select a simulator at the proper level depending on the stage of the design process, or may choose a *mixed-level* simulator that combines capabilities at two or more levels.

Circuit simulators, such as SPICE2 [15] and ASTAP [14], have been used successfully for the design and the performance evaluation of integrated circuits, and provide very accurate output waveforms. However, due to long computer run times, it is almost prohibitive to use these simulators for the analysis of large integrated circuits. An extensive effort has been made to reduce the required CPU-time for large circuits, while maintaining the same

Level	Use	Simulator Examples
Behavioral	Algorithmic Verification	GASP, SIMULA, ISPS, ADLIB
RTL	Logic Verification	ISPS, ADLIB, SPLICE2
Logic	Logic Verification	LOGIS, ILOGS, SPLICE2
Switch	Logic Verification	MOSSIM, RSIM
Circuit	Performance Evaluation Logic Verification	SPICE2, ASTAP, SPLICE2
Device	Device Model Development	GEMINI, PISCES
Process	Process Development	SUPREM, SAMPLE

Table 2.1 Hierarchy of large integrated circuit simulation

waveform accuracy. Techniques include the use of relaxation methods, such as Iterated Timing Analysis (ITA) [35, 36, 37] and Waveform Relaxation (WR) [38, 39, 40]. Relaxation methods provide significant speed improvement with the same waveform accuracy as SPICE2 (assuming identical device models) and have guaranteed convergence and stability properties. However, relaxation-based electrical simulation is still much slower than logic simulation and the actual CPU-time required depends on the characteristics of the circuit under analysis.

Logic simulation is an area which has evolved over the last two decades. Since the logic circuits of interest were already large and computers were less powerful, even the early work was concerned with efficiency improvement techniques (e.g. [41, 42]). Since then, continuous progress has been made towards the simulation of very large circuits [43, 44, 45, 46, 47]. Both temporal and structural sparsity is exploited in these techniques. Due to the occurrence of new complex MOS transistor designs, which cannot easily be cast into standard logic gates, transistor level simulation [48, 49] has emerged as an alternative. Also, a new form of simulation technique has been developed which solves for the amount of time required for a network variable to make a particular change, rather than solving for the network variables at the given time point (as in conventional circuit simulation) [50].

Recently, to simulate the behavior of a system across several abstraction levels in one simulator, so-called *mixed-level* simulation techniques have been developed [51,52,53,24,54,55,56,57]. In a mixed system, models of more than one abstraction level are used, such as circuit analysis, simplified macromodels, and logic simulation. Numerous mixed-mode and/or multilevel systems include several simulation analysis levels, combining many of the techniques developed for each level.

Simulation has been used for hardware design verification for so long that for many people verification means simulation. It remains true that simulation is still the best known way of answering the question "is my high-level description actually what I want?" However, simulation has several limitations for verifying the correctness of a design process. As a system increases in size and complexity, simulating the system is very costly in both space and time. The second limitation is that selecting a sufficient set of test inputs to "cover" a sequential description is almost impossible. In all but the simplest systems, the space of possible inputs can be vast. For example, a simple multiplier that multiplies two 16-bit integers can require over four billion different inputs to be sure it simulates correctly, and a system that contains a single 32-bit register can potentially have over four billion different responses to each input, depending on previous input sequences. Clearly, we cannot hope to test a system on every possible input with every possible system state. Rather, a subset of these tests must be selected, from which we can extrapolate or otherwise conclude the correctness of the design. Fortunately, the number of test inputs needed grows linearly with the number of components in the system. Unfortunately, the task of finding them is known to be NP-complete [58]. Therefore, simulation alone cannot guarantee that *complete verification* has been achieved unless exhaustive simulation has been performed. Here the meaning of the term "complete verification" is that when a verification result is positive, it is guaranteed that

a design operates correctly with all possible tests in the input domain. Finally, even if exhaustive simulation were feasible, the interpretation of the simulation output is not easy. Because of these difficulties, there is a need for new verification techniques to supplement the traditional simulation-based approach to design verification.

An important new approach that has emerged in the last decade is *formal verification* [59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84]. In formal verification, both a design and its specification are described by a language in which semantics are based on mathematical rigor, and the verification is then performed using symbolic manipulations. This guarantees complete verification when the verification result is positive. When a formal description is used during the complete design process, numerous advantages can be seen. For example the description of the actual behavior of an existing VLSI circuit or specification of the intended behavior of a circuit to be designed is generally performed informally, usually by means of natural language sentences or timing diagrams. By formally specifying the required behavior of a design, it becomes possible to communicate these requirements unambiguously to the people who will implement the specification, while providing a precise statement of design requirements. The ability to describe behavior formally in a design language allows the correspondence between specified behavior and the behavior of the constructed design to be established. As the primary aim in any design exercise is to produce an implementation that satisfies the behavioral specification, the importance of behavioral description is evident.

The use of behavior in a design language has not been fully explored to date. The advent of hierarchical designs requires the rigorous description of both the structure and the behavior of design components at the various intermediate stages in the evolution of a complete design. The manipulation of the large amounts of descriptive data requires that some of

the well-established techniques found in programming languages be used to aid the design process. The use of behavioral description "languages" (whether they be graphical, textual, a combination of the two forms, or a data structure) as the medium in which a designer will work will increase steadily in the future, due to the ever-increasing complexity of VLSI designs.

Large designs are usually produced in segments, with some parts of the design being produced by hand and others being produced using design automation tools. An interface between the components must be specified; this can most naturally be done using a design language which supports the description of structural, geometrical and behavioral attributes. Input to design automation systems will require behavioral information if designers are to work at a higher and more abstract level of design than that of pure structure or geometry. Most design automation systems, such as automatic placement and routing programs, work in the structural or geometrical domains. However, abstract behavioral descriptions are being used as input by an emerging number of true silicon compilers [85, 86, 87].

Finally, the most significant reason for including behavioral description capabilities in a language is for automatic design verification. Verification techniques using simulation do not generally make use of behavioral description languages, as they restrict themselves to describing circuits at a single level of representation such as the circuit level of SPICE2 [84], or the switch level of MOSSIM [48] and RSIM [88]. Simulation then occurs monolithically over the complete device description after the design is complete, with the inherent problems of descriptive complexity and redesign effort when flaws are discovered. An integrated design and verification process allowing increasingly detailed descriptions of an evolving top-down design to be verified by simulation or verified by mathematical proof techniques, breaks both the design and verification tasks into manageable sub-tasks. This approach

requires a behavioral description language in which behavior is described at different levels of abstraction, from the interaction among functional blocks down to the behavior of primitive cells. This approach is explained in more detail in the following section.

2.4. Hierarchy and Three Classes of Hardware Modules

Hierarchical techniques are frequently applied in managing complex technical and organizational problems. It is the purpose of such hierarchical techniques to master complexity by decomposing a complex system into a hierarchy of sub-modules. For a proper appreciation of the merits of hierarchical techniques, it is essential to have an understanding of the nature of complexity. Hierarchy applied to VLSI design will not generally lead to a reduction of the number of components in the chip -- on the contrary, there are several reasons to expect the number of components to increase. It can, however, affect the number of *interactions* between components which must be taken into account (by abstracting the interfaces between components), as well as the number of configurations to be considered during design. These latter aspects are also a measure of system complexity and are known to be dominant in the design of large systems [89,90]. Hierarchy is designed to reduce the magnitude of these contributions to complexity by using appropriate decomposition techniques. The intuitive notion is straightforward. One decomposes a large problem into a number of smaller parts and while each of the parts can be expected to exhibit only limited complexity, it is the expectation that the integration of the parts will not lead to a significant increase in overall complexity. Hierarchical decomposition techniques have long been in use for IC design supported by CAD tools. Many of these were devised for obtaining a reduction of the amount of design data; they did not intend to provide nor did they achieve a reduction of design complexity as measured by CAD program run times. Apparently, hierarchical decom-

position alone is not sufficient for reducing system complexity. Upon developing an understanding of the nature of complexity, it follows that abstraction methods form an essential ingredient.

In a hierarchical approach, a hardware module consists of a behavioral specification part and a structural implementation part. A behavioral specification describes how a particular design responds to a given input, and a structural representation describes how components are interconnected. According to how a hardware module is composed, modules can be classified as *interior modules* and *leaf modules*. The leaf modules can be further classified into *primitive leaf modules* and *special leaf modules*. These modules are illustrated in Figure 2.3. In a primitive leaf module, the structural part does not exist, and therefore does not need to be verified. The special leaf module consists of a behavioral part and a structural part, but its structure does not use any other module and is given as a set of net-lists of elementary parts. The net-lists are obtained using an extraction program, or are given as the results of several stages of the synthesis process. In an interior module, the structural part uses some number of submodules, each of which is either a primitive leaf module or a special leaf module or another interior module. For example, when a full-adder is given as a top level module, the Boolean gates may become primitive leaf modules. Or when a microprocessor is given as a top level module, the control path of the module can be a special leaf module and the logic blocks for data path can be interior modules. For the verification of a partially designed system, a designer may declare proper modules as primitive leaf modules. Later, when the design is implemented by refining the specification of the primitive leaf module, the module can be verified as a top level interior module.

From the verification point of view, a special leaf module can be treated as an interior module whose structural part consists of primitive gates and latches. However, the

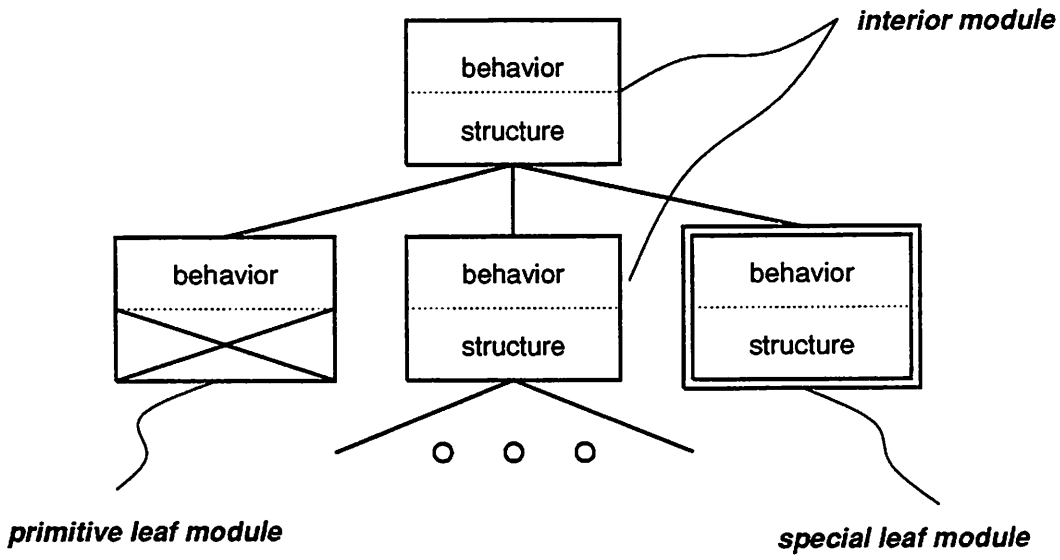


Figure 2.3 Three classes of hardware modules

verification approach for interior modules may be impractical for some leaf modules.

2.5. Verification Problem

In general, design verification problems can be classified into two categories. One category is correctness checking [65, 91], the other equivalence checking [74, 82]. In correctness checking, the verifier checks to see if the structural implementation implies the behavioral specification. Usually there is more than one way to implement a given behavioral specification and the implemented result may have more detail than the specification. Hence even if there were no errors involved during the design process, the

implementation may not be equivalent to the specification; however, the implementation may satisfy all the requirements of the specification. Thus a correctness checker verifies that the implementation implies the specification, i.e., the sufficiency is checked. On the other hand, an equivalence checker verifies that the two descriptions are functionally equivalent. In equivalence checking, two implementations are usually checked to see if they are equivalent. However, if the description language allows that specifications and implementations are described by the same language, the equivalence check between a specification and an implementation can also be performed. (Note that each of the correctness and equivalence checkers may verify functionality only or may verify the combined behavior of function and timing).

In this research, the objective is to develop a verification system with two verifiers: one is a practical correctness checker for special leaf modules of finite state machines and the other is an equivalence checker for interior modules. The correctness checker has been implemented using an implicit enumeration method while the equivalence checker has been implemented using formal verification techniques with the combined behavior of function and timing checked. In functional verification, the target level still uses a discrete Boolean form for signal values -- continuous circuit-level variables are not supported. In timing verification, synchronous digital design is assumed here.

CHAPTER 3

FINITE-STATE MACHINE VERIFICATION

3.1. Introduction

In hierarchical design, consistency between representations of a design at different levels of abstraction must be maintained throughout the design process if the designer is to be sure of a correct implementation. In this chapter, the correctness checking problem of a finite-state machine is considered. The verification problem of finite-state machines has been dealt with in a few different contexts [91,92,93,94,95]. In the temporal logic approach [91,92], the specification is described by temporal logic formulae and the implementation is represented as a state transition graph. The verification problem is determining if the state transition graph is a model of the temporal logic formula. In the symbolic approach [93], the verification problem involves deciding whether two logic-level sequential circuits with differing numbers of latches are functionally equivalent by using a symbolic comparison. However, due to the intractability of the problem, the formal approaches taken thus far have been restricted to medium-sized circuits with few memory elements (4 - 6 latches). In [94], a finite-state circuit model based on coordinating finite-state machines is proposed. In this approach, the circuit analysis task consists of determining whether or not the circuit model performs a given task by proving that a smaller derived finite-state system performs a derived task. Recently, an algorithm for verifying the equivalence of two sequential machines was presented [95]. In [95], a specification state transition graph is extracted from a register-transfer level description and an implementation state transition graph is extracted from net-

lists of gates and latches. Then a graph multiplication method is used to check the equivalence. From a practical point of view the approach in [95] is one of the most promising among the finite-state machine verifiers in terms of speed, and it will be used for comparison in the experimental section of this chapter. In [95], some practical-sized circuits could be verified with reasonable cpu-times. However, the verifier of [95] requires every Mealy machine to be converted to a Moore machine (which will usually have a much larger number of states and transition edges than the original Mealy machine). The verifier presented in this chapter outperforms the aforementioned approach by an order of magnitude in speed for Moore machines, and even better performance improvements are obtained for Mealy machines.

In the study of machine identification and fault detection experiments [96,97], it is necessary that a machine described by a state transition table must be strongly connected (i.e., for every pair of states, an input sequence must exist which transfers from one state to the other), must be minimal (i.e., contains no redundant states), and must have a *distinguishing sequence* (i.e., for every distinct state pair, an input sequence must exist which produces different output) [97]. However, the specification machine generally does not have all those properties and thus it is not always possible to devise a test sequence to verify the correctness of an implementation machine.

In this chapter, an efficient algorithm for the verification of finite-state machines is presented based on the concept of machine cover [96,97]. Definitions of terms and notation are given in Section 3.2, and the verification problem dealt with in this chapter is given in Section 3.3. In Section 3.4, the algorithms used to check the correctness of finite-state machines are described along with implementation details. In Section 3.5, the verifier is evaluated with some examples and the experimental results are discussed, conclusions are

given in the last section.

3.2. Definitions and Notation

In this chapter, standard terms and notation are used (e.g. [96,97,98]). However, a number of definitions are included here for completeness.

Definition 1 (cube and *minterm) :*

In an n-dimensional Boolean space, any set of vertices can be represented in sum-of-product form. Each product terms is called a *cube.* In a cube notation, each variable takes one of three values: 0, 1, or *. The *don't-care* (DC) value "*" means that a variable can take either of the values 0 (OFF) or 1 (ON). When a cube represents only one input combination of 1's and 0's, this special cube is called a *minterm.* In general, the number of minterms in a cube is given by 2^x, where x is the number of variables whose value is *.

Definition 2 (ON-set, OFF-set, DC-set, ON-set cover, and *OFF-set cover) :*

The *ON-set* (*OFF-set*) of a single-output logic function is defined as a set of minterms which evaluate the logic function to ON (OFF), and the *DC-set* is defined as a set of minterms for which the logic function can take either value of ON or OFF. The *ON-set cover* (*OFF-set cover*) of a logic function is a set of cubes that cover all the minterms of the ON-set (OFF-set). In general, the number of cube elements used in a ON-set cover (OFF-set cover) is not unique.

Definition 3 (completely specified and *incompletely specified function) :*

When the DC-set of a logic function is empty, the logic function is called *completely specified*, otherwise, the function is said to be *incompletely specified.* Note that when the logic is given by a net-list of primitive gates, as is the case in this chapter, each of the min-

terms evaluates to 1 or 0. That is, when the logic function is given by an implemented circuit, the logic function is completely specified.

Definition 4 (finite-state machine) :

A *finite-state machine* is a system that can be characterized by a quintuple

$$M = (\Sigma, S, Z, NSF, OF) \tag{2.1}$$

where
- Σ = finite nonempty set of input symbols
- S = finite nonempty set of states
- Z = finite nonempty set of output symbols
- NSF = next-state function, which maps $S \times \Sigma \rightarrow S$
- OF = output function, which maps $S \times \Sigma \rightarrow Z$

The above definition of finite-state machine is referred to as a Mealy machine [97]. The other type of finite-state machine is a Moore machine [97]. In a Moore machine the output function OF depends only on the state space S. Note that a Mealy machine is more general in the sense that any Moore machine can be converted into a Mealy machine without increasing the number of states and transition edges.

Definition 5 (incompletely specified machine) :

In equivalence checking, two machines are usually specified by structural descriptions, such as net-lists of gates and latches. In this case, the next-state function NSF is defined for all the states in the domain of S of Eqn. (1.1), and NSF is a completely specified function. However, in a hierarchical verification paradigm, the specification is given by a behavioral description -- in case of finite-state machine, the description is usually an equivalent form of a state transition table. And if a designer does not specify a next-state or output entry when it normally would be specified, it is usually because the machine is not expected to enter that next-state condition. Since the designer does not care what the next-state or output is, it could be specified as any valid next-state or output. In fact, it could be specified differently under

different machine conditions. Thus, it is reasonable to let an unspecified state transition table entry assume as many different values as desired. Such a machine is called an *incompletely specified machine*.

Definition 6 (applicable input sequence) :

Whenever a state transition is unspecified, the behavior of the machine may become unpredictable. In order to avoid such a situation it is assumed that the input sequences applied to the machine, when in any of its possible states, are such that no unspecified next state is encountered. Such an input sequence is said to be *applicable* to the state s of a machine.

Definition 7 (state cover and *machine cover) :*

A state q of a machine M_Q is said to *cover* a state s of another machine M_S if, and only if, every input sequence applicable to s is also applicable to q, and its application to both M_Q and M_S when they are initially in q and s, respectively, results in identical output sequences whenever the outputs of M_S are specified. The covering concept can be extended to machines as follows : a machine M_Q is said to cover a machine M_S if, and only if, for every state s in M_S, there is a corresponding state q in M_Q such that q covers s. Thus when a machine M_Q covers a machine M_S, for any state in M_S, there is a state in M_Q which can not be distinguished by input-output behavior.

3.3. Problem Formulation of Finite-State Machine Verification

The meaning of verification in this chapter is as follows: Given a specification and its implementation, the verifier checks the correctness of the design with respect to the specification. If the implementation satisfies the specification, the design is correct. Otherwise, the design is incorrect.

In this chapter, it is assumed that the specification is described by a state transition table and the implementation is given as a net-list of gates and latches. The specification machine will be denoted by M_S, and the implementation machine by M_Q. Also, a state of a specification machine will be denoted by s and a state of an implementation machine by q. The design verification problem is defined as a checking if implemented machine M_Q covers the specification machine M_S.

3.4. Verification Algorithm

The number of states of a finite-state machine grows exponentially with the number of latches in the implementation machine. However, for large machines the number of states actually visited, given the input sequences, is typically a small fraction of the total number of possible states. This is especially true if a state assignment program such as KISS [99] has been used in the synthesis process, where a minimum amount of combinational logic is the target and this may or may not produce a minimum bit encoding of the states. Since the encoding information is generally not available for verification purposes, to deal with general problems it must be assumed that the state encoding information is not available.

As previously mentioned, the number of latches is not minimal and the number of states that the implementation machine can take is much greater than that of the specification machine. There are 2^l possible states in a machine with l latches. From these states, a set of candidate states is selected. The procedure involved in selecting these candidates will be explained shortly. From the set of candidates, a state q_0 is arbitrarily chosen. A check is performed to see if the chosen state covers the initial state of the specification machine. This process is repeated until the correct initial state is found. If all the candidates fail to cover the initial state of the specification machine, it is concluded that the implementation is not

correct. This checking procedure is referred to in this chapter as *state generation and output checking* and it is described later in this section. The high-level pseudo-code of the main procedure is shown below. Of course, in the worst-case, this process has complexity of $O(2^l \times 2^m)$, where l is the number of latches and m is the maximum among the numbers of don't-care variables in the primary input segments of state transition edges of M_S. However, the selection and search procedures described later in this chapter result in much better performance. This is illustrated by the results obtained for real examples, as shown in Section 3.5.

```
main( )
{
  /* read in specification machine M_S */
  read_spec_machine( );

  /* read in implementation machine M_Q */
  read_impl_ckt( );

  /* levelize the circuit of M_Q and find transitive fan-ins
    for each output variable */
  levelize_ckt( );
  conify_ckt( );

  /* find all the candidate initial states of M_Q */
  Q_0 = enumerate_q0( );

  /* main loop */
  foreach ( q_0 in the set Q_0 ) {

    /* initialize */
    add s_0 to the set to be covered by q_0;

    if ( generate(s_0,q_0) fails ) {
      design is incorrect;

      /* prepare for the next candidate q_0 */
      re-initialize;
    }
    else
      design is correct, exit the main loop;
  }
```

}

In procedure read_spec_machine( ), the verifier builds up the state transition graph. The input format is shown in Figure 3.1. In the input format, the first field is an input segment of a transition edge, the second is a present state name, the third is the next state name, and the fourth is the output segment associated with the given inputs for the transition edge. The procedure read_impl_ckt( ) reads in the net-list of logic elements. The current implementation can deal with the primitive gates listed in Table 3.1. After reading in the implementation circuit, the verifier levelizes the circuit in topological order for the fast event-driven simulation used in

---

```
name traffic

# Highway and Farm road Traffic Light Controller
# (ref. Introduction to VLSI Systems, Mead and Conway, p.87)

#input    p-state    n-state    output
 0**        HG         HG        00010
 *0*        HG         HG        00010
 11*        HG         HY        10010
 **0        HY         HY        00110
 **1        HY         FG        10110
 10*        FG         FG        01000
 0**        FG         FY        11000
 *1*        FG         FY        11000
 **0        FY         FY        01001
 **1        FY         HG        11001

#end
```

Figure 3.1 Input format of a specification machine

---

gate name	description
AND	logical and
NAND	negation of and
OR	logical or
NOR	negation of or
INVERTER	logical not
XOR	logical exclusive or
XNOR (EQV)	logical equivalence (negation of xor)

Table 3.1 Primitive gates

later stages of the verification. Also, for each of the output variables, a set of input variables is found which affects the value of the output variable. The input variables thus obtained are called the transitive fan-ins of each of the output variables. The procedures of enumerate_q0() and generate( ) are explained in the following subsections.

3.4.1. Enumeration of Candidate Initial States

The goal of the enumeration process is to determine all the candidate initial states in the state space of the implementation machine. The procedure of the enumeration of candidate initial states begins with finding a set of output variables which have the same values for all the transitions from the starting state s_0 of the specification machine M_S. In the next step, for each of these output variables, enumerate from the input space the ON-set cover when the logic value of the output variable is 1 and OFF-set cover when the logic value is 0. The covering set is denoted as $cube\text{-}set_j$, $j = 1,..., k$, where k is the number of output variables which produce the same known output values. In the final step, take the intersection of the latch part of the cube-sets as follows:

$$cube\text{-}set_1 \cap cube\text{-}set_2 \cdots \cap cube\text{-}set_k$$

Then the initial state of the implementation machine should be a member of this set. Note that if none of these candidate initial states covers the initial state of the specification machine, it is concluded that the implementation is incorrect. The algorithm for enumeration of candidate initial states is as follows:

```
enumerate_q0( s0 )
{
  /* initialize the result */
  set_of_candidates = UNIVERSE;

  /* find output variables which produce the same value for all the
    transitions from the starting state of the specification machine MS */
  O = find_output_vars( s0 );

  /* enumerate the ON-set or OFF-set from the implementation
    machine MQ and take their intersection */
  foreach (output_variable in O ) {

      if (output_value = 0) {
        /* find OFF-set cover */
        S = podem(output_variable, 0);
        set_of_candidates = set_of_candidates ∩ (latch part of S );
      }

      else {
        /* find ON-set cover */
        S = podem(output_variable, 1);
        set_of_candidates = set_of_candidates ∩ (latch part of S );
      }
    }

  /* when there is no candidate state in the implementation
    machine, don't waste time */
  if (set_of_candidate = ϕ) {
      the implementation is inconsistent;
      exit( );
    }
  else
      return( set_of_candidates );
}
```

The worst case complexity of the above algorithm is $O(N_{PO} \times COMPLEXITY_{PODEM})$, where N_{PO} is the number of primary outputs and $COMPLEXITY_{PODEM}$ is the complexity of PODEM (path-oriented decision making) method [100]. The PODEM method is used to enumerate the set of cubes in the input space. In PODEM, given an objective, a signal and the desired value on the signal, a procedure called *back trace* traces a path from the signal backwards to a primary input to obtain a primary input assignment. The primary input assignment is then propagated to see if the desired value at the objective signal has been set up. If the objective signal has been properly set up, the procedure terminates. If an opposite value is set up, the procedure backtracks; that is, the previous primary input assignment is ripped up and the opposite value is assigned to that primary input. If the signal remains unspecified, the whole process is repeated. The above procedure continues until either a successful primary input assignment is found or all the primary input assignments have been exhausted. The process is an implicit enumeration algorithm in that all possible primary input patterns are implicitly, but exhaustively, examined.

To show the enumeration process, a simple example is given as follows:

Example : Consider the sequential circuit shown in Figure 3.2(a). It consists of four primary inputs, five primary outputs, and two latches. The partial state transition diagram is also shown in Figure 3.2(b). There are two outgoing edges from the initial state s_0. In the diagram only the output parts are shown for each transition edge. The output variables out1, out2, and out5 are consistent in their values for the two edges. Assume that the ON-set cover of out1, the OFF-set cover of out2, and the ON-set cover of out5 are given as follows :

$$
\begin{aligned}
cube\text{-}set_1 &= (1\ 1\ *\ 0\ \ 1\ *) \\
cube\text{-}set_2 &= (1\ 1\ 0\ 0\ \ 1\ *) \\
cube\text{-}set_5 &= (*\ 1\ 0\ 0\ \ 1\ 0)
\end{aligned}
$$

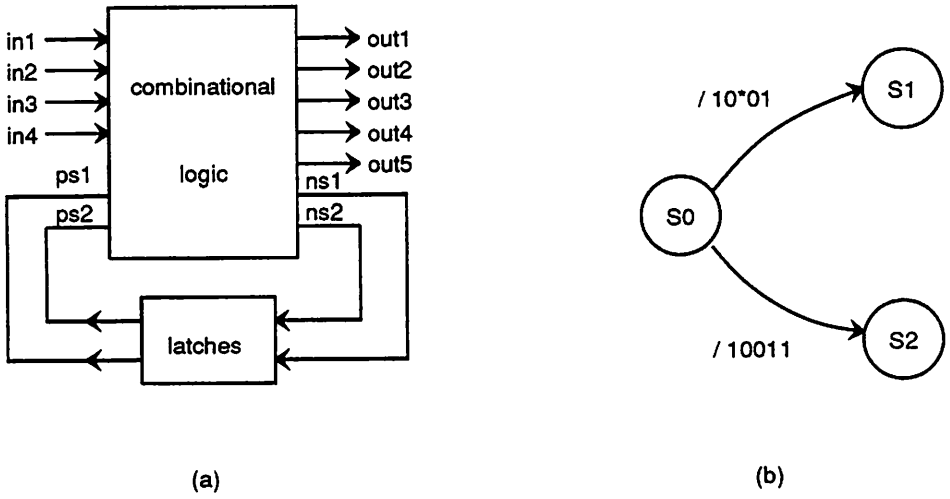


Figure 3.2 (a) a sequential circuit example
(b) state transition diagram

The intersection of the latch part is (1 0), and there is only one possible candidate initial state in this particular case. Generally, however, each of the cube-sets may consist of many cubes and the intersection of the latch part would consist of many cubes. For example, if (* 1 0 0 1 *) was obtained instead of (* 1 0 0 1 0) in $cube\text{-}set_5$, then the intersection would be (1 *), and there would be two candidate initial states, (1 0) and (1 1).

Note that if any don't-care values are among the variables of the latch part, all the possible minterms should be included in the set of candidate initial states. As will be demonstrated in the experimental results section, when an incorrect initial state is tried the verifier

notices the error so early in its checking process that the total cpu-time spent for wrong initial states is only a small fraction of total run time for most examples.

3.4.2. State Generation and Output Checking

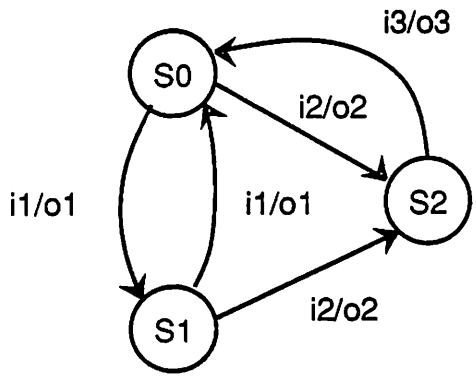
In this subsection the algorithm of checking if a given state q of the implementation machine covers a state s of the specification machine is described.

Since the implementation machine is given by a logic circuit, it is completely specified thus every input sequence is applicable to a state q. Thus to check if a state q in the implementation machine covers a state s of the specification machine, it is sufficient to check if every input sequence applicable to s when applied to both machines results in identical output sequences whenever the outputs of the specification machine are specified. The basic strategy employed in this work is as follows : for each applicable input at the given state s of M_S, obtain the next state and the output by simulation on the machine M_Q. Check if the output of M_Q implies the corresponding output of M_S. When the value of the output variable of the specification machine is don't-care, the value of the output of M_Q can have any value. This checking is referred to as output checking. If the output checking fails, the input becomes a component of the sequence of a counter-example. It is then concluded that the given state q does not cover the specification state s. If the output checking succeeds, see if the next state q_n has already been generated. If it is a newly-generated state, call the same procedure with the state pair (s_n, q_n) recursively, where s_n is the next state of s in M_S. If q_n was generated earlier, then check if q_n covers s_n by referring to the covering set of the state q_n. If s_n is in the covering set of q_n, there is no more work to do. If s_n is not in the list, repeat the above procedure with the state pair (s_n, q_n).

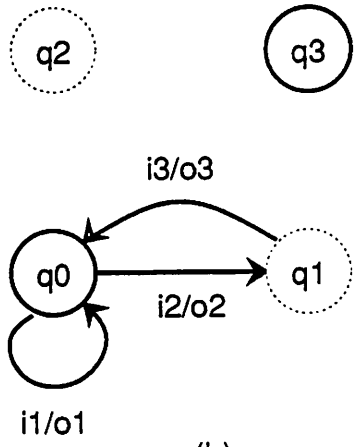
3.4.2.1. Example

Before the explanation of the detailed algorithm, a simple example is used to illustrate the state generation process.

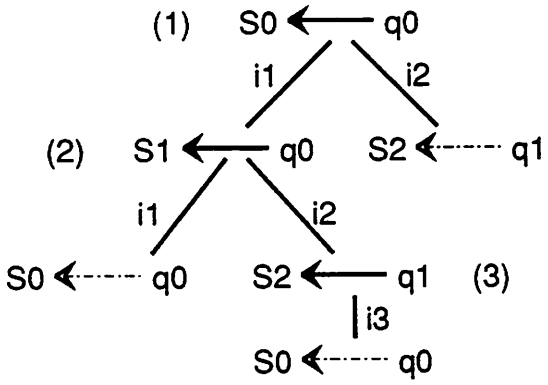
In Figure 3.3, the specification machine is given as an unoptimized machine and the implementation machine as an optimized one. The specification machine consists of three states with five edges. Assume that the implementation has two latches and among the four possible states, two states q_0 and q_3 are enumerated as candidate initial states as illustrated in Figure 3.3(b). The state q_0 is chosen arbitrarily, and a check is performed to see if state q_0 covers the state s_0. Figure 3.3(c) shows the recursive execution tree with each of the arguments, and Figure 3.3(d) shows the evolution of the covering list. First, the state s_0 is added to the covering list of q_0 (the stage of the evolution of the covering list is at (1) in Figure 3.3(d)). The outgoing edge with input $i1$ is simulated on the implementation machine. The next state of implementation is q_0, which has already been generated. The next state of the specification machine, s_1, is not in the covering list and is therefore added to the covering list of q_0 (at (2) in Figure 3.3(d)). Now a check is performed to see if state s_1 is covered by state q_0. When the edge with input $i1$ is tried, the next state pair (s_0, q_0) is obtained. Since the state s_0 is in the covering list of the state q_0, the check finishes with success (the dotted line means that the check is performed by referring to the covering list). The next edge with input $i2$ is tried. A new state q_1 is generated and the state s_2 is added to the covering list of the state q_1 (at (3) in Figure 3.3(d)). The only remaining edge from s_0 with input $i3$ leads to state pair (s_0, q_0) which is checked by examining the covering list. Finally, the only remaining edge from s_0 with input $i2$ is checked. Note that the total number of simulations used for this check is five, which is the same as the number of edges in the specification machine.



(a)



(b)



(c)

at (1)	q0 : S0
at (2)	q0 : S0, S1
at (3)	q0 : S0, S1
	q1 : S2

(d)

Figure 3.3 Unoptimized specification vs. optimized implementation example

(a) specification machine
(b) implementation machine
(c) execution tree
(d) evolution of covering list

3.4.2.2. Cube Simulation and Cube Splitting

Since the input can take on one of the three values (0, 1, and *), the term simulation here means *three-valued logic simulation* [101]. A *simulation pass* refers to simulation with one input vector. A simulation pass is said to be *complete* if the outputs of the simulated circuit (after simulation), are known (i.e. 0 or 1). Otherwise, it is *incomplete*. When the input vector is a minterm, the simulation pass is called *minterm simulation*, otherwise, it is called *cube simulation*. Minterm simulation is always complete because all the inputs are specified as 1's or 0's. Cube simulation may be either complete or incomplete; in the latter case, to make the simulation complete, unknowns in the input vector shall be assigned known values, 0 and 1 at different times. This assignment can be performed for one unknown at a time until the simulation becomes complete. Finding out if the simulation has become complete also requires a simulation pass and many passes may be required before the simulation eventually becomes complete. Because the input vector is a cube, the assignment process can be considered *cube splitting* -- every time an unknown is assigned certain values, the cube is split into two smaller cubes. So, instead of value assignment, cube splitting is considered to make the simulation complete. The cube splitting process can be depicted by a tree structure as shown in Figure 3.4. At the first level, the first unknown variable is split and this procedure is repeated similarly. Which unknown to select at each level is important because a good split order may prevent the tree from growing into its full magnitude ($O(2^m)$ complexity, where m is the number of unknowns). At each node, the next unknown to be split is selected using the heuristics explained in the next section. The deeper the eventual cube-split, the more simulation passes that are needed.

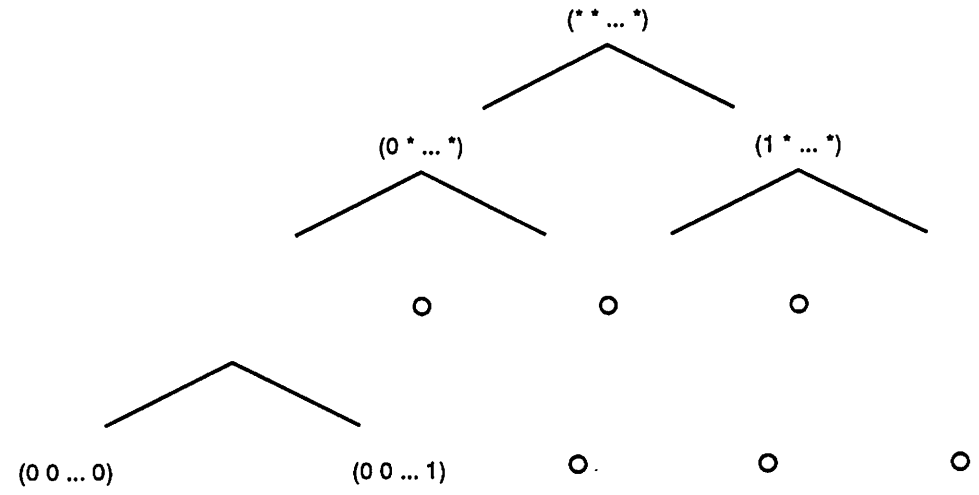


Figure 3.4 Tree structure of cube splitting

3.4.2.3. Cube Splitting Heuristics

Rather than choosing the unknown variables from the inputs in an arbitrary order (left-to-right in the example of Figure 3.4), input variables which have a high probability of resolving the unknown outputs are selected. Figure 3.5 illustrates the heuristics. First, for each of the unknown simulation output variables, find the set of primary input variables which may affect the output, then obtain the intersection and union of these sets. As mentioned in the algorithm of the main procedure, the implementation circuit is segmented as a cone circuit for each output variable [101]. The unknown input variables are then chosen from the intersection first, and from the union second, to split the cube. This process is

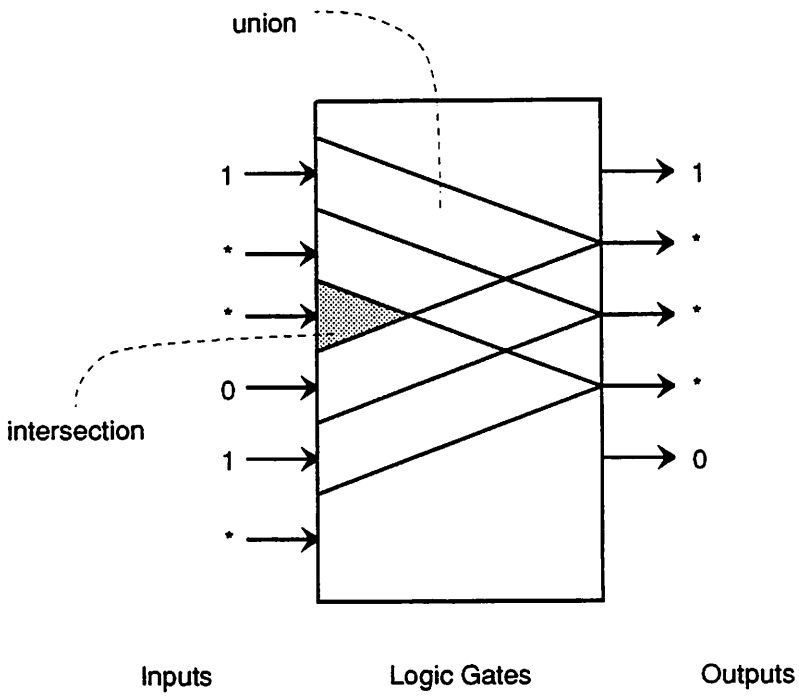


Figure 3.5 Ordering heuristics of split variables

repeated until all the output variables become known. This heuristic may avoid some useless cube splitting. For example, the last input variable in Figure 3.5 does not need to be considered in this case.

3.4.2.4. Output Checking

When the implementation circuit is simulated with a primary input vector and the present state vector, the present state vector contains no unknowns while the primary input

vector may have many unknowns. As explained in Section 4.2.2, both the next state outputs and primary outputs may have unknown values. In the case of next state variables, all the unknowns must be resolved by input space cube-splitting. However, in the case of primary output variables, the unknowns need only be resolved when the corresponding specification output variables are known. For example, with the specification output (1 0 * * 0 1), and simulated output (1 0 1 * * 1), only the fifth output variable must be resolved.

3.4.2.5. Detailed Algorithms

The procedure generate( ) and procedure check_state( ) return *success* if the given state of the implementation machine covers the state of the specification machine, otherwise they return *fail*. The only difference between the procedure generate( ) and check_state( ) is that when state q has been generated already, to check the covering, the procedure check_state( ) refers to the covering list first then checks each of the outgoing edges. If the state s is in the list, the state s is assumed to be covered by the state q. The actual covering check will be performed when all the previous calls of generate( ) and check_state( ) procedures return to the point where the state q was first generated. The procedure simulate( ) returns the information about completeness of cube simulations. The procedure performs the three-valued logic simulation using an event-driven technique [42] with topological-ordered [102] level information. If there are any unknown values in the next-state variables, procedure split_simulate( ) is called, this splits the cube by assigning values to unknown input variables using the heuristics explained above. The procedure split_simulate( ) calls itself recursively until all the next-state variables are resolved. The detailed algorithm of the state generation process is as follows:

generate(s ,q)

```
{
  foreach ( applicable input at s ) {
        /* get the next-state q_n of q and primary output by simulation */
        if ( simulate( ) is not complete )
          Q_n = split_simulate( );
        else
          Q_n = { q_n };
        foreach ( q_n in the set Q_n ) {
          if ( check_output( ) fails )
            return(fail);
          if ( q_n is newly generated) {
            add s_n (the next-state of s) to the set
              to be covered by q_n;
            if ( generate(s_n ,q_n ) fails )
              return(fail);
          } else if (check_state(s_n ,q_n ) fails)
            return(fail);
        }
  }
}

check_state(s ,q )
{
  if ( s is an element of the set to be covered by q )
        return(success);

  else
    add s to the set to be covered by q;

  foreach ( applicable input at s ) {
        /* get the next-state q_n of q and primary output by simulation */
        if ( simulate( ) is not complete )
          Q_n = split_simulate( );
        else
          Q_n = { q_n };
        foreach ( q_n in the set Q_n ) {
          if ( check_output( ) fails )
            return(fail);
          if ( q_n is newly generated) {
            add s_n (the next-state of s) to the set
              to be covered by q_n;
            if ( generate(s_n ,q_n ) fails )
              return(fail);
          } else if (check_state(s_n ,q_n ) fails)
            return(fail);
        }
  }
}
```

3.4.2.6. Remarks

When the *generate*( ) procedure succeeds with a (s_0,q_0) pair, the algorithm guarantees that for each reachable state s from s_0 in M_S, there exists a state q in M_Q which covers s. For every meaningful machine M_S, every state of M_S should be reachable from the initial state of M_S. When the specification state transition graph of the machine M_S is strongly connected, any state can be used as an initial state for verification purpose.

The total number of simulations is a measure of the algorithm's complexity. However, since the total number of simulations depends strongly on input parameters, a tight bound on the total number of simulations is very hard to obtain. First of all, the number of candidate initial states is unpredictable. As mentioned earlier, its worst case bound is simply $O(2^l)$, where l is the number of latches. For a given candidate initial state, the bound of the number of simulations is roughly given by $O(M_e \times 2^m)$, where M_e is the maximum among the numbers of transition edges from each of the states in M_S and m is the maximum among the numbers of don't-care variables in the primary input segments of state transition edges of M_S. The above bound is rough because sometimes the output checking requires some additional cube splittings.

The complexity of the verification problem of combinational logic is NP-complete and that of sequential logic is even worse. The algorithm presented in this chapter is basically a special case of branch-and-bound method. The branch-and-bound technique has scored several notable successes in practical computations. However, it is rarely possible to establish good bounds on its expected complexity. The basic premise of this work is that even for problems of worst-case exponential complexity, by exploiting special properties of the real problems that must be solved, the exponential case will never occur. Further, by exploiting these properties the expected-time complexity can be reduced substantially for real designs.

Even though the worst case bound is exponential, because the verifier developed in this work exploits the don't-care information available in the description of a specification machine, the expected total number of simulations required is almost linear with respect to the number of transition edges of the specification machine, as illustrated in the next section.

3.5. Experimental Results

The performance of the verifier was evaluated with three groups of examples. The six examples of the first set are obtained from finite-state machines developed for real chips. The second set of five examples is a series of resettable binary up/down counters obtained from running the program KISS [99]. The last set of examples is also a set of counters, obtained using the program MUSTANG [103], with an option that makes the program encode states randomly. The description of the examples are given in Table 3.2. The first and second columns show the number of states and state transition edges in the specification machines, respectively. The third column shows the number of primary inputs and primary outputs. The last two columns show the number of gates and latches of the implementation machines. The last example of the first group, the *cc* machine, was obtained from the SPUR [104] cache controller unit, designed at the University of California, Berkeley and is the largest single finite-state machine used in the SPUR system. In this example, the number of state transition edges is very large when compared to the number of states. This is due to the fact that in the state transition table, all the transition edges are expanded in the primary output space into minterms that can be grouped into cube notation. This fact leads to long cpu-time for the verification of this example and it should be thought of as a "worst case" situation. Each of the counters has two primary inputs: one for "reset" and the other for selecting up or down counting. The number of primary outputs of each example corresponds to the number of state bits. Note the very large numbers of gates in the third group of examples, due to the random state encoding.

Table 3.2. Description of three groups of examples

Example	Finite-State Machine			Logic Circuit	
	no. of states	no. of transition edges	no. of inputs/outputs	no. of gates	no. of latches
sse	16	56	7 / 7	130	6
cse	16	91	7 / 7	192	4
planet	48	115	7 / 19	606	6
sand	32	184	11 / 9	555	6
scf	121	165	27 / 54	959	8
cc	143	20,736	13 / 17	731	10
4bit_cnt	16	64	2 / 4	54	4
6bit_cnt	64	256	2 / 6	94	6
8bit_cnt	256	1,024	2 / 8	142	8
10bit_cnt	1,024	4,096	2 / 10	198	10
12bit_cnt	4,096	16,384	2 / 12	262	12
14bit_cnt	16,384	65,536	2 / 14	334	14
random_4	16	64	2 / 4	414	4
random_6	64	256	2 / 6	2,196	6
random_8	256	1,024	2 / 8	10,778	8
random_10	1,024	4,096	2 / 10	51,232	10

Table 3.3 summarizes the results of the enumeration and state generation process during verification. The first column shows the total number of enumerated candidate initial states and the second column shows the number of tried candidates before the implementation machine proved correct. In the *scf* example with eight latches, all the possible states are enumerated as candidates. In each example of counter, the state variables are the primary outputs, and only one state (reset state) is enumerated as a candidate initial state. This is also true for the examples of the third group, even though the initial states are quite differently encoded from those of the examples of the second group. The third column shows the number of states generated by the procedures generate( ) and check_state( ). The verifier

Table 3.3. Results of enumeration and state generation process

Example	enumeration of candidates		state generation	
	no. of candidate initial states	no. of tried candidates	no. of generated states	no. of unreachable states
sse	64	21	13	3
cse	15	5	16	0
planet	5	2	48	0
sand	6	5	32	0
scf	256	58	115	6
cc	360	65	143	0
4bit_cnt	1	1	16	0
6bit_cnt	1	1	64	0
8bit_cnt	1	1	256	0
10bit_cnt	1	1	1,024	0
12bit_cnt	1	1	4,096	0
14bit_cnt	1	1	16,384	0

reports any states which cannot be reached from the initial state of the specification machine, the number of such states is shown in the last column. The sum of the third and fourth column should be greater than or equal to the number of states in a specification machine. When the sum of the two numbers is equal to the number of states in a specification machine, as is the case with all the examples, it is concluded that the implementation machine does not have any redundant states that can be collapsed into a single state.

Table 3.4 shows the number of calls of the procedure simulate( ), memory size needed, and cpu-time used for the verification of each example. All the examples were run on a VAX[1] 8650 under the Ultrix[1] 2.0 operating system. The procedure simulate( ) is called not only while the states are generated but also while the outputs are checked. When the simulation results contain any unknown values in the primary output variables which does not have

[1] VAX and Ultrix are trademarks of Digital Equipment Corporation.

Table 3.4. Cpu-time comparison

Example	no. of calls of simulate( )	total memory (kbyte)	cpu-time (sec)			Graph Mult. Appr.
			finding Q_0	for wrong q_0's	total	total (sec)
sse	75	85	0.2	0.1	0.4	-
cse	95	99	0.2	0.1	0.3	-
planet	128	197	8.6	0.1	10.4	97
sand	266	193	2.9	0.1	5.0	-
scf	431	645	20.2	3.2	27.3	587
cc	197,373	13,330	1.5	283	357	-
4bit_cnt	64	67	0.1	0	0.2	-
6bit_cnt	256	103	0.1	0	0.6	-
8bit_cnt	1,024	233	0.1	0	3.1	-
10bit_cnt	4,096	725	0.1	0	19.8	-
12bit_cnt	16,384	2,659	0.1	0	101	-
14bit_cnt	65,536	10,357	0.1	0	571	-
random_4	64	139	2.0	0	2.8	-
random_6	256	535	66.0	0	77.1	-
random_8	1024	2,501	27.1min	0	30.4min	-
random_10	4,096	11,533	624min	0	681min	-

a corresponding specification output don't-care, they are resolved by cube splitting which requires simulation. If there were no don't-care values in the primary input segments of all the state transitions of a specification machine, all the simulations would be complete and the number of calls to simulate( ) would equal the number of state transition edges in the specification machine. The second column shows the memory size used. Most of the examples took less than one megabyte (except a few very large examples). The third column shows the cpu-time spent finding the set of candidate initial states. The cpu-time spent on wrong initial states is only about ten percent of the total run-time for all examples except *cc*. Note that in the third group of examples, since the states are encoded quite differently from the corresponding primary output, the output logic gate part is very complex and thus the

portion of the cpu-time spent for finding the initial state Q_0 for each of the examples dominates the total run-time. The cpu-time for finding the initial state of the "random_10" example is quite significant. The example, however, has more than 50,000 gates.

Since there are no standard benchmarks for verification, it is hard to compare the efficiency of our approach with other techniques. However, note that the verifier in [93] which is written in Franz Lisp and can deal with somewhat more general problems, took 31 minutes on a Pyramid 90x (approximately three times slower than the VAX 8650) to verify a 4-bit pre-settable binary up-down counter. The last column shows the cpu-time on a VAX 8650 spent by the verifier of [95] which used a graph multiplication approach. This other approach took more than ten times longer than the approach described here for some examples.

3.6. Conclusions

An efficient algorithm for the verification of the design correctness of finite-state machines has been presented. The concept of machine cover enables the verifier to check the sufficiency efficiently. The verifier checks to see if the implementation is correct with respect to its specification, which is given as a form of finite-state machine. This approach is suitable to hierarchical verification systems. The experimental results show that the verifier is fast enough to be used in real system designs.

CHAPTER 4

RELATED WORK ON FORMAL TECHNIQUES

4.1. Introduction

Formal techniques for specification and verification have been investigated earlier and more intensively in software design than in the hardware design. Due to the similarities between program design and VLSI system design, especially at the behavioral level [105, 106, 107], most formal hardware verification techniques have stemmed from program verification techniques.

The formal methods that will be described seek to do for programming what mathematics has done for engineering. That is, they provide symbolic methods by which the attributes of an object can be described and predicted. The objects in which this chapter is interested are computer programs or hardware descriptions, which are themselves strings of symbols. It should be possible to define transformations upon strings of symbols that constitute a program or a description, the result of which will enable us to predict how a given computer or design entity will behave. If this prediction is independent of specific values of input data of the program or the description, then it becomes a general statement about that program or description. If the general statement is formed so that it provides an argument that the program achieves its purpose, then it becomes the desired replacement for exhaustive testing and we call it a *formal proof of correctness.* When this approach is to be employed, the notion of the purpose of a program or a description must be made rigorous. This formalization becomes the *specification*, which serves to state precisely the requirements and objectives the

program or the hardware is to satisfy.

In this chapter, previous work on formal semantics is reviewed. Program specification and verification methods, and existing formal hardware verification techniques are also reviewed. Finally, the problems of previous formal hardware verification approaches are discussed.

4.2. Semantics for Programming Languages

In order to reason about any subject, a representation of its various elements is required. If the reasoning is to be carried out with mathematical rigor, the representation must be a formal model of the subject. Such a model must satisfy three requirements [108]:

1) It must be complete, representing all the essential aspects of the subject being modeled.

2) It must be predictive with conclusions drawn from the model corresponding to the results obtained by observing the subject itself.

3) It must be well formed. The model should not permit fallacious or ill-formed reasoning.

Three major methods have been developed for the definition of the semantics of programming language constructs [108, 109]: *operational*, *denotational*, and *axiomatic* approaches. In this section, each approach is reviewed and a comparison of the three is presented.

4.2.1. The Operational Approach

With the operational approach [110], the semantics of programming constructs of a programming language are defined in terms of a more primitive (lower-level) abstract machine, on the assumption that the state space and operational transformations defining the primitive abstract machine are so simple that their meaning or effect cannot possibly be misunderstood.

Specifically, the semantics of the abstract programming language constructs are defined in terms of the state space and operational transformations of the primitive abstract machine. Usually this is done for each programming construct by providing a "program" that translates the construct into a series of primitive transformations, so that for each programming construct of the abstract programming language there exists a "defining" program in the primitive abstract programming language. To determine the semantics of a programming construct, one must trace through its associated (defining) primitive program. Consequently, to determine the semantics of a program written in abstract programming language, one must trace through the "translated" program step-by-step to establish its precise meaning. For example, definitions of programming language constructs that use this approach are provided for PL/1 (by the Vienna definition method) [111] and Algol 68 [112].

To verify programs defined by the operational approach requires the execution of a trace through of the program written in the primitive programming language. The effect of program execution may then be determined by the individual transformations. Obviously, this method is applicable only to specific input values. Hence, a program is verified by observing the results of program executions and demonstrating that those results are in accord with the specification of expected results. The concept underlying this verification method, which maps one input state into one output state, is the common one of *program testing*. The specification of the particular set of input states and the corresponding set of output states constitutes the definition of the test cases -- that is, of the test data selection.

The operational approach characterizes the actual effect of program execution by relating it to executions of a separate, lower (more primitive) level. The approach does not solve the original problem of rigorously defining the semantics of programs and programming constructs, but merely pushes the problem to the lower level. More importantly, however, the

operational approach tends to define the semantics of a program only for specific computations of that program, rather than for the class of all computations that it can perform. In particular, its use to define semantics of a programming language forces us to consider all programs that could possibly be written in the language. Thus, instead of giving "functions" from which the semantics of any program written in the language can be derived, the operational approach tends to suggest implementations of the language.

4.2.2. The Denotational Approach

With the denotational approach, the semantics of programming constructs of an abstract programming language are defined by so-called *semantic valuation functions* [109]. Semantic valuation functions map programming constructs to values (numbers, truth values, functions, and so on) that they denote. These valuation functions are usually defined recursively: the value denoted by a construct is given in terms of the values of its constituent parts, and an emphasis on the values *denoted* by the constituent parts gives the approach its name.

Therefore two things are necessary in the denotational approach to semantics. First, a state space must be given and with the denotational approach this state space may include functions in addition to "normal" data objects. Second, a technique for defining semantic valuation functions must be given. The lambda-calculus [109, 113] may, for example, be used to model the concepts of function and functional abstraction, and conversion rules exist for syntactic transformations on lambda-expressions. Hence, if the class of functions representable by lambda-expressions are used to represent valuation functions, the lambda-calculus transformation may be used to manipulate these valuation functions.

The denotational approach to semantic definitions allows us to talk about construct and program *equality* in the sense that two constructs or programs are equal if they both denote

the same value in the selected value (state) space.

Verification based on this approach proceeds by constructing the valuation functions for the program constructs and then combining them and the valuation function representing the input conditions by (algebraic) transformation rules to arrive at valuation function for the program. Via further transformation-rule applications the valuation function for the program is mapped to the valuation function that represents the program final or output condition.

The denotational approach, in contrast to the operational approach, is independent of specific input values and so supports the definition of the semantics of the class of all computations that can be performed by a program written in the denotationally-defined abstract programming language.

4.2.3. The Axiomatic Approach

The idea of the axiomatic approach is to associate the semantics of programming language constructs with logical assertions of two kinds. The first, an input assertion, is assumed true prior to execution of a programming language construct and, from that assertion and the nature of the language construct (program), a second assertion, the output assertion, is derived that is true after execution of the construct. The pair of assertions thus characterize all legitimate input and output states of the construct and hence the effect (semantics) of the construct. Assertions are derived from the construct state-space and from the construct itself. This being the case, program verification based on this approach is independent of particular execution flow -- that is, of particular input-output pairs -- and proceeds in a given program by deriving output assertions from previously obtained input assertions for each construct, with the derivation guided by both the input assertion and the construct. The output assertion for one programming construct may be used as the input assertion to a subsequent program-

ming construct, so that program verification proceeds in an *inductive manner*. The process begins with an assertion about program input and concludes when an assertion about the program as a whole has been reached (derived).

4.2.4. Comparison of the Three Approaches

The relation of the three approaches is illustrated in Figure 4.1 [109, 108], where P denotes a program written in a programming language and S denotes the state space of the program.

In the operational approach, a program text P is translated into a parse tree $T_{(P)}$, which indicates the syntactic structure of the program. This process is denoted as (1) in the figure. Process (2) represents a translation of the parse tree into a program $MC_{(P)}$ in the machine

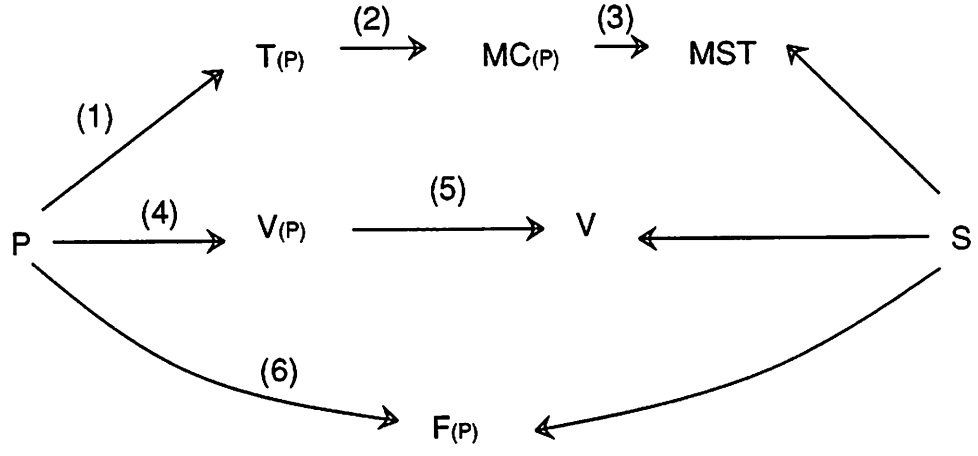


Figure 4.1 Comparison of the three approaches

code of some standard machine, process (3) represents the execution of the machine code program to produce the machine state transformation *MST*. Process (2) is defined by giving the translation rules for generating code from the tree, and (3) by specifying the operations of the machine. Different languages can also be described by giving their translation into the same machine code. So the combination of (1), (2) and (3) gives a standard implementation of the programming language. The processes (1) and (2) are usually performed by a "compiler" and hence the semantics are basically defined by a particular compiler. This may involve the details of an actual compiler. In hardware verification, the role of a compiler is played by a "simulator" and the semantics of a hardware description language is usually determined by its simulator. However, this approach is not suitable for formal verification.

In the denotational approach, an appropriate value space V must be defined onto which the state space S can be mapped. The program P is then transformed into a program $V_{(P)}$ in a programming language that allows an association between programming constructs and values in V to be established. Finally, semantic valuation functions are defined which associate the programming constructs in $V_{(P)}$, and $V_{(P)}$ itself, with the appropriate subspaces of the value space V (in case of λ-calculus, the subspaces of V correspond to members of the class of functions representable by λ-expressions). Hence, the semantics of the program $V_{(P)}$ are explicitly defined. The semantics of program P are indirectly defined by virtue of the conversion of program P to program $V_{(P)}$. The conversion must guarantee that different constructs are mapped into different value in the value domain V; this allows one to talk about *equality* in P: two expressions are equal if they both denote the same value in V. Note that in the operational semantics, the value of a program is defined in terms of what an abstract machine does with the complete program. Hence the local sub-components of the program may have global effects. Conversely, in denotational semantics, the value of a program is

defined in terms of the values of its sub-components, and it is easy to treat any particular part of the program.

In the axiomatic approach, a rule is associated with each statement of the programming language. These rules allow one to say what will be true after the statement has been executed and to relate it to what was true beforehand. The process is illustrated as (6) in Figure 4.1. From each of the blocks of the program P, a logical formula $F_{(P)}$ is obtained using the predefined rules. And the logical formula is the semantics of the program. This approach grew out of Floyd's work [114] on attaching assertions to the links of flowcharts. Its application to high level languages, mainly the work of Hoare [115], has made an important contribution to the art of proving the correctness of programs.

4.3. Program Specification Methods

A specification is the embodiment of the requirements a system is to satisfy [108] -- a precise, formal statement that expresses desired behavior in a manner intelligible to an implementor. The formal specification techniques discussed in this section maintain a semantic distinction between description, which is referred to here as the specification, and the described program, which is referred to here as the implementation. Loosely speaking, a specification describes behavior in terms of results, whereas an implementation defines behavior in terms of the procedure used to get the results. This distinction gives rise to the informal notion of describing "what" (specification) as opposed to "how" (implementation), and hence to the notion that specification languages are "nonprocedural" in nature.

There are three users of a specification [108], each distinct from the author of the specification: the validator, the implementor, and the verifier. The validator is the person who acts as the representative of the sponsor of the system. His or her concern is that it prop-

erly embodies the requirements. The implementor is concerned with producing a procedural, and ultimately executable, definition of the behavior described in the specification. The verifier is concerned with the correctness of the implementation, where correctness is defined with respect to the specification. The verifier's task is, therefore, to show the consistency between two representations of the same behavior, one substantially more detailed than the other.

In general, the validator seeks a specification language (technique) that enables arguments to be made about the properties of the described implementation, such that it will be secure or will exhibit other behavior consistent with the sponsor's goals. The implementor seeks a specification language that describes desired behavior in the least constraining way, so that the implementor has maximum freedom in producing that behavior from a program that must fit the constraints of a real computer. The verifier seeks a specification language that describes behavior in a manner easily mapped into one of the techniques used for formal verification. Not surprisingly, these three uses occasionally conflict, and no consensus on the desirable features of a specification language has yet emerged.

There presently exist three basic families of specification approaches: the algebraic, the state-machine, and the abstract model (also called the predicate transform method). These approaches define behavior in units called "functions," and in a result-oriented or "non-procedural" way that suppresses most details of implementation, including the step-by-step procedures that achieve the result. In the remainder of this section the three approaches are explained.

4.3.1. Algebraic Specifications

The initial theoretical work on algebraic specifications was done by Guttag [116]. From this work two specification languages emerged -- the AFFIRM language [117] and the OBJ language [118]. The underlying abstraction for algebraic specifications is the set of integers. Algebraic specification languages also assume "built-in" functions, typically *if-then-else* and the Boolean operators, which can be defined trivially using the technique.

Functions are defined in the algebraic technique by stating their relation to each other. They are functions in the mathematical sense because they may not have side effects and may only map a value in their domain to a value in their range. The technique is called "algebraic" because the values and functions of a specification can be viewed as forming an abstract algebra.

The *rewrite-rules* are an important aspect of abstract algebras. Rewrite rules are used to reduce expressions systematically in the given algebra. Each rewrite rule defines a transformation or "rewrite" that may be applied to an expression in the algebra. The rules are of the form

$$left \rightarrow right,$$

which defines a rewrite of any expression containing *left*: the replacement of the subexpression *left* by the subexpression *right*. In rewrite processes, two basic questions can be asked of any set of rewrite rules:

1) Will the reduction terminate? If yes, the so-called *finite termination property* holds.

2) Will the result be independent of the order in which the rules are applied?

Rewrite rules that produce results independent of their order of application exhibit *unique termination* or the *Church-Rosser property* [119]. A set of rules that is both finitely and

uniquely terminating is said to be *convergent*. The problem of determining whether an arbitrary set of rewrite rules is convergent is algorithmically undecidable.

The strong advantage of algebraic specification is its elegant mathematical simplicity; thus validation of a specification can be carried out mathematically. However, this technique presents disadvantages. Although algebraic specification is largely free from any representational or operational contents and consequently can avoid undue bias on the subsequent implementation, auxiliary or hidden functions are still necessary in specifying the behavior. As the number of auxiliary functions increase, more information about artifacts of the specification that are not directly related to description of the behavior are included. Another major disadvantage is that from the implementor's or validator's viewpoint, this technique shows difficulties in reading and understanding because functions are defined indirectly in terms of each other.

4.3.2. State-Machine Specifications

The idea of specification based on a state-machine was first developed by Parnas [120] and evolved significantly as reflected by the work of [121, 122]. The underlying abstractions of state-machine specifications are integers and Boolean objects. The existing languages also use real numbers and character strings. In addition, they make available to the specifier elementary extension mechanism such as vectors, sequences, and structures.

In this approach, an abstract data type is viewed as a state-machine; first, the abstract states are identified and then the behavior is defined by a set of functions to observe and change these states. Two classes of functions are used: *V-function* and *O-function*. A V-function is used to observe the state. It cannot define any aspect of state-transition. In contrast, an O-function defines state transitions by means of effect. The relationship between O-

functions and V-functions is very similar to that between variables and operations in a programming language. Informally, a V-function is analogous to an unbounded array with symbolic indices, one index corresponding to each argument position. Another way to think of V-functions is as mappings between names and values. The V-function name designates a multi-set of values, and the arguments' values for a specific invocation select a value from the multi-set.

The state-machine technique evolved from more pragmatic roots than did the algebraic approach, and perhaps for this reason the state-machine technique has not received as much attention in the literature. It has, however, been the technique most used in actual practice. This is due to the fact that state-machine specifications are more readable in practice and therefore reviewable for purposes of validation and for guiding implementors. Specification languages based on this technique include SPECIAL [121], and INA JO [123]. These languages have a rich syntax that permits the expression of very similar semantics in a range of alternative forms. This richness in turn means that the specifier's style is very lightly constrained by the language. Therefore the quality of a given specification is greatly dependent upon the skill of the specifier, to a much greater degree than the quality of a program depends upon the skill of the programmer (especially with languages like Ada, where the language syntax explicitly forbids stylistic pathologies). While such pathologies are very obvious in simple cases, they can be subtle in more complex specifications. In general, clean specifications can be written for functions whose behavior is easily expressed in terms of "set-oriented" name/value relationships. However, this method is less satisfactory for more complex and arbitrary structures and for expressing effects that involve the evaluation of algebraic formulas. Another inadequacy of state-machine specifications is that this technique basically lacks the requirement of abstract specification and violates the functional or nonpro-

cedural spirit of formal specifications, and may bias the subsequent implementation.

4.3.3. Abstract Model Specifications

The abstract model techniques, which is also sometimes called the predicate transform method, was developed by Hoare [115] as part of a unified technique for the specification and verification of abstract data types. In this technique, each function comprising an abstract data type is defined in the form of pre-conditions and post-conditions based on the underlying abstraction selected by the designer. Before specification, this underlying abstraction has to be defined formally so that the resulting specification can be reasoned formally. An abstract model specification, therefore, has no intrinsic meaning derived from a specific abstract model specification language; instead, its meaning depends upon the underlying abstraction selected. For example, when the *sequence* is selected as the underlying abstraction for the specification of a stack, the behavior of the stack is reasoned in terms of the sequence. Hence the usefulness of a given abstract model specification depends greatly upon the appropriateness of the selected underlying abstraction to the functions being specified. A clean and precise specification is possible when each function is easily defined in the form of pre- and post-conditions in terms of the underlying abstraction.

When the underlying abstraction consists of concrete programming objects, such as program variables, the specification closely directs its implementation. This idea was incorporated with the verification-oriented programming languages, Alaphard [124] and Euclid [125].

The state-machine technique and the abstract model technique share a number of similarities. In fact, in [122] it is observed that the state-machine technique is a variant of the abstract model technique, and in [108] it is shown that there is a straightforward transforma-

tion between these two specifications.

4.3.4. Comparison of the Three Techniques

Among the three approaches, the algebraic technique is the neatest and most abstract, expressing only the essential aspects of the item being specified and thereby coming closer to the mathematical concept of abstraction under which seemingly disparate entities may be described in a way that demonstrates their true similarity. However, when the ease of reading and writing a specification are considered, the abstract model and state-machine techniques are favored over the algebraic technique because of their close relationship with the correctness of the underlying abstraction.

In the behavioral specification, the state-machine and abstract model techniques retain the flavor of the operational semantics approach by defining semantics of functions constituting an abstract data type separately. In contrast, semantics of functions are defined through relationships among the functions in algebraic specification. Thus the algebraic technique deliberately avoids the operational flavor.

From the aspect of expressive power of a specification technique, which is a measure for clean and precise specification, each technique has advantages over the others in some respects while showing weaknesses in others. In conclusion, any one specification technique is not sufficiently versatile to satisfy all requirements. It is desirable to develop a unified approach incorporating different techniques into a single specification language without losing the advantages of each approach. In fact, work to study the relationships between these techniques has been carried out in [126] and a formal verification approach has been proposed [127] in which the algebraic and the operational approach are combined for a specification language.

4.4. Program Verification Methods

Program verification is the demonstration of the correctness of a computer program with respect to its specification. A common approach to the problem of program correctness is program testing -- the program is made to run on a sample of 'critical' input data. This method may increase confidence in the correctness of a program but it is far from a guarantee that the program is free from semantic errors. The notion of critical input data is simply much too vague. Another approach is program verification, which is the subject of this section. This approach avoids the deficiencies of program testing by proving mathematically that the meaning of the program satisfies its specifications. These specifications must, of course, be defined with mathematical precision (as described in the previous section). In this section, the basic concepts of three major verification methods are introduced.

4.4.1. The Inductive Assertion Method

This method, originated by Floyd [114], is one of the earliest verification techniques. The basic idea of inductive assertions is as follows. Assertions about the relationships among program variables are placed in the text of a program. These assertions in fact constitute the specification of a program. The assertions are generally expressed in the predicate logic. They consist of input, output, and intermediate assertions. Input and output assertions are located at the entry and the exit of a program. Intermediate assertions are located between statements of a program such that every loop is cut by at least one assertion. Each assertion claims that a stated relationship holds each time the program control reaches that assertion, i.e., assertion is an invariant.

In verifying a program, a formula called a verification condition is generated and proved to be a theorem for each simple path connecting two adjacent assertions on the pro-

gram. The validity of all the verification conditions for a program is sufficient to demonstrate the *partial correctness* of a program; for all inputs satisfying the input assertion, the output assertion is satisfied if the program terminates. The techniques to prove termination of a program for *total correctness* were developed by introducing the induction assertions that bound the number of loop executions. Since verification conditions are predicate calculus formulas, it follows that the deductive system used in an inductive assertion proof, consists of the axioms and inference rules of the predicate calculus. The validity of such a proof relies upon the *predicate transformer*, which links program semantics to a predicate calculus formula. A predicate transformer is a function mapping an assertion and a syntactic unit to another assertion. In the work of Floyd [114], the predicate transformer is a "strongest verifiable consequent" transformer. A strongest verifiable consequent,

$$svc(s_j, Q_i) = R_k,$$

is a function that, given a precondition Q_i for syntactic unit s_j and s_j itself, yields the "strongest" postcondition R_k for all outgoing paths from s_j. Here the meaning of "strongest" is that any postcondition obtainable from the precondition Q_i and the syntactic unit s_j is deducible from the condition R_k.

A difficulty inherent in the inductive assertion method is the relative independence of syntactic and semantic definitions. Programs are developed, initially, without formal regard to their ultimate verification. Formal semantic definitions are then tagged on during program interpretation. Failure to prove the subsequent verification conditions may be due either to a fault in the program or to the occurrence of an intermediate assertion that is not implied by the corresponding strongest verifiable consequent.

4.4.2. The Axiomatic Method of Hoare

Examining the work of Floyd [114], Hoare [115] introduced an alternative to his work which he presented in the form of the so-called Hoare calculus. Essentially this approach is identical to the inductive assertions method introduced in the previous subsection; however, it restricts the programming language to that without interleaved loops. Hence the inductive assertions method can be simplified. This simplified form leads naturally to the Hoare calculus. The Hoare calculus is a calculus for Hoare logic, in which one can formulate propositions about the partial correctness of "while-programs". Here the while-program means that there are no interleaved loops, and every loop can be expressed by structured while-programs. In this method, the semantic properties of syntactic units (blocks of a program) are viewed as theorems in a deductive system. A so-called *Hoare formula* is defined as follows:

$$\{p\}\ S\ \{q\}$$

where p and q are formulas of the underlying predicate logic (typically the first-order logic) and S is a block of a program. The formula, then, is to be proven by the usual techniques of applying inference rules to axioms and previously proven theorems to produce the desired conclusion.

The theorems that can be derived from Hoare's extended deductive system describe how the execution of a particular type of semantic of a syntactic unit modifies a given program state. However, it is required to associate different semantic properties with a single syntactic unit, as the meaning given to syntactic units is ultimately defined by the abstract machine upon which the syntactic units execute.

The axiomatic method does not provide a way to get stronger conclusions than the inductive assertions method. Its advantage lies in the more direct means it provides for expressing semantics. It will be applied only to the local properties of iterative constructs,

rather than (as with the inductive assertions method) to the program as a whole. Consequently, the axiomatic method is more suitable to modern control constructs than the inductive assertions method.

4.4.3. Verification Methods Based on Denotational Semantics

With the denotational approach to semantic definition, the semantics of syntactic units are defined by a *semantic valuation function*. An example of a denotational approach to the definition of the execution function of a small computer can be found in [128]; other approaches to denotational definitions of the execution function and verification using the lambda-calculus are discussed in [109]. Here the execution function $E(P, d_0)$ is defined by a binary relation, $R \leftarrow D \times D$, which defines the input-output behavior of the program P where D is the program state space. The relation R is defined by a semantic valuation function f formulated in an algebra of binary relations:

$$R = f(R_1, ..., R_m),$$

where R_i is an input-output relation of syntactic unit s_i which is a subcomponent of P. The semantic valuation function f is found by an algebraic method that consists of writing and solving a set of fixed-point equations.

To prove the correctness of a program, the desired semantics may be specified in terms of a function of symbolic values. The correctness of a program may then be shown by proving the equivalence between the symbolic values obtained from the program and the symbolic values denoted by specification.

4.5. Previous Formal Hardware Verification Techniques

Due to the similarity between program and hardware descriptions at the behavioral level, most formal hardware verification techniques have been based on the program verification techniques described in the previous sections. In this section several formal hardware verification techniques are reviewed.

4.5.1. Symbolic Simulation

To avoid the limitations of the simulation approach to verification, an enhanced simulation approach called *symbolic simulation* [60, 59, 66, 80] can be used. This is an offspring of conventional simulation, in the sense that it uses a model for hardware and a simulation engine, but it differs from the conventional simulation because it considers symbols rather then actual values for the circuit under consideration. In this way it is possible to simulate the response to entire classes of values with a notable improvement over the traditional technique. Symbolic simulation can be extended to verification of correctness because specifications and implementations may be run concurrently and the results manipulated and compared to establish a proof.

Darringer [60] addressed the application of the symbolic simulation technique to hardware verification by verifying a logic level description consisting of gates and flip-flops with respect to a specification in an RTL language such as ISP [129] and DDL [130].

In [59], the authors developed a language for symbolic simulation LSS, which was used to construct an abstract state machine.

Cory [66] describes designs at a certain level of abstraction as dual structural and behavioral descriptions in SDL [1] and Adlib [131].

Bryant [80, 132] developed a program that symbolically simulates the behavior of a MOS circuit represented as a switch-level network. During simulation the user can set an input to either 0, 1, or a Boolean variable. The simulator then computes the behavior of the circuit as a function of the past and present input variables. By using heuristic algorithms, the verification of a circuit by symbolic simulation can proceed much more quickly than by exhaustive logic simulation.

4.5.2. Predicate Logic

Since the difference between formal and non-formal verification lies in the presence of a mathematical proof, it is essential to have a well-founded formalism in order to represent hardware systems. Formal verification of hardware correctness often makes use of mathematical logic, thus it is worthwhile to consider it in more detail. In this subsection the basic concepts and terminology are introduced.

A formal deductive system [133, 134] is defined by the following items:

- The vocabulary of logical symbols and syntactic objects which is finite or may be enumerated.
- A set of formulae that can be generated according to specific rules. These formulae are called *well-formed formulae*.
- A finite set of *axiom schemes*, which are decidable subsets of the vocabulary; the elements of an axiom scheme are called *axioms*
- A finite set of *inference rules* which transform a well-formed formula into another one.

An *interpretation* of a well-formed formula in a formal system is an assignment of truth values to each of its atomic components. If the system includes both functions and predi-

cates, then the interpretation requires the assignment of functions and predicates to the function and predicate symbols. An *evaluation* of a well-formed formula is a function associating a truth value to it starting from its possible interpretations. A formula is *valid* if, and only if, its evaluation yields true for all its interpretations. A valid formula is often called a *tautology*. A formula is *satisfiable* if, and only if, there exists at least one interpretation for which it yields value true. A *deduction* in a formal system is a sequence of well-formed formulae $f_1, ..., f_n$ where each f_i is either an axiom or a formula obtained by applying an inference rule on another formula. A *theorem* t is a well-formed formula for which a deduction exists with t as the last well-formed formula in the deduction sequence.

In a formal deductive system, the problem of deciding whether a formula is a theorem or not is called the *decision problem.* If it is possible to find an algorithm performing such proofs, the formal system is *decidable* and the algorithm is called a *decision procedure.*

A formal deductive system is *complete* if and only if all valid formulae are theorems; that is, when all valid formulae can be derived. A formal deductive system is *sound* if and only if only valid formulae are theorems; that is when all derivable formulae are valid.

4.5.3. First-Order Logic Approaches

Wagner presented in [61] the first attempt to apply predicate calculus to the verification of hardware design using an available theorem prover for a number of simple proofs of unit-delay descriptions at the register transfer level. He developed a transition algebra and used a non-procedural RTL language modified from CDL [135]. The specification and the circuit are both represented in his language. The correctness is verified by proving or disproving the goal from the circuit description using axioms and definitions established from the language and transition algebra.

Hanes [64] developed a program that accepted functional and structural design descriptions in a higher-level language, translated them into predicate calculus clauses, and used a general-purpose theorem prover to establish design correctness. Hanes recognized the importance of hierarchical structure in designs as a means of reasoning about them, but did not fully exploit it.

Wojcik [65] demonstrated an approach similar to Wagner's for verifying logic level designs. In his method, both the specification and the behavior of the circuit are encoded into a set of axioms. The verification is then performed by checking the consistency of the axioms using a theorem prover [136].

The inductive assertion method developed for program verification was applied to the area of hardware verification. Pitchumani and Stabler [63] extended Floyd's inductive assertion method to formal verification of RTL hardware descriptions. They established axiomatic semantics of a simplified RTL language for synchronous design and demonstrated the verification condition generation from the hardware description with assertions. However, no mechanization of this approach has been yet reported.

Suzuki [137] explores a methodology which is halfway between simulation and formal verification. The tie to formal verification is represented by the specifications under the form of input/output assertions in first-order predicate logic. Instead of showing that output assertions are satisfied by the implementation for all inputs satisfying input assertions, he shows that this holds for selected inputs only. Such inputs are called "test data". In this method, both behavior and requirement specifications of hardware are described in Prolog [138, 139].

In the DDL Verifier [140], a verification system is applied to synchronous systems at the functional level. A translator reduces the circuit under consideration to cause/effect tables, i.e., to tables which show necessary and sufficient conditions for circuit operations.

The proof method uses backward reasoning and reduction to absurdity. In a later paper [91], the authors abandoned first-order predicates as a means of expressing specifications and resorted to *temporal logic*. (Verification approaches with temporal logic will be described shortly).

Hunt uses the Boyer-Moore approach [141] to describe and verify the FM8501 microprocessor [82]. The formalism is a quantifier-free first-order logic. Recursive functions are the primary means of description for hardware devices. Sequential devices operate through time, which is modeled as a stream of values.

4.5.4. Higher-Order Logic Approaches

Higher-order logic is an extension to first-order logic that allows variables to range over functions and predicates. Unrestricted higher-order logic suffers from a number of paradoxes, the most famous being Russel's paradox [142]. These can be avoided by resorting to type theory and a type hierarchy. Only propositional functions belonging to certain classes in the hierarchy are allowed. Higher-order logic generally encompasses the axioms of *infinity* and *choice*. The former states that the domain of individuals is infinite, the latter is used to introduce new primitive formulae.

Higher-order logic was originally developed as a foundation for mathematics [143]. Its use for hardware specification and verification was first advocated by Hanna in the VERITAS system [78]. The VERITAS system is supported by various software tools in charge of establishing and handling the theory database by a functional programming language, ad hoc parsers, user-defined inference rules, and goal-directed theorem provers. This approach was successfully applied to a simple example, demonstrating the correctness of a NOR gate.

The higher-order logic approach has been investigated intensively by Gordon [76,84,144]. In this approach the specification and the implementation are expressed directly in the logic, and hence no predicate transformer is needed for each syntactic unit. The approach to mechanizing logic in HOL system [84] is due to R. Milner [145] who developed the approach for a system called LCF designed for reasoning about higher-order recursively defined functions. The HOL system is implemented on top of LCF [145] which is implemented in Lisp environment. The language of LCF is called *ML* (the LCF *Meta-Language*). ML is an interactive programming language like Lisp. At top level, one can evaluate expressions and perform declarations. In HOL logic, there are five axioms and eight primitive inference rules. A proof in the HOL system is constructed by repeatedly applying inference rules to axioms or to previously proved theorems.

The higher-order logic is a very powerful formalism. However, due to the complexity of the language, an automatic proof is not easily obtained. The current implementation of the HOL system lacks automatic proof capability. Also, it is difficult to learn how to use the system.

4.5.5. Temporal Logic Approaches

Predicate logic is very powerful when reasoning about the properties of static situations, but it fails when dealing with dynamic phenomena. In the domain of hardware representation, it is necessary to cope with particular aspects of reality: timing and temporal evolutions. To satisfy the requirements of hardware, two choices seem possible: an explicit introduction of the time variable t into predicate logic, and a generalization of predicate logic to encompass the temporal domain.

Following the former approach, some authors introduce time functions, treating time just as one of many variables, with the usual rules for terms, formulas, and inference [146, 144]. Other authors augment standard logic to cover temporal evolutions, leading to *modal logic* [147, 148]. Predicates in first-order and higher-order logic stand for eternal verities. On the other hand, modal logic introduces the concepts of "possibility" and "necessity" in the future.

Modal logic, although more expressive than traditional predicate logic, still lacks the ability to cope with changes, an essential feature in hardware descriptions. To handle changes it is necessary to have a formal system that can reason from past events to what can or must be true at present and in the future. Such formal systems are generally grouped under the category of *temporal logic* [148].

In general, temporal logic assumes all usual connectives while adding some typical operators. Although there are many variants, the basic operators are: *henceforth*, *eventually*, *next*, and *until*. Temporal logic systems may be classified according to the way they consider time. With time generally modeled discretely, there are different ways of viewing the future. Past is always linear, while future may be either a unique world or a set of possible worlds. In the first case, time is linear in the future, too, and such logic is called *linear temporal logic*. In the latter case, time is branching in the future and such a system is called *branching time temporal logic*. In the former case, a system is supposed to have a unique evolution along time, whereas in the latter, a set of possible evolutions is considered.

In addition to linear and branching time, instant-based versus interval-based logic is another important distinction in temporal logic. A temporal logic is instant-based when propositions are asserted on single states only. A temporal logic is interval-based when propositions are asserted on sequences of states, i.e., on intervals in discrete time. *Interval temporal*

logic may be either global or local (global when truth of propositions is determined over an entire interval, i.e., on all its states, and local when truth is determined on the first state of an interval and holds unchanged on the rest).

Temporal logic was originally applied to the specification of the properties of concurrent programs [149, 150]. The properties of safety and liveness were the major interest. However, recently many researchers investigated applying the temporal logic to hardware verification. Research work in each direction of temporal logic for hardware verification is reviewed in the following.

4.5.5.1. Linear Time Temporal Logic

One of the first attempts to use temporal logic as a formalism to describe and reason about hardware is Bochman's work [67]. The temporal operators used in this work are as follows: "henceforth", "eventually", "next", and "while". The operators, "henceforth" and "eventually", include the present time in the assertions. The operator, "next", introduces the concept of discrete time and the concept of a transition that occurs between subsequent time instants. The "next" operator is used for the definition of the "while" operator. The author uses temporal logic to describe and verify properties of an arbiter, a device for regulating access to shared resources. The presentation reveals some tricky aspects in reasoning about such components. This approach was not mechanized and the verification was performed manually.

Bennet [151] presents an approach for proving correctness of asynchronous circuits in a deductive system for propositional temporal logic (PTL). In this approach, axioms describing the behavior of atomic elements (gates and modules) are formulated in PTL. A logic formula is composed from the conventional logical operators ("and" and "not") and a set of temporal

operators ("henceforth", "eventually", "next", and "until") excluding quantifiers. Desired properties of the circuit are formulated as specifications in PTL. As gates or modules are composed to form larger modules, their corresponding PTL formulas are combined using PTL theorems. Verified examples range from latches to a self-timed asynchronous pipeline.

Malachi and Owicki [152] applied the temporal logic to specifying a self-timed logic [153].

4.5.5.2. Branching Time Temporal Logic

In [69], an approach and the supporting tool for formal verification of synchronous and asynchronous control parts of digital hardware are presented. Implementations are described resorting to an ad hoc language called "state machine language" and specifications are expressed in the temporal domain using "computational tree logic" (CTL) which is a branching time propositional temporal logic. The tool supporting this approach is called the "model checker". In a later version, the tool is updated to deal with a more expressive branching time temporal logic with the "extended model checker" system [70]. This system takes a circuit described in terms of Boolean gates and Muller elements, and derives a state graph that summarizes all possible circuit executions resulting from any set of finite delays on the outputs of the components. The correct behavior of the circuit is expressed in CTL, a temporal logic. This specification is then checked against the state graph using a verification program.

In branching time temporal logic, asynchronous behavior can be described but as there is no global time grid, no quantitative delay value can be handled.

4.5.5.3. Interval Time Temporal Logic

In interval time temporal logic, an *interval* of time is defined as a nonempty, finite sequence of states and the temporal operator "next" is based on a globally fixed time interval. Based on this operator, any discrete-time signal waveforms can be described. Using the typical temporal operators "always", "sometimes", and "next", a number of temporal operators are defined based on the time interval, such as *temporal equality*, *stable*, *rising*, *falling*, *unit delay*, *transport delays*, and *temporal assignment*. Moszkowski presents in various papers [154,71] the interval temporal logic. The author presents as a logical consequence of his work *Tempura*, a logic programming language with imperative constructs for assignment. A multiplier and the Am2901 bit slice circuit are given as examples of its use in the hardware description domain [68].

In [155], the authors present *Tokio*, a concurrent logic programming language. Tokio is based on Prolog, but extends it by incorporating local interval temporal logic concepts.

4.5.6. Other Approaches

Many authors in the domain of hardware verification have created their own formal systems which may be reduced to first-order or higher-order logic. However, some of them can not be completely categorized into one of the previously reviewed techniques.

Milne [75] presents *CIRCAL*, a calculus to describe and analyze circuit behavior. Its use is not restricted to a specific domain -- it is both a very special HDL and a framework for device analysis by formal verification and constructive simulation. CIRCAL is supported by a Lisp environment, and is still under development; currently it supports expression manipulation and CIRCAL simulation, but not correctness proofs.

Before migrating to higher-order logic, Gordon produced the LCF-LSM [73] system. LCF is a verification oriented programming environment, which can be extended to the manipulation of specifications. LSM is a specification language for synchronous systems used at register transfer and gate levels. The author reports an application of the methodology to a simple computer [72].

Barrow [74] presented *Verify*, a verification system written in Prolog. He used a state-machine specification approach. Each hardware module is specified by output equations and next-state equations. From the structural information of a given module, the behavior of the implementation is composed, and a check is performed to see if the two behaviors are equivalent. When the system fails to perform an automatic proof, human intervention is required. In this work, there is no type defining mechanism and no inductive proof is provided. The system has been applied to verify the same simple computer presented by Gordon in [72].

Weise [81] presented a methodology and a tool, called *Silica Pithecus*, to prove functional correctness for designs at switch level.

Musser et al. [83] presented a method for specifying bidirectional hardware devices. The author points out that the predicate model can handle simple cases of bidirectionality, but breaks down in others, particularly when gate capacitance and/or charge-sharing are involved. He suggested a more elaborate model of hardware devices for formal verification that can handle gate capacitances and charge-sharing.

4.6. Conclusions

Study of previous work on formal hardware verification illustrates that the choice of a formalism involves a compromise between expressive power and the ease of automatic

synthesis/verification. In a simple and restricted formalism, it is hard to specify complex devices simply and concisely. Nonetheless, in a powerful formalism, it is difficult to automate the synthesis and verification.

The formalism of higher-order logic is very powerful and provides enough expressive power in some sense. However, due to the complexity of the higher-order language, the proof procedure needs human guidance. Currently, the higher-order logic may be viewed as an environment of constructing proofs rather than an automatic theorem prover.

The temporal logic is useful for specifying asynchronous behavior such as the liveness and safeness properties of communicating processes. It can also be applied to hardware design to specify the relations of signals. Even though there exist a few theorem provers for temporal logic [151, 79], the theorem provers for temporal logic are still in their infancy, and the lack of powerful automatic proof procedures prevents this approach from wide application to real design problems.

Most systems aim to provide a model or tool tailored to a specific level of abstraction. There is no panacea for all levels of abstraction involved in the design and implementation of a piece of hardware. Therefore the best formalism to use depends on the nature of the problem under investigation. As design work from the gate level to the physical mask level becomes more and more automated, computer-aided tools at the higher level becomes more and more important. In this research work, for the behavioral description language, a functional formalism is chosen. The appropriateness of this decision is explained in Section 5.3 of the next chapter.

For a hardware verifier to be used in real design problems, the efficiency of a verifier is one of the most important concerns. In VLSI design, the complexity of managing all the information is already beyond human capability. Hence the design usually proceeds in a

hierarchical manner. Also many parts of the VLSI hardware are designed in a regular structure and some of the hardware modules are repeatedly used. Thus a mechanism that can exploit the structural hierarchy and regularity is needed.

The importance of hierarchical structure in designs as a way to reason about them was recognized by Hanes [64], but in the work the hierarchy was not exploited heavily. The exploitation of hierarchy is well illustrated in the work of Barrow [74].

The exploitation of design regularity was illustrated in the work of German and Wang [77] by parameterization. The advantage of this approach is that a single proof can demonstrate the functional correctness of all instances of a design.

In the verifier developed in this work, the hierarchy and regularity is automatically exploited, as explained in the following chapters.

CHAPTER 5

BEHAVIORAL DESCRIPTION LANGUAGE

5.1. Introduction

In hardware design, the "real world" consists of transistors, integrated circuits, boards, etc.; anything expressed on paper to describe hardware is actually a *model* of the "real world". A great variety of models have been, and continue to be used for a variety of purposes. For the analysis and synthesis of any design, a model of each component is needed. Finally, it is through the use of models that the communications between designers, between machines and designers and between machines is performed. Some models, such as circuit schematics, mechanical drawings, printed circuit art work, layout mask art work, etc., are very explicit and offer all the detail necessary to fabricate hardware. They may reveal what the hardware looks like, but the organization, operation, and function of the hardware is difficult to determine from such models because of their volume and because they do not attempt to reveal such things clearly.

Shannon [156] first showed the advantages of the logic diagram and/or Boolean equation models for hardware description, which replace detailed fabrication and electrical information with abstract logic equations (a model) and hence make it easier to ascertain operation and function. However, they do not make it easy to do so for sizable digital systems, because these models also tend to be voluminous. They serve well at certain stages in the engineering and maintenance of digital systems by de-emphasizing fabrication detail while emphasizing information that is more important at those stages. These models are said to be more *abstract*

in that they are symbolic and mathematical.

Before the logic details of a digital system can be determined, designers require even more abstract models that present functional detail explicitly while not implying logic and fabrication detail. Block diagrams, flow charts, timing diagrams, English text, etc. have all been used as parts of suitable models. However, "register transfer" or "computer description" languages have been developed and accepted more widely as useful ways to represent models of suitable abstraction for simulation and design.

If the organization and operation of a sizable digital system is to be ascertained easily, then only the most relevant information must be presented and in a concise fashion. If fabrication is intended, then a description must be complete and precise enough so that logic detail can be determined directly by man or machine, a process known as *synthesis*. If machine synthesis or verification is desired, then a description should be suitable for machine manipulation.

Hardware description languages are designed to meet various goals. They can be used for documentation purpose at each stage of design, or can be used as design aids -- the manipulation of the large amounts of description data requires that the well-established techniques found in programming languages be used to aid the design process.

As explained in Chapter 2, a digital system can be described at several different abstraction levels. While many languages can be used at several levels, each language is especially convenient for a certain level. For example PMS (Processor, Memory, Switch) [157] and ISP (Instruction Set Processor) [129] are good for the system level and the instruction level, respectively. CDL (Computer Description Language) [135] is especially good for the microprogramming level. And DDL (Digital System Design Language) [130] is suitable for the register transfer level. The classical HDL's and their references are surveyed quite

extensively in [25].

Most of those HDL's, however, do not address and explore the use of behavioral aspects in a description language. The advent of hierarchical design requires the rigorous description of both the structure and the behavior of design components at the various intermediate stages in the evolution of a complete design. The manipulation of the large amounts of descriptive data require that the well-established techniques found in programming languages, as the medium in which a designer will work will increase steadily in the future due to the ever-increasing complexity of VLSI designs. Some of the modern languages, such as STRICT [158], muFP [159], ELLA [27], Helix [160], HSL-FX [161], and VHDL [28] have been developed with that purpose in mind. A useful design environment should facilitate both maintenance and future upgrading, which among other things means that the documentation must be accurate and easy to access and understand. Another related point is reusability of design efforts. It is important that efforts made during one design can also be of use in other designs. Modularization and regularization are features that improve design reusability [89,90].

Hardware description languages can be classified into three groups. The first group is characterized by its structural and single-level description. Most of the circuit-level, switch-level, and gate-level representation and some of the classical hardware description languages belong to this category. The description languages of this group are used mainly for design-data interchange and simulation purposes. Every simulation techniques requires some kind of behavioral information and in case of the languages of this group this is usually embedded in the simulation program code rather than in a separate description language. The second group is characterized by its efficiency for simulation. Since simulation has been the prevailing approach to design verification, every description language assumes its particular simulator

and most of them employ an event-driven simulator for efficiency. Most modern description languages, other than those provided for use with formal methods, belong to this group. However, for the formal approach to be utilized in verification process instead of simulation, a description language must have mathematical rigor in its semantic definition. The languages of the third group have emerged to fulfill such a requirement, as described below.

5.2. Behavior and Structure

The design of a large digital system is a complex task. Most designers have their own personal preferences about how the design work should be accomplished. Designers should be free to use whatever method or technique they prefer. The freedom offered by a design environment should include choice of abstraction level and implementation strategy. The environment should also support co-existence of specifications at different levels and also specifications made with different implementation strategies in mind [162].

The process of design is usually described as bottom-up, top-down or a combination of both. The term "meet-in-the-middle" design has been used recently to describe an evolving approach [163] -- some designers design at the bottom and design basic building-blocks, while others begin at the top and decompose the problem until it can be described in terms of the building blocks; the "middle". In the bottom-up design, the specification of a new structure is created by combining the specifications of pre-existing structures. In the top-down design a specification which contains a mixture of behavior and structure is progressively transformed to one which contains more structure -- almost invariably any pre-existing structure is unchanged. However, if some structure is found to be inappropriate, the design is iterated back to an earlier behavior and a new route to structure is explored. Thus the process of hardware design is essentially one of transforming a behavior into a structure [163]. In

practice, the behavior transformation process is guided, either by a human designer or by an algorithm in a synthesis program, towards structures which are thought to be a good implementation in some sense.

5.2.1. Structure

Modules of a hardware system can often be viewed as an interconnection of other modules. There is thus a need for two kinds of structuring capabilities [162], or as it is pointed out in [26]: machines are designed (ideally) from logically disjoint functional units and both hierarchical relationships and relationships of symmetry (e.g. arrays), that is, interconnections, are needed in order to describe interactions between units. Another aspect of hardware modules is that a large number of their submodules are of the same type. That is, they typically have a high regularity factor. This is particularly true of the lower-levels of design (e.g. the gate or transistor level). However, as systems become more complete, there is increasing re-use at ever-higher levels of abstraction. Type declarations which can be instantiated in a large number of places are thus a valuable aid. Typing mechanisms are convenient, since different instantiations of a module often have some small differences depending on their surrounding. It should also be possible to easily specify regular structures of modules and give special treatment to modules at their boundaries.

The relationship between a module and the submodules from which it is composed is a hierarchical relationship. Any of the modules composing a module can, in its turn, be composed of other modules. This hierarchy of modules defines the hierarchical structure as seen in Figure 5.1 The hierarchical structure thus conveys information about composition (and decomposition) of modules in terms of other modules.

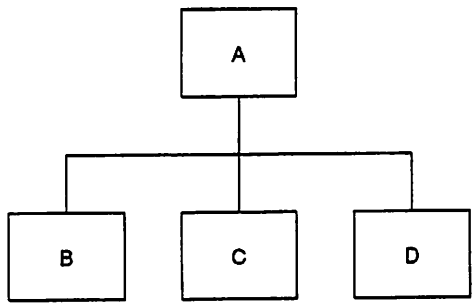


Figure 5.1 A hierarchical structure

5.2.2. Behavior

In any design approach, bottom-up, top-down, or a combined approach, a designer should be able to specify the behavior of any hardware module. In the case of top-down strategy, a behavior becomes a specification of the module and in the case of bottom-up strategy, a behavior becomes an abstraction describing the interface specification of the module to form a new module of an interconnected structure. The new specification of the interconnected module is usually an abstraction of its behavior, i.e., of the behavior of the module as seen from the outside. This can be said to be the *external behavior* and it should not be interpreted as specifying the mechanisms implementing the behavior, i.e., the *internal behavior*. The specification of internal behavior describes the inner workings of a module that gives the desired external behavior. Note that there can be many different specifications of internal behavior which give the same external behavior.

In a hierarchical design environment, it is natural to provide a mechanism to compose behavior from sub-behaviors and similarly for structure (modules made from sub-modules). When a designer approaches with a bottom-up strategy, it is clearly necessary that structure should be composable from behavior. This is a typical situation at the primitive level of design hierarchy, i.e., an AND-gate of an adder is generally described by its behavior. All modern HDL's provide users with these facilities. Less obviously, it is useful for behavior to be able to contain structure. This facility is needed in the combined strategy of top-down and bottom-up to incorporate pre-existing structure into behavior. If this is not allowed, either a behavior description has to be totally converted into structure (i.e. more top-down design) or, instead of incorporating the structure of the pre-existing component, a behavioral representation of the pre-existing component must be substituted. This means that when composing bottom-up, an equivalent pure behavior representation has to be grown in parallel with the structural description, which certainly means extra work to generate two representations. Furthermore, in practice, it is nearly impossible to match behavior and structure models of any complexity. Because of the different ways of configuring behavior and structure descriptions, some modern HDL's, such as VHDL, suffer from this inability to describe arbitrary compositions and decompositions of behavior and structure, while the ELLA language and the language developed in this dissertation work avoid the problem by defining a specification interface which is used *both* for structure and behavior. That is, use a form of description that does not distinguish syntactically between structure and behavior. Rather, the *intent* (structure or behavior) is simply inferred from the *use* of the description.

5.3. Functional Formalism

The question of which formalism to employ as a behavioral description language is fundamentally dependent upon the underlying abstract model of hardware behavior. If the underlying model is simulation, the behavioral description is likely to be oriented toward efficient simulation of the digital system being described, generally on a VonNeumann computer. Imperative or procedural languages such as ISPS [26] and VHDL [28] are appropriate for this purpose. However, it is most likely the case that a description based on a simulation model will not prove suitable for automatic synthesis and formal verification of design. An analogous situation exists with models of computation for programming languages. Generally, the preferred programming language for a particular machine architecture is one in which the underlying computational model matches that of the architecture. Such matchings simplify both analysis and compilation and usually result in better performance. For example, on a multiprocessing machine where communication time between processors dominates local memory-reference time, a message-passing language is often used [164]. If effective communication time is reduced by the use of cache or efficient hardware implementation, a shared-memory model is more often selected [165, 166]. For these reasons, imperative programming languages are preferred when programming von Neumann architectures. This style of programming has become so pervasive that it is common to employ procedural languages even in digital hardware descriptions.

However, the behavior of digital hardware exhibits inherent parallelism and the procedural languages become clumsy when attempts are made to specify *what* the hardware is supposed to do without making any assumptions about *how* the behavior is to be obtained. Contrary to imperative languages, declarative languages allow clean behavioral description without any implication about the implementation. In a declarative domain, two major for-

malisms have been investigated by many researchers, especially in the field of formalizing programming languages. The strong connection between the formalization of mathematical logic and the formalization of computer programming languages was clearly recognized by Alan Turing as early as 1947, when he reported his expectation in a talk to the London Mathematical Society [167],

> "that digital computing machines will eventually stimulate a considerable interest in symbolic logic and mathematical philosophy. The language in which one communicates with these machines, i.e. the language of instruction tables, forms a sort of symbolic logic."

In software engineering, the formulation of valid specifications, the construction of efficient programs, together with a proof that the programs meet their specifications are the central problems facing the computer scientist [149, 167]. There have been a continuous stream of studies for establishing a formalism which is appropriate for the formal specification of programs and whose execution is practical and efficient. Currently, there are two major formalisms in the declarative domain: the formal language of *predicate logic* [115, 167, 167, 133] and the *functional language* [169, 159, 146]. In the formal language of predicate logic the behavioral specifications can be viewed as a set of constraints which the final design should satisfy. The order of the constraints does not affect the result of the total specification and each of the constraints narrows the domain of satisfaction. If the domain becomes null, that means the specification is over-constraining and no entity can satisfy all the constraints. Whether there exists an over-constraint or not can be checked by consistency checking among the constraints.

Since the whole of mathematics is too general to formalize with practical efficiency, it is unavoidable to employ some notational framework less powerful and less general. The

restricted notation is so designed that its use will be rewarded by more efficient execution on a computer. Even in the best known logic programming language, Prolog [138, 139], the meaning of "not" is not the same as the NOT of mathematics, logic, or even normal technical discourse, and its meaning cannot be fully understood except in terms of the way that Prolog programs are executed. In fact, in Prolog, the "and" and the "or" also have peculiar meanings, consequently they are not even symmetric. The motive here was to achieve greater efficiency of execution, particularly on traditional sequential computers [167].

In spite of considerable ingenuity of implementation, predicate logic approaches are inherently inefficient, particularly if the specification takes proper advantage of the power of the available combination of conjunction and disjunction. An existential quantifier multiplies the number of cases by a larger factor, and if recursion is involved, the number of cases to be explored increases exponentially. The alternative functional (or applicative) language approach is a compromise, suitable both for specification and for executable programs: in expressive power it is nearly as good as logic; and in execution efficiency it approaches a conventional procedural language. Where necessary, algebraic laws can be used to optimize a specification before execution. Functional formalism deals with structured data, is hierarchically constructed, and does not require the complex machinery of procedure declarations to become generally applicable. Function composition can use high level descriptions to build still higher level ones in a style not possible in conventional languages.

5.4. Overview of the Behavioral Verification System (BEAVER)

The behavioral verification system described in this section is the focus of the later part of this dissertation. A prototype framework which can verify automatically not only the functional correctness but also the presence of any timing violation has been developed.

The overall strategy employed for the behavioral verification is as follows. First, primitive language constructs are defined. The semantics of the primitive language constructs are based on functional formalism and are simple enough to be formally manipulated by machine while expressive enough to describe the behavior of hardware descriptions at any level above the circuit level, i.e. where signals are represented by discrete values.

Based on the primitive constructs, libraries are built up. Since the semantics of a cell instantiation is the same as that of a function call in a functional programming environment (by definition and by common use), composing functions is quite natural and easy. The user can also define his or her own cells on top of the built-in cells. Each of the cells can be used either in a behavioral description or in a structural description. In this case, there is no distinction between behavior and structure in cell descriptions -- such terms simply represent a particular interpretation of the description. Hence, equivalence checking can be performed without any distinction between a "behavior" and a "structure", between two behavioral descriptions, or between two structural descriptions.

From the structural information available in the description of a cell, the timing behavior is abstracted when a cell is defined. While timing information is abstracted, checks can be performed to determine whether there are any timing violations within the cell definition. If so, the cell definition is inconsistent and should not be accepted. If the user still wants to define the cell without any consideration of timing behavior, only the function of such a cell can be described, even in a cell where the next higher level cell deals with timing behavior, as will be seen later.

To handle the complexity of VLSI design, a mechanism which can exploit the hierarchy and the design regularity is required. The hierarchy is automatically exploited as follows. When the specification and the implementation share the same hierarchy, as illustrated in Fig-

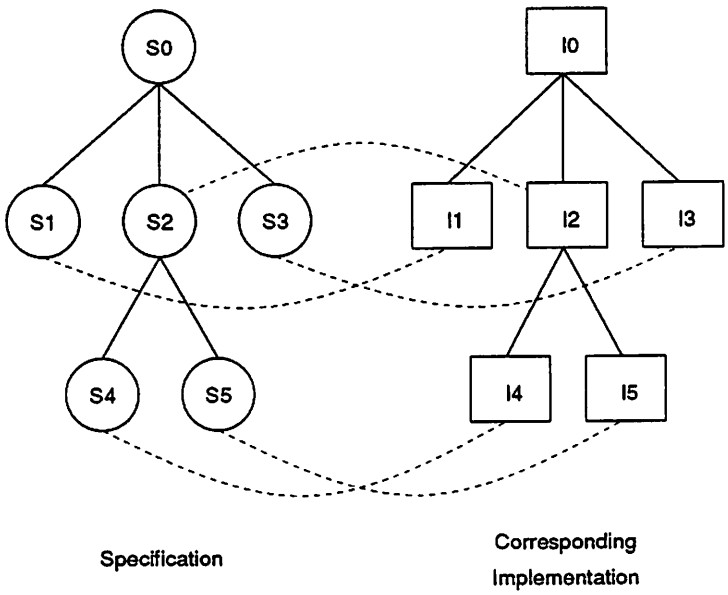


Figure 5.2 Specification and implementation sharing the same hierarchy

ure 5.2, it is easy to exploit the hierarchy during verification. In a verification system which assumes the same hierarchy between the specification and the implementation, the specification and the implementation of a module are usually described in one unit. However, in general, the specification and the implementation might not share the same hierarchy. Hence, in this dissertation, the research focuses on a verification method that can exploit hierarchy even when the hierarchy trees have different structures. Consider the case illustrated in Figure 5.3. In verifying the equivalence of Cell A and Cell B, if some of the pairs of (A_1, B_1) and (A_2, B_2) are proved equivalent, these facts are utilized. Otherwise, the cells are flattened down one level lower and an attempt is made to utilize the already-proven facts

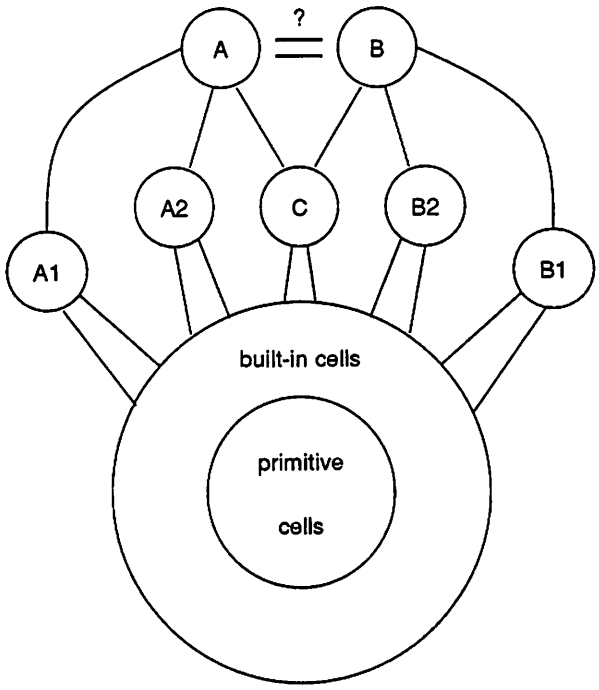


Figure 5.3 Exploitation of hierarchy

again. This procedure is repeated until the cells are flattened-down to primitive cells.

5.5. Primitive Language Constructs

As mentioned earlier, the description language in the verification system BEAVER is based on functional formalism. One of the most important concepts in a functional formalism is the concept of *value transformation* denoted by a description. The behavioral description of a hardware component can be viewed as a transformation applied to objects, which performs certain operations. The semantics of those descriptions or transformations can be

captured through observing how the objects are mapped to new values. As described in [163], in digital hardware, the objects are usually signals, busses, or devices, these are viewed differently according to the context and conceptual level in which they are considered. At the lowest level, signals are considered to represent voltages and currents; at the next, logic levels with various strength; at the higher levels, bits, numbers, addresses, or even computational objects. This abstraction of objects is very important in a hardware description language.

In this section, the primitive language constructs of the verification system are introduced. The predefined types and the type-defining mechanism are described as well as how a cell is defined and the meaning of a cell.

5.5.1. Types

The concept of abstraction of data objects was introduced in the structured programming technique, many years ago [116]. The abstract data type, based on the concept of data abstraction, has evolved as a major structuring mechanism in program development, and is supported by most of the modern programming languages, under such different names as class [170, 171], package [172], form [124], and module [173]. In general, data abstraction comprises a group of related functions or operations that act upon a particular class of objects [117]. The behavior of a hardware device or system can be thought of as performing certain operations upon objects and can be represented by transformations applied to data objects.

Most modern hardware description languages agree that a rich variety of data types and data structuring facilities are required, usually with strong type checking included in the compiler. Analogous to syntax and type checking in program compilation, a certain amount of static checking can be performed in a hardware verification process. Static checking, such as checking to ensure that no signal wire is driven by two outputs simultaneously, that every

line has an output and an input, and that every output and state variable is accounted for by a value assignment, is useful in detecting many elementary errors. Also the signal types can be checked for compatibility.

The behavioral description language of BEAVER is strongly typed and the type checking is performed statically. In the language, every named object should have a type associated with it. Four types are predefined and three mechanisms of type definition are provided for data abstraction.

The predefined types are: *bool*, *nat*, *sync*, and *time*. The type **bool** is for Boolean values of *true* and *false*. The type **nat** is for natural numbers including zero. These two types comprise the elements of the basic value-domain of the system. The remaining two types are used for the description of timing behavior. The type **sync** differentiates the type of synchronizing variables from other variables. The type **time** is used for the measure of temporal distance. The value of the **time** variable might denote real-time in nanosecond or in another unit; however, it might have no connection with "real-time". The connection of meanings between the type **time** and real-time is completely up to the user.

The three type-definition mechanisms of BEAVER are: *enum*, *array*, and *struct*. The LISP style syntax of type definition is shown as follows.

(**deftype** type-name [enum_type_def | array_type-def | struct_type-def])

Each type is defined with a unique name. When names are allowed in type definitions, two notions of equivalence of type expressions arise, depending on the treatment of names -- *name equivalence* and *structural equivalence*. Name equivalence views each type name as a distinct type, so two type expressions are name-equivalent if and only if they are identical. Under structural equivalence, names are replaced by the type expressions they define, so two type expressions are structurally equivalent if they represent two structurally equivalent type

expressions when all names have been substituted. In the verification system for the efficiency of type checking, name equivalence is employed.

The keyword **enum** is used for enumeration type definition. In digital hardware design, especially in the architectural level, when the detailed encoding schemes for specific types (e.g. the opcode of a microcomputer) are not yet determined, it is convenient for the user to introduce a symbolic encoding. Then the enumerated names can be replaced by the final binary encodings later. This construct can also be used for the definition of a basic type which is not provided in the primitive set of the language. For example, when the user wants to define a type called *bit* instead of using the built-in type bool, the following definition can be put in the head of the description:

(**deftype** bit (**enum** 0 1 X))

Then the defined type of bit can be used anywhere after its definition. Note that the "encoding" of a binary value can be thought of as equivalent to phase assignment [98].

The keyword **array** is used for the definition of indefinite array type. The indefinite array is quite different from the commonly accepted array type. In the usual array type, the size of the array is fixed or predefined, while in the indefinite array the size is not fixed. The type of the element can be any type which has previously been defined but the type of each element should be common in an array. For example, in the following type definition, each element of the array is of type bit, but the size of the array is undefined.

(**deftype** bit-vector (**array** bit))

The actual size of the array is determined when an object with type bit-vector is parameterized within a recursive cell definition.

The keyword **struct** is used for the definition of a structure (or record) type. While an array is an aggregate of elements of the same type, a structure is an aggregate of nearly-arbitrary types. For example:

```
(deftype bus (struct (flag bit)
                     (data bitv)))
```

defines a new type called *bus*, consisting of two items, flag and data, whose types are bit and bitv, respectively.

Some HDL's, such as VHDL, allow subtypes. A subtype of a type is a subset of values of the type. A subtype of a type usually consists of a contiguous subset, for example: a natural number as a subtype of integer. As far as a language itself is concerned, subtypes are elegant and useful. However, there is a price to pay for their inclusion. In the verification system, it was decided to have no overloading of types (i.e. subtying) as it was judged that the consequent improved performance of the verifier and the clearer diagnostics of type checking outweighes the advantages of introducing subtypes.

In some HDL's there are a large number of types built into the language [158, 28], and in others, such as ELLA, all types are user defined. In BEAVER, four primitive types are built-in and an arbitrary number of types can be constructed. To the hardware designer, as an end user, it should not matter much whether types are built-in or are part of a library. In ELLA, unions are provided so that different sorts of objects on the same devices might share the same memory space. However, in a formal approach, every type conversion should be expressed explicitly. In the verification system presented here, a type conversion is achieved through a function application which reconfigures the structure by type-casting the member slots of the type.

5.5.2. Cells

The basic description unit of an object is called a *cell*. Each cell has its declarations of input, output, state, and local variables. For the verification to be performed within reasonable cpu-time, the state mapping information is provided. Also a directive for cell expansion at definition time is provided. This construct is useful for the verification of timing constraints and will be explained in the next chapter in more detail. The syntax of a cell definition is shown as follows:

(**defcell** cell-name
[(**input** arg-list)]
[(**output** arg-list)]
[(**state** arg-list)]
[(**local** arg-list)]
[(**mapping** map-list)]
[(**directive expand**)]
body)

The arg-list is a list of pairs of a variable and its type as follows:

(var1 type1) (var2 type2) ···

The map-list is a list of pairs of two variable as follows:

(var1 another_var1) (var2 another_var2) ···

In the verification of special leaf cells presented in Chapter 3, it is assumed that no state mapping information is available. However, in behavioral descriptions the mapping information is assumed to be retained during the design process. In a hierarchical design, a behavioral specification and its structural implementation are generally on different abstraction levels. For the purpose of formal verification, the implementor of a specification has to describe how the specification is mapped into the environment of an implemented structure. The correspondence between two levels is called *mapping*. From the specification point of view, this mapping is in fact the designer's decision, made while achieving a particular

implementation from a given specification. By the mapping, a specification can be interpreted in the environment of the structural implementation, furthermore the interpretation can be automated. In BEAVER, only minimal state mapping information is assumed.

5.5.3. Assignments

The body of a cell consists of assignments. The syntax of an assignment is shown as follows:

(**let** var-name expression)

To avoid aggregating every multiple object when values are returned from a cell, a multiple-valued assignment is allowed as follows:

(**let** var-list multiple-valued-expression)

Here the meaning of an assignment is closer to that of an equality in mathematics rather than to that of an assignment of usual imperative languages. In procedural languages variables are often used as a temporary scratch-pad; they are assigned a value and then reassigned another value within the same section (scope) of a program. On the other hand, in functional languages, a variable may be assigned an expression only once within a scope, while allowing the variable to be read as many times as desired. This is called the *single assignment rule* and is found in data-flow languages such as VAL [174], and ID [175]. In functional languages, variables are not storage locations, but are like macro names for expressions.

5.5.4. Recursions

A cell can be defined by a recursive function. The principle is based on the notion of well-founded relations. In particular, (cell $x_1 \cdots x_n$) may be defined to be some term involving recursive calls of the form (cell $y_1 \cdots y_n$), provided there are a measure and a

well-founded relation such that in every recursive call the measure of the y_i is smaller, according to the well-founded relation, than the measure of the x_i [141].

It is assumed that a particular cell '<' is well founded and when an indefinite array is defined, two axioms are added to the knowledge-set of the verification system, which are required for the inductive proof involving the cell. This is explained in Chapter 7.

5.6. Primitive Cells

With the primitive language constructs, some cells are provided and these cannot be derived from other cells. In this sense, they are called *primitive cells*. The primitive cells are as follows: *if, equal, 1+, 1-, values, nth, type-of, delay, and prev*. In the following, each primitive cell is described in more detail.

Since the system is based on the functional paradigm, there is no predicate of the usual meaning. Hence instead of formulas, there are only terms. The functions which return a value of type bool can be viewed as predicates. The cell **if** is used to compose terms involving conditional expressions. The syntax of if cell is as follows:

(**if** bool-term term1 term2)

The first argument bool-term should have bool type and the types of the two argument, term1 and term2 should have the same type. When the bool-term is true term1 is returned otherwise the term2 is returned.

The cell **equal** returns a value of bool type.

(**equal** lhs rhs)

When lhs term and rhs term denote the same value, the result is true, otherwise the result is false. Since the language is strongly typed and the type checking is performed while a

description is read in, it is guaranteed that the types of the terms lhs and rhs are equal.

The two cells of

(1+ nat-term)

(1- nat-term)

are the successor and the predecessor functions of natural numbers. When the argument of the cell **1-** is zero, the result is zero.

The cell **values** is used to aggregate multiple values as one return value and the cell **nth** is used for the accessing of a value from the aggregation.

(**values** val_1 val_2 $\cdots$)

(**nth** nat-term multiple-valued-term)

In the cell "values", each of the type of val_i can be an arbitrary valid type. In the cell "nth" the first value of the aggregation is indexed with one rather than zero, as in some programming languages.

The cell **type-of** returns the type of a term. The syntax of cell definition is shown as follows:

(**type-of** term)

This cell is mainly used for the *polymorphic* [176] cell definition. An ordinary cell allows the body to be expanded with arguments of fixed types; each time a polymorphic cell is called, the body of the cell can be instantiated with arguments of different types.

In general, the term "polymorphic" can also be applied to any language constructs, so both polymorphic cells and polymorphic operators can be used. Built-in operators for indexing arrays, applying functions, and manipulating pointers are usually polymorphic because they are not restricted to a particular kind of array, function, or pointer. Polymorphic

functions are attractive because they facilitate the description of algorithms that manipulate data structures, regardless of the types of elements in the data structure.

In BEAVER, polymorphic cell definition helps the user describe a cell, which is then representative of all cells, without which cell definitions would have to be written many times separately. The typical usage is illustrated as follows:

(**defcell** cell-name
  (**input** (x) (y (**type-of** x)) ···) ···)

5.7. Macros

A macro definition mechanism [177] is provided to provide syntactic clarity of descriptions. The usage is as follows:

(**defmacro** name form)

The form can be either a simple symbolic constant or a general term. The macro is valid, after its definition, in every cell definition.

As an example of the use of the macro command, a description which deals with enumeration types, the conditional assignments usually have many branches. Sometimes the description might get lengthy and difficult to read if only the primitive cell "if" can be used. To ease the description of multi-branching conditionals, the macro cell *case* is provided. Semantically it is nothing but a short hand for a list of "if" cells, although it may sometimes provide great convenience and improved readability. The syntax is as follows:

(**case** ($enum\text{-}var_1$ ··· $enum\text{-}var_n$)
  (($enum\text{-}value_1$ ··· $enum\text{-}value_n$) $term_1$)
  ···
  [(**otherwise** $term_k$)])

Here the enum-var's are variables with some enumeration type and each of the enum-values

are one of the enumerated constants of each type. The last optional term of "otherwise" is the default case. The type of $term_i$ should be compatible to each other, and the return type of a case cell is determined by the type of $term_i$. Note that each conditional branch should be mutually exclusive and when there is no **otherwise** term, the cases of all branches should be exhaustive to avoid any undefined values. The current implementation checks the type compatibility between $enum\text{-}var_i$ and $enum\text{-}value_i$ and between $term_i$'s.

5.8. Conclusions

In this chapter, a behavioral description language has been presented, which is used in the following two chapters for the description of timing and functional behavior. The semantics of the primitive language constructs is based on functional formalism and is simple enough to be formally manipulated by machine. For the description of timing behavior, two language constructs are included. For data abstraction, three type defining mechanisms are provided. Also, recursive definitions are supported.

CHAPTER 6

HIERARCHICAL TIMING VERIFICATION

6.1. Introduction

The design of a VLSI system involves many decisions relating to several levels of timing detail. The typical design decisions include choices of timing disciplines, such as *synchronous* [153, 178, 179, 180] versus *asynchronous* [153, 180] disciplines; circuit design paradigms, such as static versus dynamic circuits in CMOS; the number of clock phases; and the temporal behavior demanded of specific input signals, such as intervals of time over which they must be stable. Usually the abstraction levels of timing behavior are not well-categorized and are less generally accepted than those of functional behavior. This is due to the fact that no well-defined framework exists for dealing with the various aspects of system timing.

The two very different disciplines of design, synchronous systems and asynchronous systems originate from the two different viewpoints about the connection between sequence and time. One major approach to asynchronous VLSI system design is *self-timed* [181] methodology. In self-timed systems, all signals are "held up" as necessary until the slowest one arrives. Sequence and time are connected in the interior of system components called *elements* and all system events are assured to occur in proper sequence rather than at particular times. In synchronous systems, which are a prevailing discipline for current digital VLSI system designs, system-wide synchronizing signals are generated by a clock generator. These clock signals are used for synchronization by holding up signals periodically to equalize the

delays. Hence in synchronous system, sequence and time are connected through clock signals.

Whatever system timing discipline has been chosen, the timing of each of the signals in the system must be properly managed for the successful operation of the system. Otherwise, the system will have timing problems or will fail to meet its performance objectives. When the system size was small, it was possible to design the system so that all the signals arrived at intended places in the intended sequence. However, as the system size grew larger and the design involved more complex work, the prediction of propagation delays of signals through the system became more difficult and managing all signals became extremely complex. To check the correctness of timing, the logic or circuit designer could utilize computer-aided design tools such as simulators. However, simulation is not guaranteed to locate the correct longest delay of a system unless the system is simulated with all possible input vectors. On the other hand, timing verifiers carry out only one or a few analyses using an approximate delay model and report much of the information necessary to improve or correct the design. Therefore, timing verification is preferred for the proof of absence of any timing violation in a digital VLSI system.

Since the work of Kirkpatrick and Clark [182], much research and development work on timing verification (eg. [183, 184, 185, 186, 3, 187, 4, 5, 188]) has been performed. However, most of these techniques assume that the design description is flattened down on a particular level (typically gate-level or switch-level) and is inappropriate for the use in hierarchical design environments.

In this chapter, timing constraints of synchronous digital systems are described. Then the hierarchical timing verification method with the timing model employed in this dissertation is explained. Since the functional relationships of input-output pairs are available while

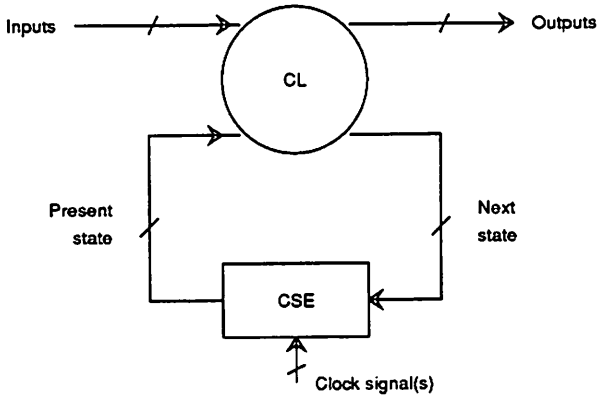
timing verification is performed, the *static-false-path* (or *static-insensitizable-path*) problem which is one of the important problems of timing verification is eliminated. Recently, a study [189] has been performed which illustrates that static timing analyzers generally cannot guarantee an upper bound of the worst case delay for some pathological circuits. Also, due to the nature of the abstraction mechanism of timing information for sequential cells, the timing verifier can be used as a "timing consultant" to determine the clock separation time needed for each phase. Finally, experimental results of the timing verification are illustrated.

6.2. Timing Constraints of Synchronous Digital Systems

In the discipline of synchronous design, the *clock* signal serves two purposes [153]. The clock is a sequence reference and also a time reference. As a sequence reference, its transitions serve the logical purpose of defining successive instants at which system state changes may occur. As a time reference, the period or interval between clock transitions serves the physical purpose of accounting for element and wiring delays in paths from the output to the input of clocked elements. The timing constraints of synchronous system are usually derived from the clock period and propagation delays of combinational logic block between two clocking elements.

6.2.1. Clocked Storage Elements

The simple logical model that synchronous systems resemble is the finite-state machine (FSM) [153]. As illustrated in Figure 6.1, any such system must satisfy a topological requirement that every closed signal path pass through a *clocked storage element.* The clocked storage elements are used in a synchronous system to hold up the movement of signals to the next stage of the combinational logic until a predefined amount of time has passed. Closed



CL: Combinational Logic
CSE: Clocked Storage Element

Figure 6.1 Simple clocking scheme of synchronous system

paths that do not pass through clocked storage elements are excluded as they may create non-deterministic behavior, either through oscillation or through asynchronous latching. With this constraint on the logic design the designer is relieved of any requirement that the combinational logic be free of transients (static or dynamic hazards) on its outputs. The only dynamic characteristic of a combinational net that matters is its propagation delay time.

As will be explained shortly, the timing constraints for the system depend upon the type of clocked storage elements used in addition to the clocking strategy. One type of clocked storage element is the *edge-triggered* type. The edge-triggered, clocked storage element samples and latches its input data value during a short period (sampling interval) around a rising or falling clock edge, and changes its output state on the clock edge. If a rising clock edge

causes the element to change the output state, the element is called positive edge-triggered type. Otherwise, the element is called negative edge-triggered type. Another type of clocked storage elements is the *transparent* type element. These are more popular than the edge-triggered type in MOS VLSI designs. There are *pulse-width sensitive* elements such as the JK master-slave flip-flop, but due to the glitch-catching problem [190], they are rarely used. In a transparent clocked storage element, the clock node works as an enable signal node, which makes the element transparent (i.e. the output follows the input) when the enable signal is active. The element may be active while the enable signal is high or may be active while the enable signal is low. The transparent type is also an example of the *level sensitive* type, because the element becomes transparent when the enable signal is active. Note that, unlike the edge-triggered element, the output of the transparent type element follows the input data as soon as it becomes available during the enable period.

6.2.2. Timing Constraints with Edge-Triggered Type Elements

In a system using edge-triggered elements, it is important to keep the input data unchanged during the sampling interval at an activating clock edge to guarantee that the correct input values are latched. A *setup time* [178, 180] (T_{setup}) is the required time an earlier input data value must be held stable prior to the activating clock edge. It is important to note that a "0" setup time does not imply there is no setup time. A T_{setup} of "0" means that the input signal value cannot stabilize later than the occurrence of the clock edge. A *hold time* [178, 180] (T_{hold}) is the required time an input data value must be stable following the activating clock edge. If the hold time is negative, the input data value may change before the activating clock edge. A negative hold time is often called a *release time* [178, 180]. Generally, if the slowest arriving signals settle at the input nodes of clocked storage elements by

the setup time before an activating clock edge, they will usually remain stable until the end of the hold time, unless the combinational logic blocks are extremely fast. Since the hold time is usually negligibly smaller than propagation delays through logic paths, or sometimes even negative, many timing verifiers check only whether or not the input data of clocked storage elements become stable before the setup time.

Consider a single-stage clocked path, illustrated in Figure 6.2. Suppose that CSE_i and CSE_j are edge-triggered type. Then the propagation delay of the combinational block, D_{CL}, must satisfy the following constraints to ensure that input data to CSE_j settles before the

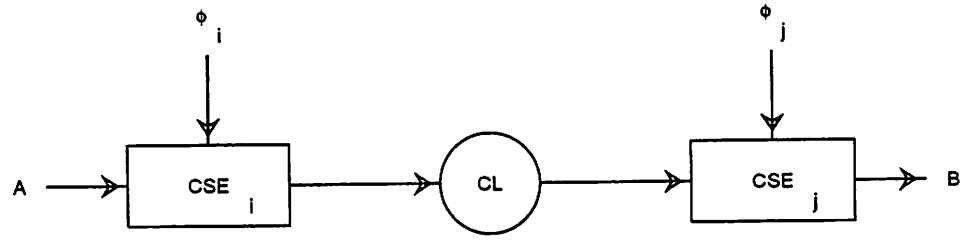


Figure 6.2 Single-stage clocked path

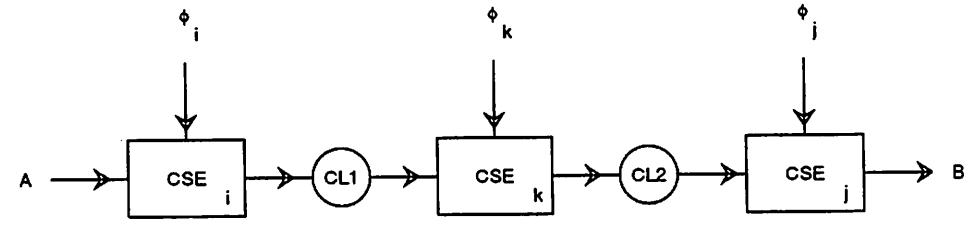


Figure 6.3 Multistage clocked path

setup time:

$$D_{CL} \leq [t(\phi_j, ACTIVATE) - T_{setup}(CSE_j)] - t(\phi_i, ACTIVATE) \quad (6.1)$$

where t is a function mapping the clock phase into real time on a timing chart. The second argument determines the edge in which the user is interested. Note that $t(\phi_j, ACTIVATE)$, a timepoint at which the activating clock edge of ϕ_j of CSE_j occurs, is the next closest one to ϕ_i on the timing chart. If the clock signals shown in Figure 6.4 are used for synchronization and the two clocked storage elements are positive edge-triggered type, Equation (6.1) can be rewritten as follows:

$$D_{CL} \leq t3 - t1 - T_{setup}(CSE_j)$$

The timing constraints of a multistage clocked path, illustrated in Figure 6.3, are derived in the same way as in the case of a single-stage clocked path. Any multistage

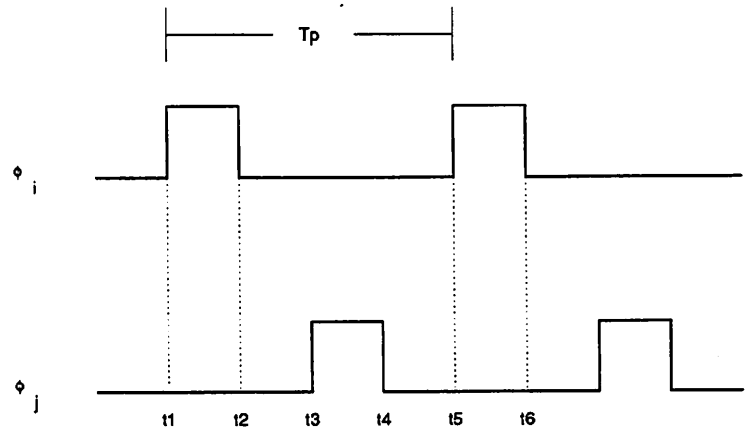


Figure 6.4 A clock signal example

clocked path using edge-triggered clocked storage elements, including the one which constrains a clock period by forming a loop, satisfies timing constraints automatically if each of its constituent single-stage clocked paths satisfies timing constraints. Thus, when verifying a system which employs edge-triggered clocked storage elements, timing verification can be performed by obtaining the maximum propagation delay through each single-stage combinational logic block and applying the Equation (6.1).

6.2.3. Timing Constraints with Transparent Type Elements

Unlike in a system using edge-triggered type elements, in a system using transparent type elements, input signals are sampled during a short time interval just before the inactivating clock edge -- this is the setup time of transparent type elements. In order for correct values to be latched, input signals must remain stable during this setup time.

Consider a single-stage clocked path, illustrated in Figure 6.2. Suppose the clocked storage elements CSE_i and CSE_j are of the transparent type. To ensure the correct input data to be latched, the propagation delay D_{CL} must satisfy following constraint:

$$D_{CL} \leq [t(\phi_j, INACTIVATE) - T_{setup}(CSE_j)] - t(\phi_i, ACTIVATE) \quad (6.2)$$

In the above inequality, $t(\phi_j, INACTIVATE)$ is the next closest one to $t(\phi_i, ACTIVATE)$ on the timing chart. If the clocked storage elements CSE_i and CSE_j are the same one, i.e. the clocked path forms a loop, the first term of Equation (6.2) should be replaced by $t(\phi_i, ACTIVATE)$. If a particular single-stage clocked path forms a loop, the path is employing a single-phase clocking strategy. In this case, the delay of the combinational block constrains a clock period. Otherwise, the delay constrains a clock separation, which is a separation between clock edges of different phases. Note that a single-stage clocked path using transparent elements may use both the time interval between the activating and inactivating

edges (or vice versa) of preceding clock phase and that of succeeding clock phase for logic evaluation, unless it forms a loop. In Figure 6.2, if CSE_i and CSE_j are positive-active transparent type and clock signals illustrated in Figure 6.4 are used for synchronization, Equation (6.2) yields the following constraint:

$$D_{CL} \leq t4 - t1 - T_{setup}(CSE_j)$$

In a system using transparent type elements, the timing constraints of a multistage clocked path are not as simple as in a system using edge-triggered type elements. This is much better understood with a specific example rather than with an explanation of a general case. Consider an example illustrated in Figure 6.5 where all the clocked storage elements are of a positive-active transparent type. The logic evaluation times of block CL1, CL2, and CL3 are from $t1$ to $t3$, $t2$ to $t5$, and $t4$ to $t6$, respectively. However, even when each of the three single-stage clocked paths satisfy their corresponding single-stage timing constraints, the resulting multistage clocked path may not satisfy timing constraints which are required for correct operation. For example, even if the delay through the block CL1, D_{CL1}, and the delay through the block CL2, D_{CL2}, satisfy the following constraints, respectively,

$$D_{CL1} \leq t3 - t1 - T_{setup}(CSE2)$$
$$D_{CL2} \leq t5 - t2 - T_{setup}(CSE3)$$

it is not guaranteed to satisfy this constraint:

$$D_{CL1} + D_{CL2} \leq t5 - t1 - T_{setup}(CSE3)$$

Therefore, when a multistage clocked path is examined, all of its nested clocked paths must be examined. If a multistage clocked path has N constituent single-stage clocked paths, it has $\sum_{i=0}^{N} i$ nested clocked paths. Since the path of Figure 6.5 has three constituent single-stage paths, the following six nested clocked paths must satisfy their timing constraints in this particular example:

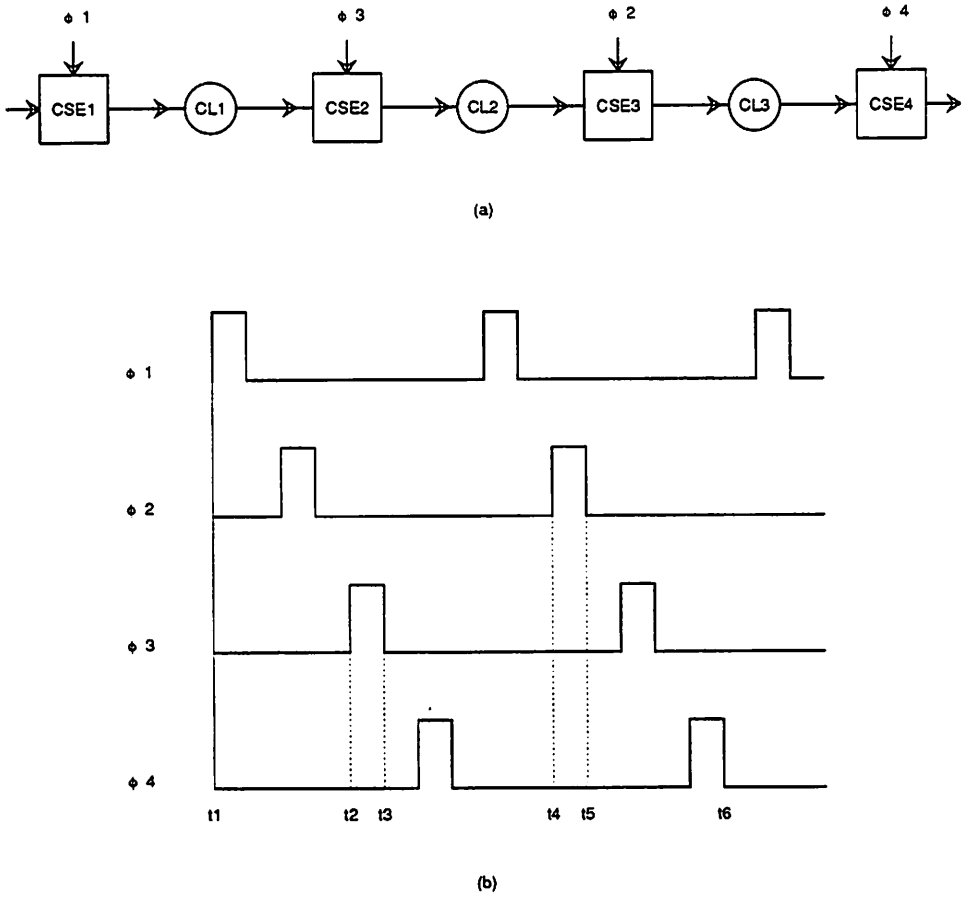


Figure 6.5 An example of three-stage clocked path in a four-phase clocking system

(1) Single-stage nested clocked paths:

$$D_{CL1} \leq t3 - t1 - T_{setup}(CSE2)$$
$$D_{CL2} \leq t5 - t2 - T_{setup}(CSE3)$$
$$D_{CL3} \leq t6 - t4 - T_{setup}(CSE4)$$

(2) 2-stage nested clocked paths:

$$D_{CL1} + D_{CL2} \leq t5 - t1 - T_{setup} \quad (CSE\,3)$$
$$D_{CL2} + D_{CL3} \leq t6 - t2 - T_{setup} \quad (CSE\,4)$$

(3) 3-stage nested clocked path:

$$D_{CL1} + D_{CL2} + D_{CL3} \leq t6 - t1 - T_{setup} \quad (CSE\,4)$$

More extensive study of the timing constraints for a system using transparent elements and a system using mixed elements of edge-triggered and transparent clocked elements has been performed elsewhere and the results can be found in [180, 191].

Note that the reason the timing constraints of a multistage clocked path with transparent type elements are more complex is because each single-stage path with transparent type may take more delay time than in the case of edge-triggered type by the interval from the activating edge to the inactivating edge (i.e. by the duration of a clock pulse). For example, if clock signals shown in Figure 6.4 are used for the synchronization of the single-stage clocked path shown in Figure 6.2, the time interval from $t3$ to $t4$ is the difference. However, as previously explained, not all single-stage paths with transparent type elements are allowed to exploit the difference for its logic evaluation. Once the interval has been taken for the evaluation of the previous stage, then the delay time for the next stage should be shorter than the delay time it can otherwise take. Note that whenever each of the single-stage paths satisfies the timing constraints of a system using edge-triggered type elements (Eq. 6.1), it is guaranteed that the timing constraints of a system using transparent type elements are satisfied. In practice, most circuit designers avoid the complex timing constraints and employ a conservative approach to the allocation of the evaluation time for their combinational logic blocks. They usually use the timing constraints of a system with edge-triggered type elements even though they are using transparent type elements for their clocked storage elements.

In this chapter, the timing constraints of a system with edge-triggered type elements are considered.

6.3. False Path Problem

There are two possible approaches in propagating signals in a system for timing verification: *value-dependent* and *value-independent* propagations. In the value-dependent approach, when a signal at an input node of a logic block changes, its effect is propagated to the output node of the logic block only when the other input conditions support it. Conventional circuit and logic level simulators belong to this approach. On the other hand, in the value-independent approach, the input change is always propagated to the output node of the logic block without attempting to check whether the signal conditions at other input nodes functionally support that propagation. Most timing verifiers [182, 183, 184, 3, 4, 5, 188] use a value-independent approach. Consider an example illustrated in Figure 6.6 which compares the two approaches. For the simplicity of comparison, it is assumed that all logic gates (AND, OR, NOT) in Figure 6.6 have a unit delay. Also, it is assumed that the longest delay from input nodes A and B to an output node E is evaluated, when both inputs are applied to at time t=0. The results of signal propagation are shown in Table 6.1. The two cases in Table 6.1 correspond to values 0 and 1 for the primary input B. Input A has value a. When B is 0, the output E reaches its steady state (value 0) after 2 unit delay. The corresponding worst case delay path is Path B-(AND gate)-D-(OR gate)-E. When C is 1, the output E reaches its final value 1 in just 1 unit delay, even though the value of node D is settled down after 2 unit delay. In this case, the worst delay path is Path B-(OR gate)-E. On the other hand, when the value-independent approach is used, both the rising and the falling worst delays are 3 unit delays through Path A-(NOT gate)-C-(AND gate)-D-(OR gate)-E. Because

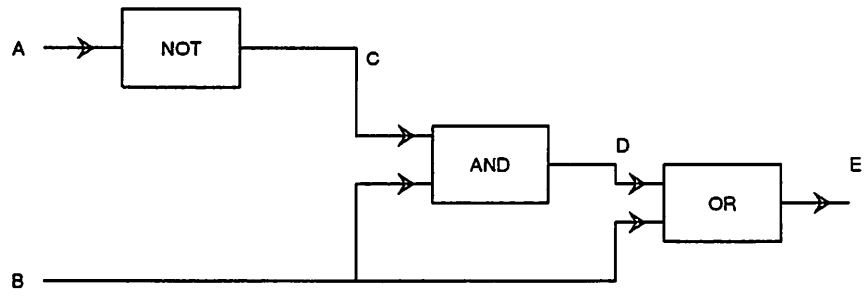


Figure 6.6 An example illustrating false path problem

case	time	node signals					critical-path
		A	B	C	D	E	
I	0	a	0	-	-	-	B-D-E
	1	a	0	$\bar{a}$	0	-	
	2	a	0	$\bar{a}$	0	0	
II	0	a	1	-	-	-	B-E
	1	a	1	$\bar{a}$	-	1	
	2	a	1	$\bar{a}$	$\bar{a}$	1	

Table 6.1 Propagation of signals in the circuit shown in Figure 6.6

these delays are the worst among all possible cases, they are called the *worst-case delays*. Value-independent worst delays are usually pessimistic, compared to value-dependent worst delays computed by simulation. In this example, the worst delay path is the one which can not be activated under real operating conditions. In Figure 6.6, in order for the signal change

at Node C to propagate to Node D through the AND gate, Node B must be at logic "1". On the other hand, in order for the signal change at Node D to propagate to Node E through the OR gate, Node B must be at logic "0". Because the two conditions conflict with each other, this value-independent worst delay path, A-(Buffer)-C-(AND gate)-D-(OR gate)-E, can not be activated. The paths that are detected by value-independent approach but cannot be activated under real operating conditions are called *false paths*. When the value-independent approach is employed, the user needs to perform a *case analysis* [184], which analyzes a number of different cases within a given circuit, one after another, to exclude false paths from consideration. In spite of the false path problem, the value-independent approach has mostly been employed due to its advantages of significantly reduced work in finding the critical paths in a system and test completeness. Recently, some studies [192, 193] have been performed to identify false paths by checking the consistency of the signal conditions that are necessary for paths to be activated.

In this chapter, it is demonstrated how the verifier uses the functional relationships it has available to eliminate static false paths. The approach of the verifier is value-independent in the sense that the user does not need to provide any values as inputs. Meanwhile, since the functional relations between signals are known in symbolic forms, the insensitizable paths can be detected and eliminated.

6.4. Timing Model

Formal approaches to the specification and verification of timing behavior has not been well investigated and no well-defined framework exists for dealing with the various aspects of timing behavior. The temporal logic approach has been suggested as a specification for timing and has been attempted in order to apply to synchronous digital systems as well as to

asynchronous systems. The use of temporal logic may help for asynchronous systems; however, there is no strong motivation to employ temporal logic in synchronous systems.

In this section, the timing model employed in this dissertation work is introduced. The timing model is aimed at solving the problem for specifying and verifying synchronous digital designs which results in the timing constraints explained in Section 6.2.

6.4.1. Two Language Constructs for Timing Behavior

Two language constructs are provided for the description of timing behavior: **delay** and **prev**. These two constructs are the names of primitive cells. The usage of the two cells are as follows:

(**delay** delay-value term)
(**prev** sync-name term)

The **delay** cell assigns a transportation delay [178, 180] to a term. The type of the delay-value is **time** which was explained in Chapter 5. The detailed meaning of the value is determined by the user. It may denote the quantity measured by the so-called "minimum resolvable time" [37] unit or the value in real time in some unit such as nanosecond or picosecond. The semantics of **prev** cell is determined by the returning value. The cell returns the value of the term at the closest previous instant of the synchronizing signal of "sync-name". Note that unlike most hardware description languages in which the temporal viewpoint is from the present to the future, in this work, the temporal viewpoint is from the present to the past [146]. This is more natural and more appropriate for functional formalism. The type of the sync variable is **sync** which was explained in the previous chapter.

6.4.2. Declaration of Global Synchronizing Signals

Global synchronizing signals are defined using **defsync** construct. The usage of defsync form is as follows:

(**defsync** period (sync-name1 time-point1) (sync-name2 time-point2) ···)

The period can be an arbitrary cycle period. It may represent a machine cycle or just one clock cycle or even multiples of several machine cycles. For a given period, as many synchronizing signals as necessary may be introduced. Each of the synchronizing signals may represent the rising or falling edge of a certain clock phase or other special timepoints in which the user is interested. As an example, consider Figure 6.7. The timing chart illustrated in the lower part of Figure 6.7 can be thought of as an *unrolled* version of the circular timing diagram. The origin of timepoints may be fixed arbitrarily by the user. Once the origin is fixed, the order of the timepoints of synchronizing signals is determined. In the example, all the rising and falling edges are declared as synchronizing timepoints.

6.4.3. A Modeling Example: Clock Skew

In this subsection, a simple modeling example is presented to illustrate how the two language constructs for timing are used.

In a synchronous system, clock signals are distributed to various points via conducting wires. Even though clock signals are usually distributed by metal wires to reduce the delay owing to the parasitics in a VLSI system, there is a variation in the arrival time of clock signals to different clocked storage elements due to different propagation delays. This arrival time variance is called *clock skew*. Clock skew changes both the effective clock period and the clock separations and, as a result, generally impacts system performance adversely or even causes timing errors. An extreme clock skew may turn nonoverlapping clocks into

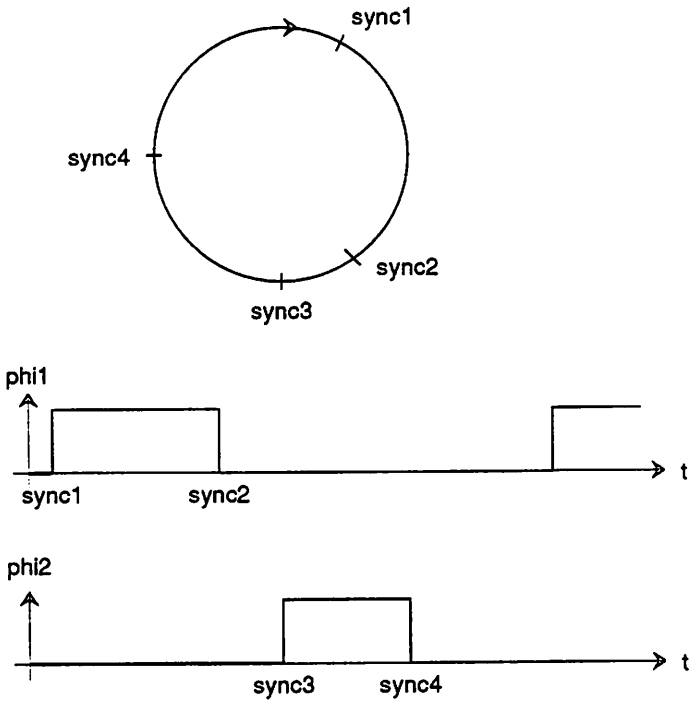


Figure 6.7 An example of a global synchronizing signal declaration

overlapping clocks and may introduce a two-sided timing constraints. Note that clock skew is relative to signal delay to a particular logic gate and is not a function of the absolute delay of signals in the circuit. Since it is difficult to predict clock skew and clock timing exactly, the control of clock skew is a problem in large systems, including VLSI systems. Although the clock delay can be reduced by placing re-shaping circuits at intervals along a long line, the overall delay is still fairly long compared to the propagation delay through a single gate.

A popular method of looking at the influence of clock skew is to use a simple delay transformation. In a logic block, if a delay (which can be positive or negative) is added to every input and the same delay subtracted from every output, the cell's timing remains unchanged [180]. This delay transformation provides a simple way to look at the influences of clock skew on the timing constraints. Suppose a clocked storage element, CSE, has a clock skew of D_{skew}. The clock skew on the clock signal path to CSE can be eliminated by adding a delay D_{skew} to the logic path following CSE, while the same delay is subtracted from the other logic path which precedes CSE. This transformation is shown in Figure 6.8.

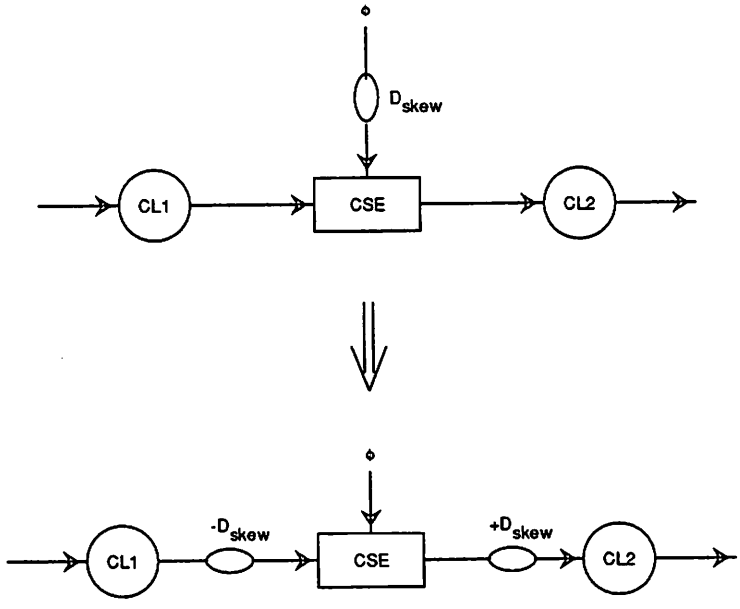


Figure 6.8 Clock skew modeling example

The clocked storage element cell, CSE, can be described in BEAVER as follows:

```
(defcell CSE                                          ; cell name
  (input (phi sync) (in bit))                         ; input declaration
  (output (out bit))                                  ; output declaration
  (let out (delay +D_skew (prev phi (delay -D_skew in))))   ; body
```

6.5. Abstraction of Timing Behavior

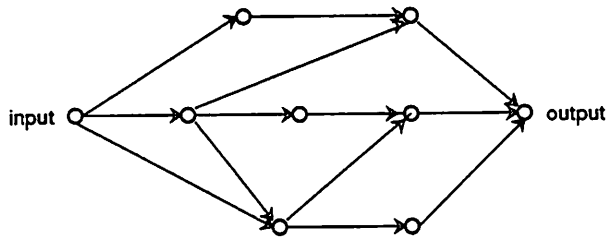
In general, it is a self-consistency property that is checked in timing verification while in functional verification it is a cross-consistency between two descriptions. In other words, timing verification can be performed within a cell definition, while functional verification requires at least two definitions for comparison. In BEAVER, when a cell is read and checked, the timing behavior is abstracted and if there are any timing constraints to be satisfied, they are checked during this phase. When the verifier detects any timing violation, it reports the cell name in which the violation occurred and the cell definition is stored without any of its timing information.

Every cell falls into one of two classes from the timing viewpoint: *combinational* and *synchronous*. A combinational cell is composed of only combinational subcells. In a combinational cell, the only timing cell allowed is **delay**. A synchronous cell can have either combinational cells or synchronous cells as its subcells and it should have at least one synchronous cell which uses the construct **prev**. In a synchronous cell, the two timing cells can be used arbitrarily in a description. This section explains how cells are abstracted.

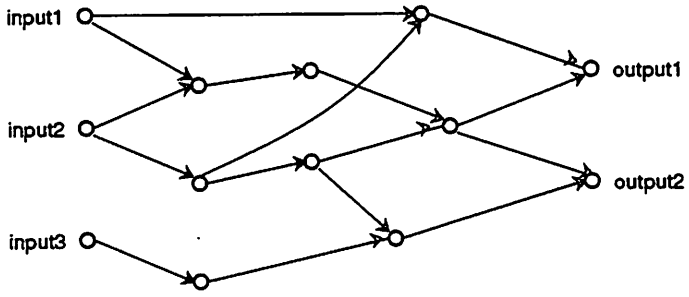
6.5.1. Combinational Cells

Combinational cells are abstracted using worst-case delay values. When a cell is single-input and single-output as illustrated in Figure 6.9 (a), the meaning of worst-case delay

value is clear. In graph-theoretical terms, it represents the sum of the weights of the longest path from the source node, *input*, to the sink node, *output*. Most methods of static delay computation choose the worst-case delay among the delays between the pairs of input and output nodes as a delay for a multi-input multi-output logic block. For example, in Figure 6.9 (b),



(a) Single-input single-output case



(b) Multi-input multi-output case

Figure 6.9 Abstraction of combinational cells

the worst-case delay value is the single largest value among the six delay values of the paths from $input_i$ to $output_j$, where $i = 1, 2, 3$ and $j = 1, 2$. This simple method is faster than the method which will be described in the following. However, this simple method tends to overestimate path delays too much. In this dissertation, the following more accurate approach, is employed. For a multi-input multi-output cell, the abstraction becomes the set of worst-case delay values for each output from each of the inputs. Each value is then associated with the input-output pair so that when the cell is instantiated within another cell, the delay value is correctly interpreted to avoid overestimation.

As mentioned earlier, since the input-output functional relationships are available, static false paths can be eliminated without any further significant amount of work. When a cell is read, all terms that contain *if*-forms are normalized to make later functional verification process more efficient. The normalization is essentially based on the following equality.

```
(if (if t11 t12 t13) t2 t3) = (if t11 (if t12 t2 t3) (if t13 t2 t3))
```

This rewrite rule is applied to a term recursively so that no if-form appears as its first argument in any term or sub-term. During this normalization, at each node of an if-form whenever a true or false case branch is traversed, the test condition is appropriately assumed as false or true, and the information prevents insensitizable paths from contributing to the longest path.

For example, the normalization process for the circuit shown in Figure 6.10 [3] is as follows:

```
t = (if y (delay 5 x) (delay 100 x))

z = (if (not y) (delay 5 t) (delay 100 t))
  = (if y (delay 100 t) (delay 5 t))
  = (if y (delay 100 (if y (delay 5 x) (delay 100 x)))
          (delay 5 (if y (delay 5 x) (delay 100 x))))
  = (if y (if y (delay 105 x) (delay 200 x))
```

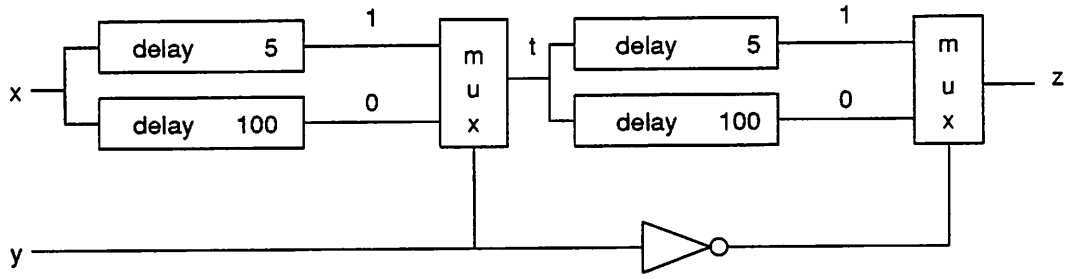


Figure 6.10 An example to illustrate the normalization process

```
        (if y (delay 10 x) (delay 105 x)))
= (if y (delay 105 x) (delay 105 x))
= (delay 105 x)
```

First, the output z is expressed as a function of an input variable y and an intermediate variable t and the true-case and the false-case term are exchanged with complementing the test-case. The intermediate variable t is replaced by its value. The delay is propagated to its arguments. Now when the true-case of the first-level if-form is normalized, since y is assumed true, the true-case term of the second-level if-form is returned. Likewise, when the false-case of the first-level if-form is normalized, the false-case term of the second-level if-form is returned. Finally, since the true-term and false-term are equal, the result is given as any one of the two. If the functional relations are not considered, the worst-case delay would be the maximum value among the terms, which is 200 instead of 105.

Now consider Figure 6.11 where two hierarchical levels are used to describe the same circuit as shown in Figure 6.6. As in the case of Figure 6.6, each gate delay is assumed as

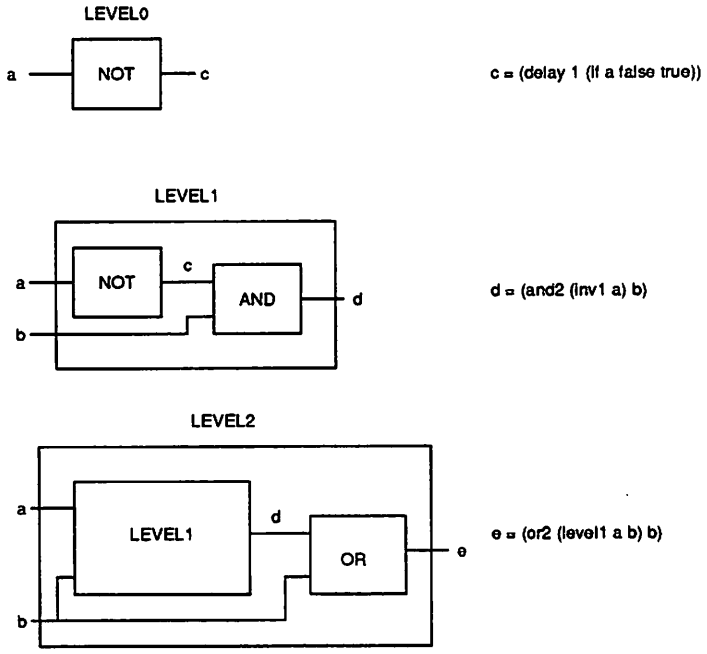


Figure 6.11 Hierarchy and directive for expansion

one. Since the timing verifier does not flatten subcells when it abstracts a cell, the worst delay of the circuit described in level 2 is evaluated as three. However, if the same circuit had been flattened completely, the worst delay would be two. To provide the user with the control of flattening cells selectively, a special language construct called *directive* is provided in BEAVER. This construct tells BEAVER to expand the current cell one level down. Thus, to evaluate the delay of the circuit correctly, the cells should be described as follows:

```
(defcell level-1
  (directive expand)
  (input (a bool) (b bool))
  (output (d bool))
```

```
    (let d (and2 (inv1 a) b)))

(defcell level-2
  (directive expand)
  (input (a bool) (b bool))
  (output (e bool))
  (let e (or2 (level-1 a b) b)))
```

Here the cells, inv1, and2, and or2 are defined using the primitive constructs **delay** and **if**.

6.5.2. Synchronous Cells

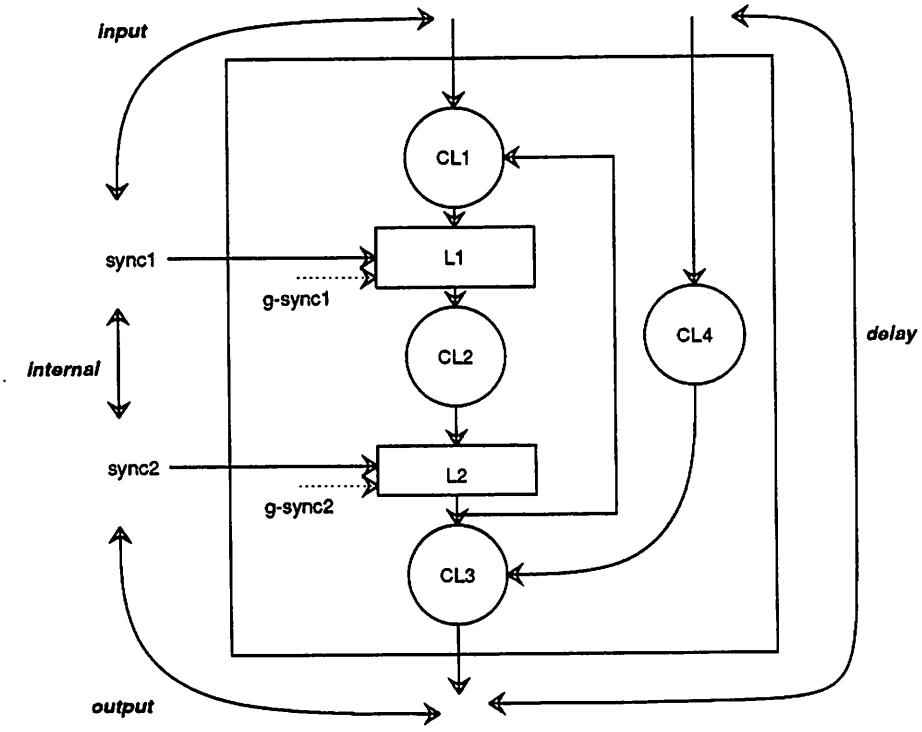
A synchronous cell contains one or more clocked storage elements or has at least one synchronous subcell. Each definition of the clocked storage elements require a synchronizing signal. Two mechanisms are provided in the verification system to introduce synchronizing signals: *local declaration* and *global declaration*. In a local declaration, synchronizing signals are defined in the formal parameter list. When a cell is defined with synchronizing signals as formal parameters, its timing constraints are not yet fully specified. The cell is to be used in different contexts of the synchronizing environment when the cell is instantiated later. The global declaration is explained in Section 6.4.2. In the case of a global declaration, the temporal distance between timepoints of the synchronizing signals are specified, and the timing constraints can be checked at this point.

6.5.2.1. Four Kinds of Timing Information

The timing information of a synchronous cell is abstracted as four kinds of information: *input timing information*, *output timing information*, *internal timing constraints*, and *delay information*.

Input timing information is a set of delay values from each input to each clocked storage element, where each clocked storage element is the first-leveled one when the net-

work is traversed in breadth-first manner from inputs. Figure 6.12 is an example to illustrate the meaning of each timing information. In this case, there is only one input which is fed into the first clocked storage element $L1$. Thus the input constraint is simply the delay value of the combinational logic block $CL1$.



CL's: combinational logic blocks
L's: clocked storage elements

Figure 6.12 Abstraction of timing behavior of a synchronous cell

Output timing information is similarly defined. It is a set of delay values to each output from each of the last clocked storage elements when traversed in breadth-first manner. In Figure 6.12, there is only one output and one last (Level 2 in this case) clocked storage element, $L2$, through the combinational logic block $CL3$. Hence the output timing information is just the delay value of $CL3$.

Internal timing constraints are derived from the delay values of the combinational logic blocks which are present in between each of the two consecutive clocked storage elements. Since the synchronizing signals are defined in two different ways, the internal timing constraints are classified into two cases. When the synchronizing signals are declared as formal parameters, the temporal distance between the two synchronizing signals is yet unknown and the delay values of the combinational logic blocks constrain the separation of the synchronizing signals. These constraints are saved with the cell definition and are checked if they are satisfied when the cell is instantiated later within a cell which uses a globally declared synchronizing signals. Conversely, when the synchronizing signals are globally declared, the separations of synchronizing signals are given and they are used to check for delays of the combinational blocks that are greater than the allowed periods. If there is no timing violation, the cell is defined without any internal timing constraints. Otherwise, the timing error is reported. In Figure 6.12, when the synchronizing signals are formal, the following two constraints are saved associated with the cell definition:

(*delay–of* CL2) < temporal distance from *sync* 1 to *sync* 2
(*delay–of* CL1) < temporal distance from *sync* 2 to *sync* 1

When the synchronizing signals are global, the above two constraints are checked based on the declarations of sync1 and sync2.

Delay information is mainly for combinational cells; however, even in a synchronous cell there may be direct paths from inputs to outputs without any clocked storage elements in the paths. As explained earlier, delay information is the set of worst delay values from each input to each output.

6.5.2.2. Instantiations of Synchronous Cells

When a synchronous cell instantiates other synchronous cells as one or more of its subcells, four possible cases can arise according to the four combinations of the mixed usage of synchronizing signal declarations. The four cases are illustrated in Figure 6.13. Figure 6.13 (a) illustrates the case when the upper level cell and the lower level cell are both defined with formal synchronizing signals. In this case, all the internal timing constraints of the subcell are abstracted untouched except the change of variable names. In Case (b) where the subcell is defined with formal synchronizing signals and the upper level cell uses global signals, the internal timing constraints of the subcell are checked and the output timing information is used when the timing constraints associated with the last synchronizing signal are checked. Note that when the subcell uses globally-defined synchronizing signals as illustrated in Case (c) of the figure, there are no internal timing constraints to be checked. Only the output timing information is used when the timing constraints associated with the last synchronizing signal are checked. When a cell is defined with formal synchronizing signals, the cell is to be used in different timing contexts and it is reasonable to assume that no synchronous subcells of a cell can use global synchronizing signals. Hence the last one illustrated as Case (d) in the figure is excluded from the possibilities.

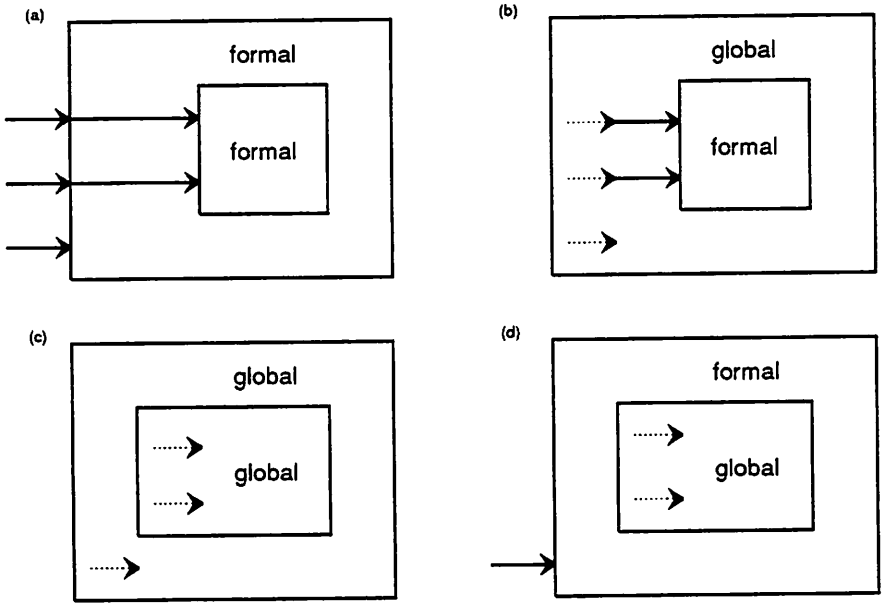


Figure 6.13 Four possible cases of a synchronous cell instantiation

6.5.3. Compositions of Timing Information

The abstraction process of timing information is based on a depth-first traversal of the network. From an output, until a primary input or a clocked storage element is encountered, the timing information of each net is propagated backward from the outputs to the inputs of each subcell. The timing information of each net is comprised of two different kinds of delay information: one is the delay from the current net to the primary output of the cell being abstracted, and the other is the delay from the current net to the closest clocked storage element which has already been traversed. Whenever a delay cell is encountered, the delay

values of the timing information of the current net are incremented by the amount of the delay value of the delay cell. When a subcell is met during the abstraction process of a cell, all the timing information of the subcell must be considered. For notational convenience, let the timing information of the current net be *current-timing-info*. In the following, each process involved in the abstraction of the four kinds of timing information is explained.

6.5.3.1. Compositions of Output Timing Information

First, the process involved in the output timing information associated with the subcell is explained. If *current-timing-info* is empty, which is illustrated as Case-1 in Figure 6.14, the output timing information of the subcell is used to update the output timing information of the current cell. Since there is no output timing information of the current cell yet, the output timing information of the subcell is copied to the slot of the current cell's timing information. On the other hand, if there is output timing information in the current cell, check first to see if it originates from the same synchronizing signal as in the subcell. If that is the case, choose the larger one as the current cell's output timing information. If they are originated from different synchronizing signals, append the subcell's output timing information to that of the current cell.

If *current-timing-info* is not empty, two different cases can occur. When *current-timing-info* is the delay information from the current net to the primary output of the current cell like the case of Case-2 in Figure 6.14, the output timing information of the current cell is updated with consideration to the delay value of the output timing information of the subcell.

When *current-timing-info* is the delay information from the current net to the closest clocked storage element as Case-3 in Figure 6.14, the output timing information of the subcell invokes an internal timing constraint of the current cell. When both synchronizing sig-

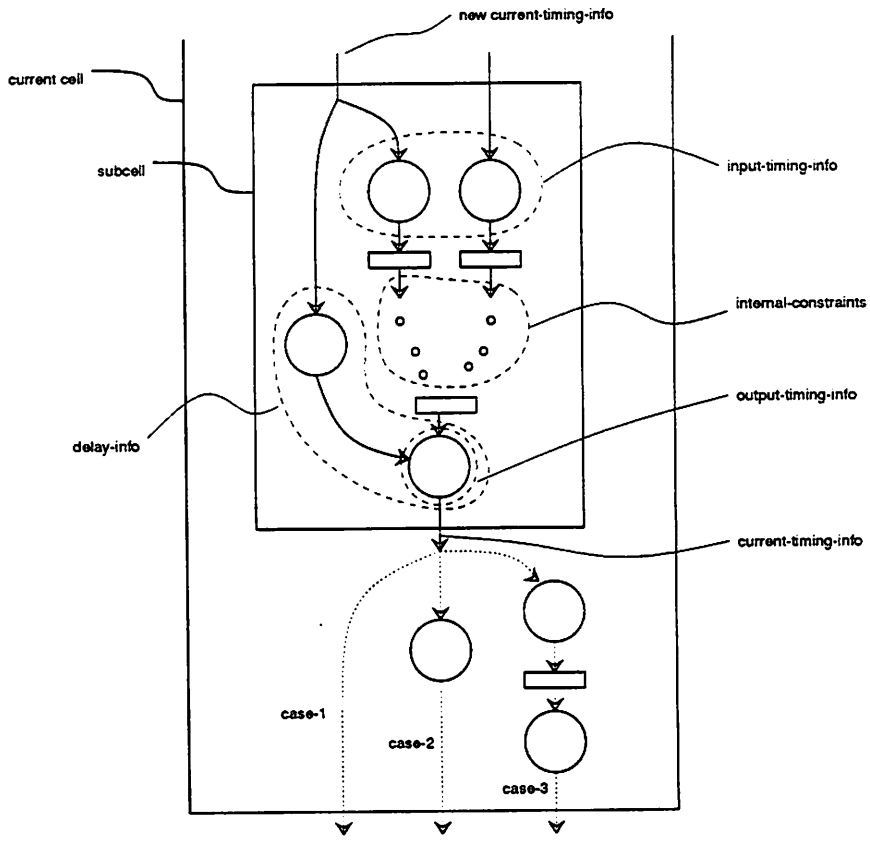


Figure 6.14 An example illustrating the timing information composition process

nals are global, a timing check is performed, and when both synchronizing signals are local, internal timing constraints are saved for later check. Because no coexistence of global and local synchronizing signals are allowed in a cell definition, when the subcell is formal and the current cell is global (Case (b) of Figure 6.13), the synchronizing signals of the subcell must have been instantiated into global signals. At this point whether the subcell uses a global

synchronizing signals and the current cell uses local synchronizing signals (which is forbidden) is also checked.

6.5.3.2. Compositions of Internal Timing Constraints

Resolving the internal timing constraints of a subcell is straight forward. When the subcell and the current cell use formal synchronizing signals, the internal timing constraints are just copied to those of the current cell. When the current cell uses global signals, the constraints can be checked, and hence either they are simply resolved or result in errors, depending on the delay and the separations of synchronizing signals.

6.5.3.3. Compositions of Delay and Input Timing Information

The output timing information and the internal timing constraints of the subcell are used for updating the output constraints and internal timing information of the current cell. Contrarily, the delay and input timing information of the subcell are used for updating the data associated with the new *current-timing-info* at the input net of the subcell. This process proceeds as follows.

For each input of the subcell, if there is a direct path from the input to the current net (which is the output net of the subcell), each delay value in *current-timing-info* is incremented by the delay of the direct path, then, the *current-timing-info* and input timing information of the subcell is merged. Conversely, when there is no direct path, which means every path from an input to an output contains at least one clocked storage element, the input timing information of the subcell becomes a new *current-timing-info* of the input net.

6.6. Examples and Results

In this section, the timing verification part of BEAVER is evaluated with several examples. The results of delay evaluation are compared to those obtained from a timing analyzer which does not consider the functional relations between signals. The library gates used in all the examples of this section are summarized in Table 6.2.

A simple example is shown in Figure 6.15. This example is included to illustrate how insensitizable paths influence the worst delay evaluation in a typical timing verifier. The output, D, is a function of both inputs; however, the dependence of the output, D, on the input, A, comes through the last AND gate rather than through Path-1 in the figure. Also note that Path-2 from the input, B, to the output is not sensitizable either. Without any functional information at each node, these false paths can not be detected. Hence the typical timing analyzer might report that the critical path is Path-1 from A to D with delay 5.8, where the actual critical path is Path-3 from B to D with delay 3.2.

To evaluate the performance of BEAVER, several examples have been selected. The experimental results are summarized in Table 6.3. In the table, the first column is the cell name described in the language of BEAVER. The BEAVER descriptions of these cells are included in Appendix A. The second column is the result of delay evaluation for an input-output pair of each circuit. When the value is represented by a single entity, it means the worst delay, and when the value is represented by a list, it illustrates the worst and the second worst. The last column is the cpu-time in seconds on a VAX 8800 machine. The cpu-times are obtained by a compiled code of VAX Common Lisp.

The first example, brand2, illustrated in Figure 6.15 is obtained from [192].

The examples r-c4d and l-a4d are four bit adders implemented by ripple-carry and carry-look-ahead, respectively. The critical paths in both cases were the path from the carry-

cell name	description	delay
inv1	inverter	1.0
and2	2-input and gate	1.6
nand2	2-input nand gate	1.6
or2	2-input or gate	1.6
nor2	2-input nor gate	1.6
xor2	2-input xor gate	1.6
and3	3-input and gate	2.0
nand3	3-input nand gate	2.0
or3	3-input or gate	2.0
nor3	3-input nor gate	2.0
and4	4-input and gate	2.8
nand4	4-input nand gate	2.8
or4	4-input or gate	2.8
nor4	4-input nor gate	2.8
and5	5-input and gate	4.0
nand5	5-input nand gate	4.0
or5	5-input or gate	4.0
nor5	5-input nor gate	4.0

Table 6.2 Library gates

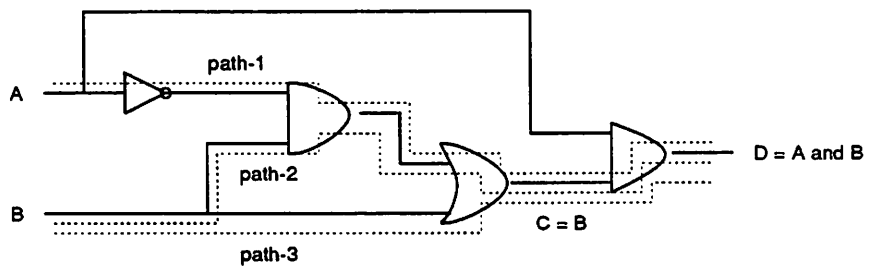


Figure 6.15 An example obtained from [192]

cell name	worst delay		cpu-time (sec)	
	BEAVER	without functionality	BEAVER	without functionality
brand2	(3.2 1.6)	(4.8 5.8)	0.02	0.01
r-c4d	12.8	25.6	5.79	0.36
l-a4d	8.4	9.6	4.16	0.20
fet-nop	(12.2 6.0)	(12.2 12.2)	4.9	0.33
fet-opt	(12.8 7.2)	(12.8 9.4)	11.8	0.32
opcode-nop	15	21.8	72.3	7.2
opcode-opt	13.6	24.8	76.7	8.4

Table 6.3 Results of comparison for delay estimates

in to the carry-out of each circuit. In r-c4d, the analyzer without functional information evaluates the worst delay as twice of the delay evaluated by BEAVER.

The examples fet-nop and fet-opt are obtained from the instruction-fetch controller of SPUR [104] CPU. They are implemented by running the BDS [194] description of the control logic using a logic optimizer MIS-II [87], which also performs the technology mapping. The same library cells of Table 6.2 are used for MIS-II. The postfix, '-opt', means that the circuit is implemented with optimization, and the postfix, '-nop', means that no optimization but only a technology mapping was performed in the MIS-II session. In both cases, the worst delays are the same, but the second worst delays are different from each other.

The final example pair of Table 6.3 was obtained from the instruction decode logic of SPUR CPU. The number of gates of opcode-opt circuit is 132 and that of opcode-nop circuit is 151. Interestingly enough, in this case the worst delay is quite different. Hence, without consideration of functional relationships, the delay evaluation would be too pessimistic.

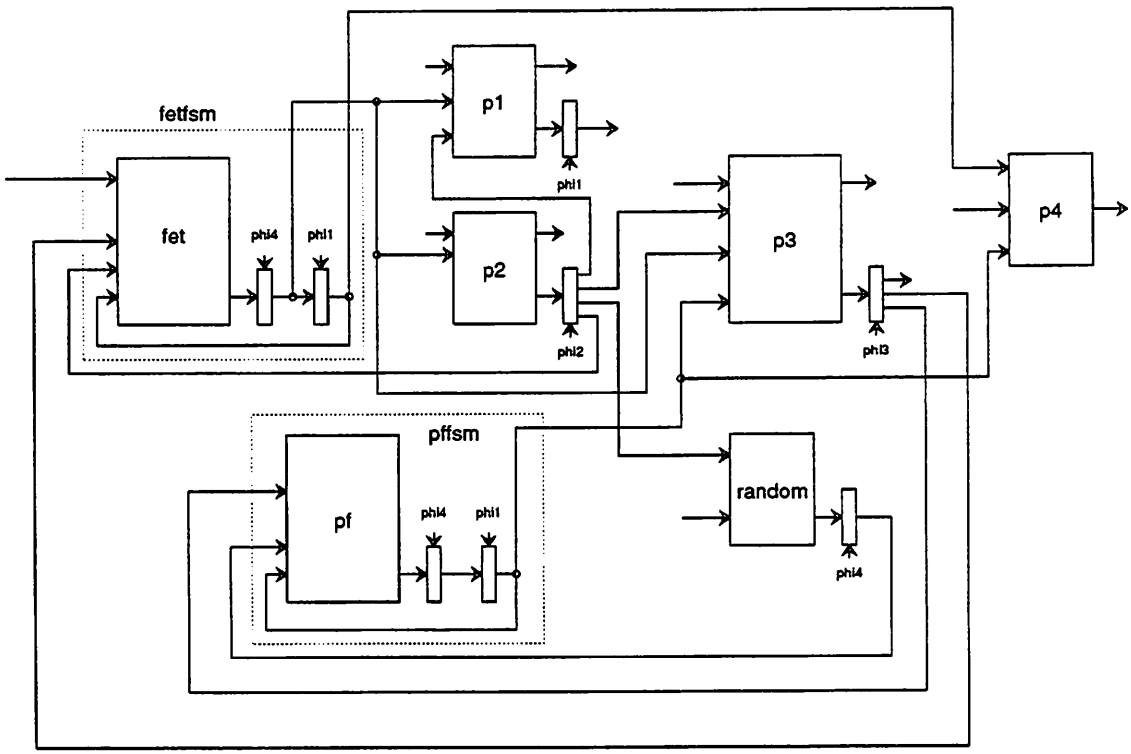


Figure 6.16 On-chip instruction cache controller in SPUR

Segment name	Clock separation
phi1 - phi2	0.0ns
phi2 - phi3	10.0ns
phi3 - phi4	8.2ns
phi4 - phi1	3.0ns
Total period	21.2ns

Table 6.4 Critical path delay estimates of the circuit in Figure 6.16

Finally, the usage of BEAVER as a timing consultant which determines the clock period and the separations of a multiphase clocking system is illustrated by an example shown in Figure 6.16. The circuit is an on-chip instruction cache controller in SPUR CPU which has been implemented by multilevel logic with 153 gates and 25 latches. The circuit contains two finite state machines and five combinational logic blocks with latches. In this example, all the clocks are declared as formal synchronizing signals and the abstracted internal-constraints are used to determine the clock separations and period. The cpu-time spent in this process was 89 seconds on a VAX 8800 machine with compiled codes of VAX Common Lisp. The results of the timing analysis are summarized in Table 6.4. The meaning of zero clock separation in the first row of the table is that there is no timing constraint between the two clock phases derived from the given circuit. The results of the other three cases and the sum of the four clock separation periods are illustrated.

6.7. Conclusions

In this chapter, a hierarchical timing verification subsystem has been presented. The abstraction mechanism based on a constraint propagation method has been developed. For the combinational subcells, the utilization of functionality provides a critical-path evaluation procedure which can eliminate the insensitizable paths. The experimental results illustrate that the symbolic approach for timing verification is efficient enough for the application to real designs.

CHAPTER 7

FORMAL FUNCTIONAL VERIFICATION

7.1. Introduction

In a formal verification system the decision about which proof mechanism to employ is affected by several considerations. First, the proof mechanism should be consistent with the formalism of the description language. For example, if the language is based on functional formalism, the proof mechanism appropriate for constraint-based language would not be a suitable choice. Second, the proof mechanism should be able to make the verification process as automatic as possible. This issue is closely related to the capability of the theorem prover of the system. Finally, the proof mechanism should be reasonably efficient in both space and time.

It would be ideal to employ a theorem prover based on the same formalism as the hardware description language while the theorem prover is powerful enough to handle any practical hardware verification problem automatically within reasonable memory space and cpu time. However, such a theorem prover is generally not obtainable. Finding a general proof procedure to verify the validity (or inconsistency) of a formula was considered long ago. It was first tried by Leibniz (1646 - 1716) and further revived by Peano around the turn of the century and by Hilbert's school in the 1920s [134]. It was not until Church (1936) [195] and Turing (1936) [196] that this was proved impossible. Church and Turing independently showed that there is no general decision procedure to check the validity of formulas of the first-order logic. However, there are proof procedures which can verify that a formula is

valid, if it is indeed valid. For invalid formulas, these procedures generally will never terminate [134].

The foundation of mechanical theorem proving was developed by Herbrand in 1930. It was impractical to apply until the invention of the digital computer. It was not until J. A. Robinson's landmark paper, [197] in 1965, together with the development of the resolution principle, that major steps were taken to achieve realistic computer-implemented theorem provers. Since 1965, many refinements of the resolution principle have been made [198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210]. These theorem proving techniques are utilized in many tools for specification and verification of programming languages. Recently, as formal verification has gained a lot of attention for an application to hardware verification as a viable alternative to simulation, several notable achievements have been reported [74, 82, 211]. However, for the formal approach to be practically utilized in the hardware verification field, more research work is needed.

In this chapter, the equivalence check BEAVER attempts to solve is described. The theorem prover employed in the verification system for the equivalence check of functionality is explained along with a description of other theorem provers. The formal verifier developed in this chapter deals with all the language constructs explained in Chapter 5. Definitions of enumeration types, structure types, and array types are formally manipulated without any human intervention. Whenever possible, the hierarchy is exploited. Inductive proof is provided for the verification of designs involving recursive cells. Since no distinction is made between behavior and structure, any combination of behavioral and structural descriptions can be verified. Finally, several verification examples are illustrated.

7.2. Equivalence Checking Problem

As mentioned in Chapter 2, the problem of functional verification in BEAVER is checking the equivalence of two descriptions. Since the term "equivalence" means several different things in different contexts of verification, here the meaning of equivalence needs clarification.

Generally, the correspondence between two machines may be a homomorphism, rather than exact equivalence (isomorphism). When two machines always produce the same outputs for applied inputs, they cannot be distinguished to an observer who can see only the inputs and outputs of the machines. In fact, these two machines exhibit the same external behavior. When one machine represents a specification and the other is an implementation, if the two machines are indistinguishable to the observer, the implementation is said to be *equivalent* to the specification.

The definition of equivalence in this chapter is further formalized as follows. A description is viewed as how input data are transformed to produce its output. Hence, if two descriptions given with the same input produce equivalent outputs, they are said to be equivalent. Two objects are said to be equivalent if they denote the same value in the selected value domain. Each object can be either a primitive constant or a cell denoting a value through several levels of functional applications. The value domain consists of primitive constants such as Boolean values of 'true' and 'false' and natural numbers including zero (0 1 2 $\cdots$). But as will be explained shortly, the value domain grows dynamically as types are defined.

When two descriptions denote different typed objects, the user needs to provide cells for type transformation. For example, consider the equivalence check of two cells, *cell1* and *cell2* as shown in Figure 7.1. If the types of input and output of *cell1* are different from those of *cell2*, cells *f1* and *f2* are provided to make the environments of the two descriptions

homogeneous. Now, the equivalence check between Y and Y' is performed for the original problem.

As mentioned in Chapter 5, when a cell contains a loop, there should be at least one clocked storage element in the loop. However, since the description has the flavor of data-flow language, two cells to be proved equivalent do not have to be isomorphic. Only one state mapping information is required per loop. For example, in Figure 7.2, even if the two cells have a different number of stages of clocked paths in the direct forward path and in the looped path, an equivalence check can be performed with only one corresponding state information per loop. For example, the mapping represented as a dotted line in Figure 7.2 will suffice for this case.

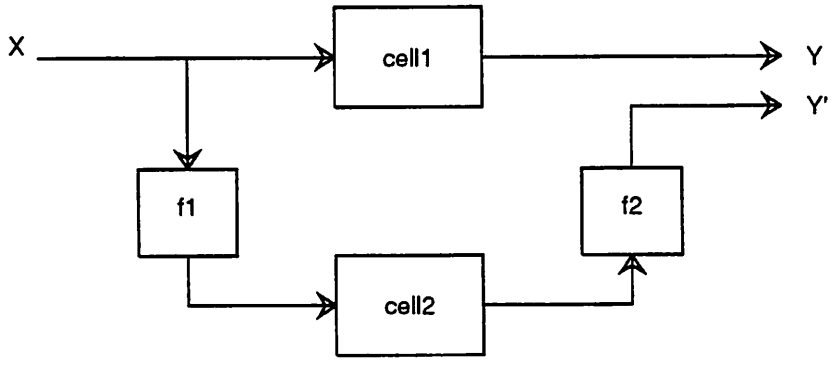


Figure 7.1 Equivalent descriptions

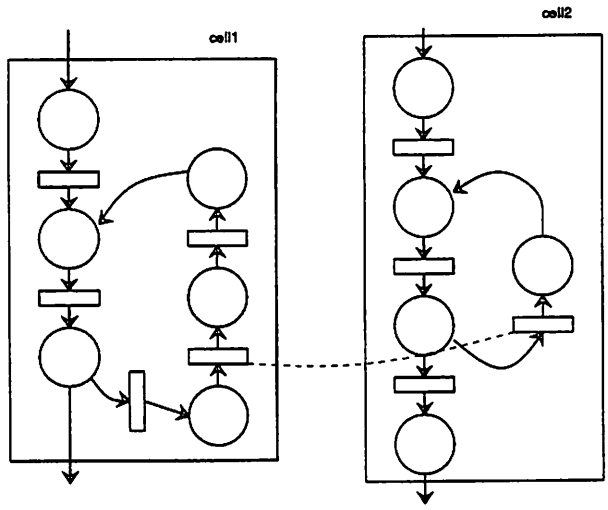


Figure 7.2 Non-isomorphic but equivalent cells

7.3. Theorem Prover

There have been many theorem provers (in the broad sense) available for the application to the formal hardware verification: the LCF proof checker [145], the HOL [84], the AURA [136], the Boyer-Moore theorem prover [141], the Veritas system [78], the REVE system [212], a temporal logic prover [151], and the Circal [75].

LCF (*L*ogic for *C*omputable *F*unctions) is essentially a typed λ-calculus together with fixed-point induction. All objects have a partial ordering and a bottom element and inference rules are encoded into *tactics*, thus supporting goal-directed (backward-chaining) proofs. The partial function principle creates the disadvantage of having to prove the totality of functions before they can be used. Additionally, proofs tend to become cluttered with extra cases due

to the bottom element. In general, since proof construction is a highly interactive activity, the user must be able to guide the proof. This offers a high degree of confidence in the correctness of the proof, but requires a lot of user expertise.

HOL is an extended version of LCF for higher-order logic. It thus shares many features with LCF, some of which are good and some of which are bad. In this system every behavior is defined as predicates and consequently non-executable. In order to simulate such behaviors, it is necessary to translate them into functions. From the point of proof complexity, the power of the logic means that proofs are more complex. The proofs are constructed in a deduction-like system, but need human guidance. Also, there is no generalized inductive proof mechanism. The LCF and the HOL systems may be more closely viewed, and more accurately, as environments for constructing proofs than automatic theorem provers.

The AURA (Argonne Automated Reasoning Assistant) system was employed in [65] to build a verification system. It is a resolution-based theorem prover that incorporates equality and a collection of inference rules. In the system, the rewrite principle is an inference rule called demodulation. Demodulation has two basic uses: canonicalization and simplification. AURA contains procedures which "weight" the arguments of the predicate EQUAL, and which weight all clauses used in a proof. An automatic selection process then allows such a simplification to be performed. The weighting process can be tailored to the specific application. Canonicalization makes use of a built-in lexical ordering mechanism that allows the user to manipulate expressions, transforming them into their most desirable form.

REVE contains an implementation of the Knuth-Bendix completion procedure [119], modulo associative-commutative operators and procedure supporting inductionless induction [213] or proof by consistency. Thus proving a theorem involves compiling a set of equations into a confluent set of rewrite rules and then using standard equational reasoning for theorems

involving ground (no variables) terms only, or invoking the inductionless induction process for theorems which contain variables. REVE is an automatic theorem prover in that once the proving process has been initiated, the user has very little influence over the course of the proof.

The Boyer-Moore theorem prover is an automatic theorem prover in the sense that once the proof of a theorem has been invoked, the user can no longer interfere. The Boyer-Moore logic is a quantifier-free first order logic with equality and the input language is a variant of pure Lisp. The inference rules consist of propositional calculus and associated inference rules, the principle of Noetherian induction and instantiation. The proof process consists of creating an environment containing the relevant definitions and invoking proof of various necessary lemmas until the target theorem can be proved. Discovering which lemmas are necessary is not trivial for most proofs. Most of all, the complicated but intelligent heuristics for inductive proof is the outstanding contribution of this theorem prover.

In BEAVER, for the verification of functionality many techniques and heuristics of the Boyer-Moore theorem prover have been utilized. The notable extensions and differences are as follows. First of all, the static type checking is employed compared to the dynamic type checking in the Boyer-Moore theorem prover. In the Boyer-Moore theorem prover, a type set is associated with every term and the type set is dynamically narrowed when more information becomes available about the term. This is due to the fact that the prover was aimed at program verification of pure Lisp style language where no static type information is available. However, in BEAVER, the language is strongly typed and the efficiency of the prover has been improved. Secondly, in BEAVER, multiple-valued return is allowed. In a hardware description, a cell often returns multiple values rather than a single value. Multiple values can be aggregated into a single list in Lisp style language. However, in describing hardware it is

awkward to provide a list structure for every such cell. Third, a proof mechanism for enumeration type and structure type have been added. Finally, for the description and verification of sequential machines, state variables are allowed in a description which may contain loops.

In the following, the approach of the theorem prover employed in BEAVER is described in more detail.

7.3.1. Value Domain

As previously mentioned, the value domain of BEAVER grows dynamically by type definitions. When an enumeration type is defined, each element of the enumerated list is added to the value domain. For example, when a type 'bit' is defined as follows,

(deftype bit **(enum** 0 1 *))

three constants are defined: bit-0, bit-1, and bit-*. Since the names of constant values are visible from anywhere, to avoid name collision the type name is prefixed to each constant name as illustrated above. Also when an array type is defined, the bottom object of the array is defined as a constant. For example, with the following definition of type 'bitv',

(deftype bitv **(array** bit))

a bottom object, 'bitv-btm', is added as a constant to the value domain.

7.3.2. Derived Cells and Axioms

Whenever a type is defined, appropriate cells and axioms are defined and added to the knowledge set of the verifier.

When an enumeration type is defined, no cell is defined but an axiom is added to the knowledge set. The axiom states that if an object is known to be the given type, then the value of the object must be one of the generated constants associated with the enumeration type definition. For example, when a type 'bit' is defined, as illustrated above, the following axiom is used for simplification during proof process.

```
(or (equal var bit-0)
    (equal var bit-1)
    (equal var bit-*))
```

When an array type is defined, three cells are defined and six axioms are added to the knowledge set. The first cell is used for constructing an array object from the two inputs: one for the element type and the other for the array type. When an array object is generated from nothing, the second object must be the bottom object of the type. The second and the third cells are for accessing the first element and accessing the rest of the array, respectively. The six axioms are used as lemmas during proof procedure. In the case of the definition of 'bitv' illustrated earlier, the following cells and axioms are automatically defined and added to the knowledge set, respectively.

cells: bitv-cons, bitv-first, and bitv-rest

axioms:
```
(bitv-cons (bitv-first x) (bitv-rest x)) => x

(bitv-first (bit-cons x1 x2)) => x1

(bitv-rest (bit-cons x1 x2)) => x2

(equal (bitv-cons x1 x2) (bitv-cons y1 y2))
=> (and (equal x1 y1) (equal x2 y2))

(not (equal (bitv-cons x1 x2) bitv-btm))

(< (size (bitv-rest x)) (size x))
```

The first three axioms are used for simplification. The fourth axiom tells the equivalence con-

dition of constructed objects, i.e., two constructed objects are the equivalent of when all components are equivalent. The fifth axiom tells that every bottom object is different from any constructed object. The last axiom is used in induction or in a recursive cell definition. This gives a hint to the prover about the well-founded ordering relations.

When a structure type with n slots is defined, $(n+1)$ cells are defined and $(n+2)$ axioms are added to the knowledge set. The first cell is used for constructing the structure object with n inputs, one for each slot. The other n cells are for accessing the corresponding slot. For example, when a structure type is defined by

(deftype bool3 **(struct** (<2> bool) (<1> bool) (<0> bool)))

the following cells and axioms are generated:

cells: bool3-cons, bool3-<2>, bool3-<1>, and bool3-<0>

axioms: (bool3-cons (bool3-<2> x) (bool3-<1> x) (bool3-<0> x)) => x

(bool3-<2> (bool3-cons x1 x2 x3)) => x1

(bool3-<1> (bool3-cons x1 x2 x3)) => x2

(bool3-<0> (bool3-cons x1 x2 x3)) => x3

(equal (bool3-cons x1 x2 x3) (bool3-cons y1 y2 y3))
=> (and (equal x1 y1) (equal x2 y2) (equal x3 y3))

The first four axioms are used for simplification. The last axiom tells the equivalence condition of constructed objects, i.e., two constructed objects are equivalent when all components are equivalent.

7.3.3. Major Proof Steps

It is one thing to describe a loosely connected set of heuristics that a human might use to discover proofs and quite a different thing to formulate them so that a machine can use

them to discover proofs. The order of applying the heuristics is very important. For some problems, one order of a sequence of heuristics might work very well, but for another set of problems, it might not.

The major steps involved in proving an equivalence are as follows. First, the equivalence conjecture is simplified by applying axioms, rewrite lemmas, and cell definitions and by converting the conjecture to a conjunction of **if**-free clauses. In most equivalence checkings involving non-recursive cells, the simplification process will be enough to prove the conjecture. When it does not, at least the complexity of the conjecture will be reduced. Following that, the simplified conjecture is reformulated by eliminating undesirable concepts using equality relations. Some terms that have played their role are generalized by introducing variables. Finally, irrelevant terms are eliminated from the conjecture to see if the conjecture can be further simplified. Since it is difficult to invent the right induction argument for anything but the simplest, strongest, conjecture available, and induction increases the size and complexity of the conjecture, induction is applied after all the simplification and rewrites are performed. In the remainder of this section, how a term is rewritten is explained. Then a brief introduction to the induction mechanism is presented. The detailed heuristics for the preparation of induction and the induction process itself are described in [141].

The rewrite process is the key in the simplification. Rewrite rules for a term can be classified into three groups. One is the built-in mechanism for primitive language constructs such as **if**, **equal**, **values**, and **nth**. The second group consists of the axioms from type definitions and lemmas of already proven facts. Finally, cell definitions are used for rewriting. A non-recursive cell definition is always expanded. Since the available lemmas have already been tried to prove the current clause, the substitution of the cell definition does not prevent the verifier from exploiting the hierarchy. When the cell is recursive and the expan-

sion contains more explicit values as arguments than the original, the expansion is substituted. Also, if the heuristic measure of symbolic complexity of some subset of the arguments of the expanded one is smaller than the complexity of that subset in the original instantiation, the expanded term is substituted.

After a rewrite, the rewritten term might have 'and', 'or', and 'if' forms. Each term is then transformed into a clausal form. A clause is a disjunction of literals, where a literal is an atomic form or its negation. It is clear how a form with logical 'and' or 'or' is transformed into a clausal form. Only a form with 'if' needs explanation. In the following example, first a cell-name is propagated into the if-form, then the if-form is transformed into its equivalent clausal form.

```
(cell-name ··· (if p q r) ··· )
=> (if p (cell-name ··· q ···) (cell-name ··· r ···))
=> {(not p), (f ··· q ···)} and {p, (f ··· r ···)}
```

Then each of the literals is rewritten with its corresponding environment of the assumptions. If any of the literals in a clause turns into 'true' by rewrite, the clause is removed from the clause-set. And if the clause-set becomes empty, the goal is true. If a literal turns into 'false', the literal is removed from the clause and if a clause becomes empty, the goal becomes false. After the entire simplification process, if there is still any clause to be proved, induction is tried. Using induction heuristics, invent a candidate induction scheme, and generate a conjunction of new clauses to prove (the base case and the induction steps). Then each of the new clauses is tried by starting from the simplification process again. Descendants of these clauses may also eventually be conjoined into the original clause-set. If the proof is successful, then eventually no clause will be in the clause-set.

The generation of inductive formulas relies on the similarity of recursion and induction. Let's consider a simple recursive definition of a cell 'plus'.

```
(defcell plus
  (input (x1 nat) (x2 nat))
  (output (y nat))
  (let y (if (equal x1 0) x2 (1+ (plus (1- x1) x2)))))
```

During the internalization process of a recursive cell, the conditions of the path which lead to the recursive call to the original cell are extracted. For the above cell, the extracted information for induction is as follows:

tests:	(not (equal x1 0))
recursive calls:	(plus (1- x1) x2)

Using the above information, the following two formulas are created by the process of inductive formula generation.

(*pred* (plus x1 x2)) =>

base step:	(equal x1 0) implies (*pred* (plus x1 x2))
induction step:	(not (equal x1 0)) and (*pred* (plus (1- x1) x2)) implies (*pred* (plus x1 x2))

Here, *pred* is a predicate which refers to the cell, 'plus'. In general, there may be many possible pairs that can be used as base step and induction step formulae. After expanding each possible induction scheme, the "best" one is picked up according to various heuristics. For the detailed heuristics, refer to [141].

7.4. Examples and Results

In this section, the functional verification part of BEAVER is applied to several groups of examples. All the examples are run on a VAX 8800 machine with compiled VAX Common Lisp codes.

The examples of the first group are the gate level descriptions of adders. Their descriptions are included in Appendix A. Table 7.1 summarizes the verification results. The first

column is the names of the pair of cells under equivalence check and the second column tells if the verification result was positive or negative. The last column is the cpu-time spent for proving the equivalence or non-equivalence. In the table, cells whose names start with 'r-c' are implemented by ripple-carry and those with 'l-a' are implemented by carry-look-ahead. The last example is the verification of two four-bit adders, one of which is intentionally described incorrectly. For this group of examples, all the equivalence checks have been performed by flattening the descriptions down to Boolean level. Note that the cpu-time increases almost by a factor of four for each additional one bit. Basically, the equivalence check of this group is a Boolean equivalence check; a theorem proving approach would be less efficient compared to other logic verification approaches [101, 132].

However, in general, behavioral descriptions must not be restricted to Boolean level in order to allow the abstraction of data object. For the formal proof of design correctness involving the abstraction, the theorem proving approach is required. The next example addresses this point. Consider an implementation and a specification of an ALU circuit. The logic is decomposed into two parts: data manipulation and operation decode part. The operation is encoded as two bit and four possible operations are defined: ADD, AND,

equivalence check	proof result	cpu-time (sec)
r-c2d ≡ l-a2d	positive	1.67
r-c3d ≡ l-a3d	positive	9.46
r-c4d ≡ l-a4d	positive	43.69
r-c4d ≡ wrong4d	negative	29.61

Table 7.1 Verification results of adder examples

COMPLEMENTARY, and PASS (no operation). The data inputs are a<0>, b<0>, and carry-in and the outputs are sum and carry-out. Figure 7.3 (a) illustrates the data and decode logic of one bit-slice. Both the data manipulation and decode logic are implemented from a BDS description into a netlist of the library gates listed in Table 6.2. The BEAVER description is given in Appendix A as cells named, 'alubit' and 'op-decode'. The implementation of a two bit ALU is shown in Figure 7.3 (b). The BEAVER description of the specification is as

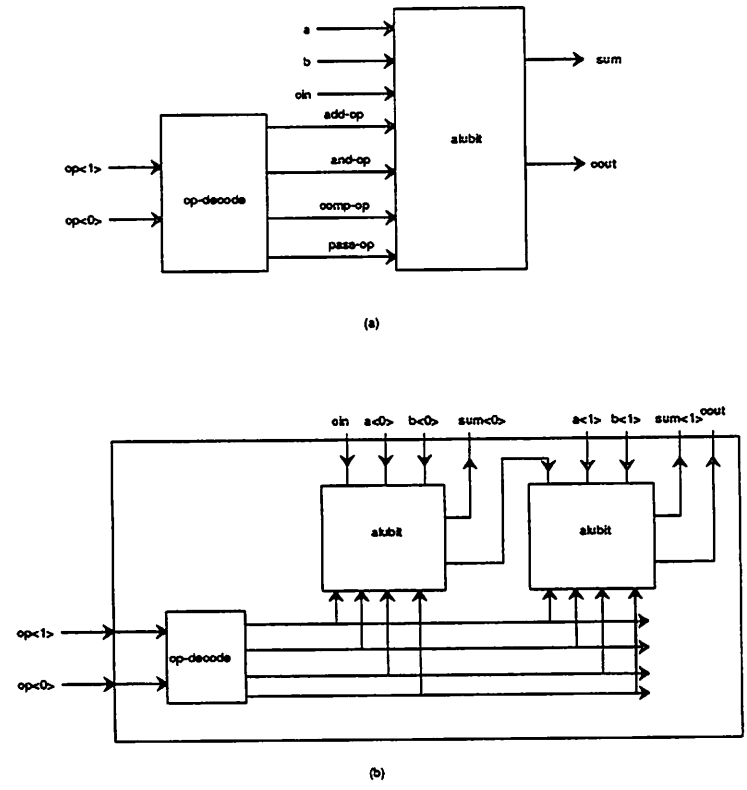


Figure 7.3 ALU bit-slice and 2-bit ALU

follows:

```
(deftype bool2 (struct (<1> bool) (<0> bool)))

(defcell nat-to-bool2
 (input (n nat))
 (output (y bool2))
 (local (tmp nat))
 (let tmp (if (< n 4) n (- n 4)))
 (let y (case tmp
            (0 (bool2-cons false false))
            (1 (bool2-cons false true))
            (2 (bool2-cons true false))
            (3 (bool2-cons true true)))))

(defcell bool2-to-nat
 (input (x bool2))
 (output (y nat))
 (local (b1 bool) (b0 bool))
 (let b1 (bool2-<1> x))
 (let b0 (bool2-<0> x))
 (let y (case (b1 b0)
            ((false false) 0)
            ((false true) 1)
            ((true false) 2)
            ((true true) 3))))

(defcell alu2bit.spec
 (input (a<1> bool) (a<0> bool))
 (input (b<1> bool) (b<0> bool))
 (input (c<0> bool) (op<1> bool) (op<0> bool))
 (output (s<1> bool) (s<0> bool) (c<2> bool))
 (local (tmp1 nat) (tmp2 bool2))
 (let tmp1 (+ (bool2-to-nat (bool2-cons a<1> a<0>))
         (bool2-to-nat (bool2-cons b<1> b<0>))
            (if c<0> 1 0)))
 (let c<2> (if (< tmp1 4) false true))
 (let tmp2 (nat-to-bool2 tmp1))
 (let (s<1> s<0>)
    (case (op<1> op<0>)
            ((false false) (values a<1> a<0>))
            ((false true) (values (not a<1>) (not a<0>)))
            ((true false) (values (and a<1> b<1>) (and a<0> b<0>)))
            ((true true) (values (bool2-<1> tmp2)
                                 (bool2-<0> tmp2))))))
```

As explained earlier in this chapter, when the type bool2 is defined as above, three cells are

automatically defined: bool2-cons, bool2-<1>, and bool2-<0>. These cells with the two auxiliary cells, nat-to-bool2 and bool2-to-nat, are used in the specification cell, alu2bit.spec. The specification is given in terms of '+' which is not directly related with Boolean values. The '+' operation is a higher abstraction than the Boolean operation. The equivalence check between the specification and the implementation given as a pure netlist of the library gates could be performed in 172 seconds.

The next example is a sequential circuit which is extracted from the instruction prefetch control unit in SPUR CPU. The circuit diagram is shown in Figure 7.4. The BEAVER description of the implementation and the specification are given in Appendix A. In the equivalence check of sequential cells, current states are considered as inputs. The next-state equations are derived from the descriptions, and the equivalence checks of these equations are also performed. The cpu-time spent for the example was 13.3 seconds.

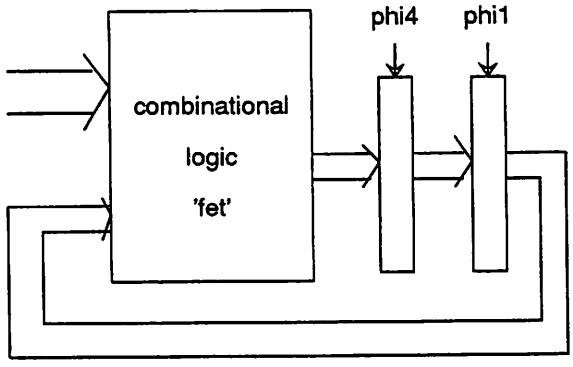


Figure 7.4 A sequential circuit example

The last example illustrates the inductive proof involving recursive cells. Consider the following three equivalent definitions of '+' operation.

```
(defcell plus1
 (input (x1 nat) (x2 nat))
 (output (y1 nat))
 (let y1 (if (equal x1 0)
             x2
         (1+ (plus1 (1- x1) x2)))))

(defcell plus2
 (input (in1 nat) (in2 nat))
 (output (out1 nat))
 (let out1 (if (not (equal in1 0) )
         (1+ (plus2 (1- in1) in2))
            in2)))

(defcell plus3
 (input (in1 nat) (in2 nat))
 (output (out1 nat))
 (let out1 (if (not (equal in2 0) )
         (1+ (plus3 in1 (1- in2)))
            in1)))
```

Cell 'plus2' is just a switched version of 'plus1' between true- and false-term in the if-form. Cell 'plus3' uses the second argument as recursive variable where 'plus1' and 'plus2' use the first argument. The cpu-time spent for the three equivalence checks are presented in Table 7.2.

equivalence check	cpu-time (sec)
plus1 ≡ plus2	0.66
plus2 ≡ plus3	2.4
plus3 ≡ plus1	0.73

Table 7.2 Inductive proof examples

7.5. Conclusions

In this chapter, the functional verification part of the verification system has been presented. The verification problem is defined as an equivalence checking between two descriptions of BEAVER cells. The verification is based on theorem proving techniques. Hence, when the result of a verification is positive, the two descriptions are guaranteed to be equivalent for all the possible inputs. Theorem provers that have been applied to hardware verification were also reviewed, the major proof steps involved in theorem proving were explained, and several verification examples were presented to demonstrate the aspects of behavioral verification of BEAVER.

CHAPTER 8

CONCLUSIONS

Until all the design processes are completely automated, verification of designs continues to be an important problem. In this dissertation, the verification problem of digital VLSI designs has been addressed. The verification problem is classified into two categories, correctness checking of a particular finite-state machine and the other is equivalence checking of two machines. Then for each problem, an efficient verification system has been developed.

For the correctness checking of finite-state machines, the specification is given as a state-transition table and the implementation is given as a net-list of gates and latches. The verifier checks to see if the implementation satisfies the specification. When the implementation is incorrect, an input sequence that distinguishes the implementation machine from the specification machine is provided to help the user locate errors. The experimental results show that the method can be applied to fairly large systems.

For the equivalence checking of behavioral descriptions, a formal technique which uses a theorem-proving method has been employed. When verification is performed using a formal technique, a complete verification can be achieved. Since formal techniques have been investigated earlier and more intensively in software verification than in hardware verification, most formal hardware verification techniques have stemmed from program verification techniques. The previous work on formal program verification and hardware verification have been reviewed. The study of previous work on formal hardware verification

illustrates that the choice of a formalism involves a compromise between expressive power and ease of automatic synthesis/verification. In a simple and restricted formalism, it is hard to specify complex devices simply and concisely. On the other hand, in a powerful formalism, it is difficult to automate the verification process. In this dissertation, a functional formalism has been chosen.

A hardware description language was presented which can deal with both timing and functionality in a single paradigm. The semantics of the language is based on functional (denotational) semantics. Type definition mechanisms and macros are provided and recursive definitions are supported.

The behavioral verification system, BEAVER, handles both timing and functionality. The static timing verification task is performed when a cell definition is processed. For a hierarchical timing verification, an abstraction mechanism for timing information based on a constraint propagation has been developed. For combinational cells, the availability of functional relations between signals allows for elimination of the static-insensitizable-path problem. The experimental results also illustrate that the symbolic approach to timing verification is efficient enough to apply to real designs.

Finally, the equivalence checking problem implemented by BEAVER has been defined and functional verification based on theorem proving techniques has been presented. The verifier can handle automatically the definition of abstract types. Whenever possible, hierarchy is exploited and the inductive proof approach provides a way of verifying designs involving recursion.

The examples described in this dissertation are listed in Appendix A. In addition, both the BEAVER programs and the examples may be obtained in machine-readable form from Industrial Liaison Program, Department of Electrical Engineering and Computer Science,

University of California, Berkeley, CA 94720.

Future work remains to be carried out in the following areas. First, the two verification systems for correctness checking and equivalence checking can be unified into a single system. Then the behavioral specification of a finite-state machine and its implementation in random logic can be more efficiently verified.

Second, for sequential circuits, it is desirable to verify two descriptions which may use a different number of iterations on a looped structure. For example, consider two different implementations of three-input adders which yields the sum of the three inputs: one is implemented with two two-input adders and no register, the other is implemented with one two-input adder and a register to hold the temporary result of the sum of the first two operands. In general, two sequential circuits might have a different number of clock cycles to produce equivalent results. This verification problem involves a more sophisticated treatment of time and remains unsolved to date.

Finally, one direct extension of BEAVER is to provide it with the capability to handle "don't-care" conditions. The functional verifier checks equivalence, however sometimes implication is actually the problem. Here, the meaning of "implication" should be distinguished from logical implication; in logical implication, false implies everything, which sometimes cause problems. In a functionality check, when the specification represents a don't-care condition, the implementation may result in any value as its output. This aspect can be handled by differentiating a specification from its implementation, and by providing an implication relation of values (e.g. a don't-care value implies any specific values) instead of the equivalence relation of values as in the current implementation.

REFERENCES

1. W. M. VanCleemput, "A hierarchical language for the structured description of digital systems," *Proc. 14th Design Automation Conference*, June 1977.

2. R. H. Katz, "A database approach for managing VLSI design data," *Proc. 19th Design Automation Conference*, 1982.

3. R. Hitchcock, "Timing verification and the timing analysis program," *Proc. of the 19th Design Automation Conference*, pp. 594-604, June 1982.

4. J. Ousterhout, "Crystal: A timing analyzer for NMOS VLSI circuits," *Proc. of the Third Caltech VLSI Conference*, pp. 57-59, 1983.

5. N. Jouppi, "Timing analysis and performance improvement of MOS VLSI designs," *IEEE Trans. on CAD of ICAS*, vol. CAD-6, no. 4, pp. 650-665, July 1987.

6. H. K. Gummel, "A self-consistent iterative scheme for one-dimensional steady-state transistor calculations," *IEEE Trans. Electron Devices*, vol. ED-11, pp. 455-465, Oct. 1964.

7. C. N. Gwyn , D. L. Sharfetter, and J. L. Wirth, "The analysis of radiation effects in semiconductor junction devices," *IEEE Trans. Nuclear Sci.*, vol. NS-14, pp. 153-169, Dec. 1967.

8. Branin, "D-C and transient analysis of networks using a digital computer," *Proc. Design Automation Conference*, 1964.

9. F. F. Kuo, "Network analysis by digital computer," *Proc. IEEE*, vol. 54, pp. 821-835, June 1966.

10. W. J. McCalla and W. G. Howard, "BIAS-3 - A program for the nonlinear dc analysis of bipolar transistor circuits," *Digest Tech. Papers, IEEE Int. Solid State Circuits Conf.*, pp. 82-83, (Philadelphia, PA), Feb. 1970.

11. L. Nagel and R. A. Rohrer, "Computer analysis of nonlinear circuits, excluding radiation (CANCER)," *IEEE J. Solid State Circuits*, vol. SC-6, pp. 166-182, Aug. 1971.

12. T. E. Idleman , F. S. Jenkins, W. J. McCalla, and D. O. Pederson, "SLIC - A simulator for linear integrated circuits," *IEEE J. Solid State Circuits*, vol. SC-6 , pp. 192-204, Aug. 1971.

13. F. S. Jenkins and S. P. Fan, "TIME - A nonlinear dc and time-domain circuit simulation program," *IEEE J. Solid State Circuits*, vol. SC-6 , pp. 188-192, Aug. 1971.

14. W. T. Weeks, A. J. Jimenez, G. W. Mahoney, D. Mehta, H. Quassemzadeh, and T. R. Scott, "Algorithms for ASTAP - A network analysis program ," *IEEE Trans. Circuit Theory*, vol. CT-20, pp. 628-634, Nov. 1973.

15. L. W. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," *UCB/ERL M75/520*, University of California, Berkeley, May 1975.

16. L. Rosenberg and C. Benbassat, "Critic: An integrated circuit design rule checking program," *Proc. 11th Design Automation Workshop*, pp. 14-18, (Denver, CO) , June 1974.

17. H. S. Baird, "A survey of computer aids for IC mask artwork verification," *Proc. IEEE Int. Symp. on Circuit and Systems*, (Phoenix, AZ), Apr. 1977.

18. R. M. Allgair and D. S. Evans, "A comprehensive approach to a connectivity audit, or a fruitful comparison of apples and oranges," *Proc. 14th Design Automation Conf.*, pp. 312-321, (New Orleanchs, LA), June 1977.

19. J. Le Charpentier, "Computer aided synthesis of an IC electrical diagram from mask data," *Digest Tech. Papers, IEEE Int. Solid State Circuits Conf.*, pp. 84-85, (Philadelphia, PA), Feb. 1975.

20. B. T. Preas , B. W. Lindsay, and C. W. Gwyn, "Automatic circuit analysis based on mask information," *Proc. 13th Design Automation Conf.*, pp. 309-317, (San Francisco, CA), June 1976.

21. I. Dobes and R. Byrd, "The automatic recognition of silicon gate transistor geometries: An LSI design aid program," *Proc. 13th Design Automation Conf.*, pp. 414-420, (San Francisco, CA), June 1976.

22. L. Scheffer and R. Apti, "LSI design verification using topology extraction," *Proc. 12th Asilomar Conf. Circuits, Systems, and Computers*, pp. 149-153, (Asilomar, CA), Nov. 1978.

23. A. R. Newton, "Computer-aided design of VLSI circuit," *Proc. IEEE*, vol. 69, Oct. 1981.

24. D. D. Hill and W. M. Van Cleemput, "SABLE: a tool for generating structural, multi-level simulation," *Proc. 16th Design Automation Conf.*, pp. 403-405, San Diego, CA, June 1979.

25. S. Y. H. Su, "A survey of computer hardware description languages in the U.S.A.," *IEEE Computer*, vol. 7, no. 12, pp. 45-51, Dec. 1974.

26. M. R. Barbacci, "Instruction Set Processor Specification (ISPS): The notation and its applications," *IEEE Trans. Computers*, vol. C-30, pp. 24-40, Jan. 1981.

27. J. D. Morison, N. E. Peeling, and T. L. Thorp, "ELLA: Hardware description or specification?," *IEEE Int. Conf. on Computer-Aided Design*, pp. 54-56, Nov. 1984.

28. *VHDL Language Reference Manual*, May 1987.

29. O.-J. Dahl, E. W. Dijkstra, and A. R. Hoare, *Structured Programming*, Academic Press, London and New York , 1972.

30. Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Trans. on Programming Language and Systems*, vol. 2, pp. 90-121, 1980.

32. J. Southard, "MacPitts: An approach to silicon compilation," *IEEE Computer*, 1983.

33. R. L. Blackburn and D. E. Thomas, "Linking the behavioral and structural domains of representation in a synthesis system," *Proc. of 22nd Design Automation Conference*, pp. 374-380, 1985.

34. A. R. Newton, "Timing, logic, and mixed-mode simulation for large MOS integrated circuits," in *Computer Design Aids for VLSI Circuits*, P. Antognetti, D. Pederson and H. De Man, eds., Nato Advanced Study Series, Sijthoff & Noordhoff, Rockville, Maryland, 1981.

35. J. E. Kleckner, "Advanced mixed-mode simulation techniques," *UCB/ERL M84/48*, University of California, Berkeley, June 1984.

36. R. A. Saleh, J. E. Kleckner, and A. Richard Newton, "Iterated timing analysis in SPLICE1," *Digest 1983 Int. Conf. on CAD, IEEE*, Santa Clara, CA, Sept 1983.

37. R. A. Saleh, "Iterated timing analysis and SPLICE1," *UCB/ERL M84/2*, University of California, Berkeley, Jan 1984.

38. J. White and A. Sangiovanni-Vincentelli, "RELAX2: A new waveform relaxation approach for the analysis of LSI MOS circuits," *Proc. 1983 Int. Symp on Circ. and Sys.*, Newport Beach, May 1983.

39. J. White, "RELAX2 : A modified waveform relaxation approach to the simulation of MOS digital circuits," *UCB/ERL M84/21*, University of California, Berkeley, 22 Feb 1984.

40. J. White, F. Odeh, A. Sangiovanni-Vincentelli, and A. Ruehli, *Waveform Relaxation: Theory and practice.*

41. M. A. Breuer , "Techniques for the simulation of computer logic," *Commun. Ass. Comput. Mach.*, pp. 443-446, July 1964.

42. E. Ulrich, "Time sequenced logical simulation based on circuit delay and selective-tracing of active network path," *Proc. ACM Nat. Conf.*, pp. 437-448, 1965.

43. E Ulrich, "Exclusive simulation of activity in digital networks," *Commun. Ass. Comput. Mach.*, pp. 102-110, Feb. 1969.

44. S. A. Szygenda, "TEGAS-Anatomy of a general purpose test generation and simulation at the gate and functional level," *Proc. 9th Design Automation Conf.*, pp. 116-127, June 1972.

45. E. G. Ulrich and T. Baker, "The concurrent simulation of nearly identical digital networks," *Proc. 10th Design Automation Conf.*, pp. 145-150, June 1973.

46. S. A. Szygenda and E. W. Thompson, "Digital logic simulation in a time-based table-driven environment: part 1, design verification," *IEEE Computer*, pp. 24-36, Mar. 1975.

47. H. E. Krohn, "Design verification of large scientific computers," *Proc. 14th Design Automation Conf.*, pp. 354-361, June 1977.

48. R. E. Bryant , "MOSSIM: A switch-level simulator for MOS LSI," *Proc. 18th Design Automation Conf.*, pp. 786-790, Jul. 1977.

49. R. E. Bryant, "An algorithm for MOS logic simulation," *Lambda Magazine*, pp. 46-53, 1980.

50. Y. Kim, S. Hwang, and A. R. Newton, "Electrical-Logic simulation and its application," *IEEE Trans. on CAD of ICAS*, Jan. 1989.

51. H. De Man , "Mixed mode simulation for MOS VLSI: Why, where and how?," *Proc. IEEE Int. Symp. Circuits System*, pp. 699-701, Rome, Italy, May 1982.

52. W. M. G. Van Bokhoven, "Mixed-level and mixed-mode simulation by a piecewise-linear approach," *Proc. IEEE Int. Symp. Circuits System*, pp. 1256-1258, Rome, Italy, May 1982.

53. A. R. Newton, "Techniques for the simulation of large-scale integrated circuits," *IEEE Trans. Circuits and Systems* , vol. CAS-26, pp. 741-749, Sep. 1979.

54. V. D. Agrawal, A. K. Bose, P. Kozak, H. N. Nham, and E. Pascal-Skewes, "A mixed-mode simulator," *Proc. 17th Design Automation Conf.*, pp. 618-625, Minneapolis, MN, June 1980.

55. T. Sasaki, A. Yamada, S. Kato, T. Nakazawa, K. Tomita, and N. Nomizu, "MIXS: a mixed level simulator for large digital system logic verification," *Proc. 17th Design Automation Conf.*, pp. 626-633, Minneapolis, MN, June 1980.

56. H. N. Nham and A. K. Bose, "A multiple delay simulator for MOS LSI circuits," *Proc. Design Automation Conf.*, pp. 610-611, Minneapolis, MN, June 1980.

57. P. H. Reynaert , H. De Man , G. Arnout, and J. Cornelissen, "DIANA: a mixed-mode simulator with a hardware description language for hierarchical design of VLSI," *Proc. IEEE Int. Conf. Circuits and Computers*, pp. 356-360, Port Chester, NY, Oct. 1980.

58. O. H. Ibarra and S. Sahni, "Polynomially complete fault detection problems," *IEEE Trans. Computers*, vol. C-24, pp. 242-250, Mar. 1976.

59. W. C. Carter, W. H. Joyner, Jr., and D. Brand, "Symbolic simulation for correct machine design," *Proc. 16th Design Automation Conference*, pp. 280-286, 1979.

60. J. A. Darringer, "The application of program verification techniques to hardware verification," *Proc. 16th Design Automation Conference*, pp. 375-381, 1979.

61. T. J. Wagner, "Verification of hardware designs through symbolic manipulation," *Proc. Symposium on Design Automation and Microprocessors*, pp. 50-53, IEEE and ACM, Palo Alto, CA, Feb. 1977.

62. M. C. McFarland, "On proving the correctness of optimizing transformations in a digital design automation system," *Proc. 18th Design Automation Conference*, pp. 90-97, 1981.

63. V. Pitchumani and E. P. Stabler, "A formal method for computer design verification," *Proc. 19th Design Automation Conference*, pp. 809-814, 1982.

64. L. H. Hanes, "Logic design verification using static analysis," Ph.D. Dissertation, Dept. of Electrical Engineering, Univ. of Illinois at Urbana-Champaign, IL, 1983.

65. A. S. Wojcik, "Formal design verification of digital systems," *Proc. 20th Design Automation Conference*, pp. 228-234, 1983.

66. W. E. Cory, "Verification of hardware design correctness: symbolic execution techniques and criteria for consistency," Ph.D. Dissertation , Electrical Engineering Department, Stanford Univ., 1985.

67. C. V. Bochman, "Hardware specification with temporal logic: an example," *IEEE Trans. on Computer*, vol. C-31, no. 3, pp. 223-231, Mar. 1982.

68. B. Moszkowski, "Reasoning about digital circuits," Ph.D Dissertation, Dept. of Computer Science, Stanford University, 1983.

69. E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications: a practical approach," *10th ACM Symposium on Principles of Programming Languages*, Austin, TX, 1983.

70. D. L. Dill and E. M. Clarke, "Automatic verification of asynchronous circuits using temporal logic," *Proc. Chapel Hill Conference on VLSI*, Computer Science Press,

1985.

71. B. Moszkowski, "A temporal logic for multilevel reasoning about hardware," *IEEE Computer*, pp. 10-19, Feb. 1985.

72. M. J. C. Gordon, "Proving a computer correct," *Technical Report No. 42* , Computer Laboratory, University of Cambridge, 1983.

73. M. J. C. Gordon, "LCF-LSM: a system for specifying and verifying hardware," *Technical Report No. 41, Computer Laboratory, University of Cambridge*, 1983.

74. H. G. Barrow, "VERIFY: a program for proving correctness of digital hardware designs," *Artificial Intelligence*, vol. 24, pp. 437-491, 1984.

75. G. Milne, "CIRCAL and the representation of communication, concurrency and time," *ACM Trans. Programming Languages and Systems*, vol. 7, no. 2, pp. 270-298, Apr. 1985.

76. M. J. C. Gordon, "Why higher-order logic is a good formalism for specifying and verifying hardware," *Technical Report No. 77*, University of Cambridge, Computer Laboratory, Sep. 1985.

77. S. M. German and Y. Wang, "Formal verification of parameterized hardware designs," *IEEE Conference on Computer Design*, pp. 549-552, 1985.

78. F. K. Hanna and N. Daeche, "Specification and verification using higher-order logic," in *Formal aspects of VLSI design*, ed. G. Milne and P. A. Subrahmanyam, North-Holland, 1986.

79. S. Bapat and G. Venkatesh, "Reasoning about digital systems using temporal logic," *Proc. 23rd Design Automation Conference*, pp. 215-219, 1986.

80. R. E. Bryant, "Symbolic verification of MOS circuits," *VLSI Conference, Chapel Hill*, pp. 419-438, 1985.

81. D. Weise, "Functional verification of MOS circuits," *Proc. 24th Design Automation Conference*, pp. 265-270, 1987.

82. W. A. Hunt, "Mechanical verification of a microprocessor design," in *From HDL descriptions to guaranteed correct circuit designs*, ed. D. Borrione, pp. 89-129, 1987.

83. D. R. Musser, P. Narendran, and W. J. Premerlani, "BIDS: a method for specifying and verifying bidirectional hardware devices," in *VLSI specification, verification and synthesis*, ed. G. Birtwistle and P. A. Subrahmanyam, Kluwer Academic Publishers, 1988.

84. M. J. C. Gordon, "HOL: A proof generating system for higher-order logic," in *VLSI specification, verification and synthesis*, ed. G. Birtwistle and P. A. Subrahmanyam, pp. 73-128, 1988.

85. R. K. Brayton, R. Camposano, G. DeMicheli, R. H. J. M. Otten, and J. vanEijndhoven, "The Yorktown silicon compiler," in *Silicon compilation*, ed. D. D. Gajski, Addison-Wesley, 1988.

86. J. Rabaey, H. DeMan, J. Vanhoof, G. Goossens, and F. Catthoor, "CATHEDRAL II: A synthesis system for multiprocessor DSP," in *Silicon compilation*, ed. D. D. Gajski, Addison-Wesley, 1988.

87. R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Trans. on CAD of integrated circuits and systems*, pp. 1062-1081, 1987.

88. C. Terman, "RSIM - A logic-level timing simulator," *Proc. of the IEEE International Conference on Computer Design*, pp. 437-440, 1983.

89. C. Niessen, "Hierarchical design methodologies and tools for VLSI chips," *IEEE Proc.* , vol. 71, no. 1, pp. 66-75, Jan. 1983.

90. C. H. Sequin, "Managing VLSI complexity: An outlook," *IEEE Proc.* , vol. 71, no. 1, pp. 149-166, Jan. 1983.

91. F. Maruyama and M. Fujita, "Hardware verification," *IEEE Computer*, pp. 22-32, Feb. 1985.

92. M. Browne, E. Clarke, D. Hill, and B. Mishra, "Automatic verification of sequential circuits using temporal logic," *Technical Report CMU-CS-85-100*, Dept. of Computer Science, Carnegie-Mellon University, 1985.

93. K. Supowit and S. J. Friedman, "A new method for verifying sequential circuits," *Proc. of 23rd Design Automation Conference*, pp. 200-207, June 1986.

94. I. Gertner and R. P. Kurshan, "Logical analysis of digital circuits," *Proc. of 8th Int'l Conf. on Computer Hardware Description Languages and their Applications*, pp. 47-67, April 1987.

95. S. Devadas, H. K Ma, and A. R. Newton, "On the verification of sequential machines at differing levels of abstraction," *IEEE Trans. Computer-Aided Design*, vol. 7 , June 1988.

96. Z. Kohavi, *Switching and finite automata theory* , McGraw-Hill, 1970.

97. S. C. Lee, *Modern switching theory and digital design*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

98. R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*, p. Kluwer Academic Publishers, Boston, 1984.

99. G. DeMicheli, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Optimal state assignment of finite state machines," *IEEE Trans. Computer-Aided Design*, vol. CAD-4, pp. 269-285, July 1985.

100. P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Trans. Computers*, vol. C-30, Mar. 1981.

101. R. S. Wei and A. Sangiovanni-Vincentelli, "PROTEUS: A logic verification system for combinational logic circuits," *Proc. of Int'l Testing Conference*, pp. 350-359, 1986.

102. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, in *The design and analysis of computer algorithms*, Addison-Wesley, 1974.

103. S. Devadas, H. K. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli, "MUSTANG: State assignment of finite state machines for optimal Multi-Level Logic Implementations," *Proc. of Int'l Conf. on Computer-Aided Design*, pp. 16-19, Nov. 1987.

104. M. Hill, et al., "Design decisions in SPUR," *IEEE Computer*, vol. 19 , no. 10, pp. 8-24, Nov. 1986.

105. H. H. Ball, "VLSI/Software engineering design methodology," *Workshop Report: VLSI and Software Engineering Workshop*, pp. 3-5, IEEE Computer Society, Port Chester, NY , Oct. 1982.

106. Z. Kishimoto et al., "The intersection of VLSI and software engineering for testing and verification," *Workshop Report: VLSI and Software Engineering Workshop*, pp. 10-49, IEEE Computer Society, Port Chester, NY , Oct. 1982.

107. G. M. Baudet et al., "The relationship between HDLs and programming language," *Workshop Report: VLSI and Software Engineering Workshop*, pp. 64-74, IEEE Computer Society, Port Chester, NY , Oct. 1982.

108. H. K. Berg, W. E. Boebert, W. R. Franta, and T. G. Moher, *Formal methods of program verification and specification*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

109. J. E. Stoy, *Denotational semantics: the Scott-Strachey approach to programming language theory*, The MIT Press, Cambridge, Mass., 1977.

110. P. Lucas and K. Walk, "On the formal description of PL/I," in *Annual Review in Automatic Programming*, ed. M. I. Halpern and C. J. Shaw, vol. 6, pp. 105-182, Pergamon Press, Oxford, 1971.

111. P. Wegener, "The Vienna definition language," *Computing Surveys*, vol. 4, no. 1, 1972.

112. A. Van Wijngaarden et al., "Revised report on the algorithmic language ALGOL 68," *Acta Informatica*, vol. 5, 1975.

113. A. Church, "The calculi of lambda-conversion," *Annals of Mathematical Studies*, vol. 6, Princeton University Press, Princeton, NJ, 1951.

114. R. W. Floyd, "Assigning meanings to programs," *Proc. Symposium in Applied Mathematics*, vol. 19, pp. 19-32, American Mathematical Society, 1967.

115. C. A. R. Hoare, "An axiomatic approach to computer programming ," *Comm. ACM*, vol. 12, no. 10, 1969.

116. J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract data types and software validation," *Comm. ACM*, no. 21, pp. 1048-1064, 1978.

117. D. R. Musser, "Abstract data type specifications in the AFFIRM system," *Proc. Specification of Reliable Software Conf.*, Cambridge, Mass., 1979.

118. J. A. Goguen and J. J. Tardo, "An introduction to OBJ: a language for writing and testing formal algebraic specification," *Proc. Specification of Reliable Software Conf.*, Cambridge, Mass., 1979.

119. D. E. Knuth and P. E. Bendix, "Simple word problems in universal algebras," in *Computational Problems in Abstract Algebra*, ed. J. Leech, Pergamon Press, Elmsford, NY, 1969.

120. D. L. Parnas, "A technique for software module specification with examples," *Comm. ACM*, vol. 15, no. 5, 1972.

121. O. Roubine and L. Robinson, "SPECIAL (SPECIfication and Assertion Language): reference manual," TR-CSG-45, SRI International, Menlo Park, CA, 1977.

122. R. N. Principato, "A formalization of the state machine specification technique," MIT/lcs/TR-2-2, MIT Laboratory for Computer Science Report, 1978.

123. J. Scheid, *INA JO*, Presentation at Air Force summer study on system security, Cambridge, Mass., 1979.

124. W. A. Wulf, R. L. London, and M. Shaw, "An introduction to the construction and verification of Alphard programs," *IEEE Trans. Software Engineering*, vol. SE-2, no. 4, pp. 253-265, Dec. 1976.

125. G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London, "Notes on the design of Euclid," *Proc. ACM Conf. on Language Design for Reliable Software*, vol. 12, no. 3, 1977.

126. L. Flon and J. Misra, "A unified approach to the specification and verification of abstract data types," *Proc. Specification for Reliable Software Conf.*, Cambridge, Mass., 1979.

127. Y. Huh, "Formal specification and verification of hierarchical VLSI design," Ph.D. Dissertation, Electronics Lab., Stanford University, Dec. 1985.

128. A. Blikle and A. Mazurkiewicz, "An algebraic approach to the theory of programs, algorithms, languages and recursiveness," in *Mathematical Foundations of Computer Science*, Warsaw, Poland, 1972.

129. D. Siewiorek, "Introducing ISP," *IEEE Computer*, vol. 7, no. 12, pp. 39-41, Dec. 1974.

130. D. L. Dietmeyer, "Introducing DDL," *IEEE Computer*, vol. 7, no. 12, pp. 34-38, Dec. 1974.

131. D. D. Hill, "Adlib: a modular strongly-typed computer design language," *IEEE IFIP 4th Int. Conference on Computer Hardware Description Languages and their applications*, pp. 75-81, Palo Alto, CA, 1979.

132. R. E. Bryant, "Symbolic manipulation of Boolean functions using a graphical representation," *Proc. 22nd Design Automation Conference*, pp. 688-694, 1985.

133. J. Loeckx and K. Sieber, *The foundations of program verification*, John Wiley & Sons, New York, NY, 1984.

134. C.-L. Chang and R. C.-T. Lee, in *Symbolic logic and mechanical theorem proving*, Academic Press, New York, NY., 1973.

135. Y. Chu, "Introducing CDL," *IEEE Computer*, vol. 7, no. 12, pp. 31-33, Dec. 1974.

136. J. D. McCharen, R. A. Overbeek, and L. Wos, "Problems and experiments for and with automated theorem-proving problems," *IEEE Trans. on Computers*, vol. C-25, no. 8, pp. 773-782, 1976.

137. N. Suzuki, "Experience with specification and verification of hardware using PROLOG," *Lecture Note in Computer Science*, vol. 163, pp. 161-173, Springer-Verlag, 1984.

138. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.

139. I. Bratko, *Prolog programming for artificial intelligence*, Addison-Wesley, 1986.

140. T. Uehara, T. Saito, F. Maruyama, and N. Kawato, "DDL verifier," *IEEE IFIP 5th Int. Conf. on Computer Hardware Description Languages and their applications*, pp. 51-64, Sep. 1981.

141. R. S. Boyer and J. S. Moore, "A computational logic," in *ACM Monograph Series*, Academic Press, 1979.

142. J. Whitehead and B. Russel, *Principia mathematica*, 1, Cambridge, UK, 1935.

143. A. Church, "A formulation of the simple theory of types," *Journal of Symbolic Logic*, 1940.

144. A. Camilleri, M. J. C. Gordon, and T. Melham, "Hardware verification using higher-order logic," in *From HDL descriptions to guaranteed correct circuit designs*, ed. D. Borrione, pp. 43-67, 1987.

145. M. Gordon, R. Milner, and C. P. Wadsworth, "Edinburgh LCF: A mechanized logic of computation," in *Lecture Notes in Computer Science*, vol. 78, Springer Verlag, 1979.

146. J.-L. Paillet, "A functional model for descriptions and specifications of digital devices," in *From HDL descriptions to guaranteed correct circuit designs*, ed. D. Borrione, pp. 21-42, 1987.

147. G. E. Hughes and M. J. Cresswell, *An introduction to modal logic*, Methuen and Co., Ltd., London, UK, 1968.

148. N. Rescher and A. Urquhart, *Temporal logic*, Springer-Verlag, New York, NY, 1971.

149. Z. Manna, *Mathematical theory of computation*, McGraw-Hill, New York, NY, 1974.

150. Z. Manna and A. Pnueli, "Verification of concurrent programs: the temporal framework," in *The correctness problem in computer science*, ed. R. S. Boyer and J. S. Moore, pp. 215-273, Academic Press, New York, NY, 1981.

151. M. J. Bennet, "Proving correctness of asynchronous circuits using temporal logic," Ph.D Dissertation, Dept. of Computer Science, Univ. of California, Los Angeles, 1986.

152. Y. Malachi and S. S. Owicki, "Temporal specifications of self-timed systems," in *VLSI Systems and Computations*, ed. H. T. Kung et al., pp. 203-212, Computer Science Press, Rockville, MD, 1981.

153. C. A Mead and L Conway, *Introduction to VLSI systems*, Addison-Wesley, 1980.

154. J. Halpern, Z. Manna, and B. Moszkowski, "A hardware semantics based on temporal intervals," *Proc. 10th Int. Colloq. Automata, Languages and Programming*, pp. 278-

291, Springer-Verlag, Barcelona, Spain, 1983.

155. S. Kono, M. Fujita, and H. Tanaka, "Implementation of temporal logic programming language Tokio," *Logic Programming Conference '85*, pp. 138-147, 1985.

156. C. E. Shannon, "A symbolic analysis of relay and switching circuits," *Trans. AIEE*, vol. 57, pp. 713-723, 1938.

157. D. Siewiorek, "Introducing PMS," *IEEE Computer*, vol. 7, no. 12, pp. 42-44, Dec. 1974.

158. R. H. Campbell, A. M. Koelmans, and M. R. McLauchlan, "STRICT: a design language for strongly typed recursive integrated circuits," *IEE Proceedings*, vol. 132, no. 2, pp. 108-115, Mar. 1985.

159. M. Sheeran, "muFP, a language for VLSI design," *ACM Symposium on LISP and Functional Programming*, pp. 104-112, Austin, Texas, 1984.

160. D. R. Coelho and W. M. VanCleemput, "Helix: A tool for multilevel simulation of VLSI systems," *The third International Conference on Semi-Custom IC's*, London, England, Nov. 1983.

161. *HSL-FX User's Manual*, NTT LSI Laboratories, 1988.

162. M. Daniels, "Design criterias and formal description techniques," in *Computer hardware description languages and their applications*, ed. C. J. Koomen, pp. 195-212, North-Holland, 1987.

163. T. L. Thorp and N. E. Peeling, "The role of HDLs in the digital design process," *VLSI Conference* , Vancouver, Canada, 1987.

164. "iPSC/2 brochures and application software reference material," *order number 280110-001*, Intel Scientific Computers, Beaverton, Ore..

165. *DYNIX Programmer's Manual Series*, Sequent Computer Systems, Inc., 1987.

166. *Butterfly User's Manual*, BBN Advanced Computers Inc., 1986.

167. C. A. R. Hoare and J. C. Shepherdson, and C. A. R. Hoare, "An overview of some formal methods for program design," *IEEE Computer*, vol. 20, no. 9, pp. 85-91, Prentice-Hall, Englewood Cliffs, N.J., Sep. 1987.

169. J. Backus, "Can programming be liberated from the von Neuman style? A functional style and its algebra of programs," *Communication of the ACM*, vol. 21, no. 8, pp. 613-641, Aug. 1978.

170. O. -J. Dahl and K. Nyqaard, "Simular - an ALGOL based simulation language," *Comm. ACM*, pp. 671-678, Sep. 1966.

171. B. Stroustrup, in *The C++ programming language*, Addison-Wesley, 1986.

172. R. H. Wallace, in *Practitioner's guide to Ada*, McGraw-Hill, New York, N.Y., 1986.

173. N. Wirth, in *Programming in MODULA-2*, Springer-Verlag, New York, N.Y., 1985.

174. W. B. Ackerman and J. B. Dennis, *VAL - A value oriented algorithmic language: preliminary reference manual*, MIT Laboratory for Computer Science, Tech. Rep., Cambridge, MA, June 1979.

175. Arvind, K. P. Gostelow, and W. Plouffe, *An asynchronous programming language and computing machine*, Dept. Information and Computer Science Technical Report 114a University of California, Irvine, Dec. 1978.

176. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*, Addison-Wesley , 1986.

177. R. Wilensky, in *LISPcraft*, Norton & Company, 1984.

178. G. Langdon, *Computer design*, Computeach Press Inc. , 1982.

179. N. Weste and K. Eshraghian, *Principals of CMOS VLSI design. A systems perspective*, Addison Wesley, 1985.

180. L. Glasser and D. Dobberpuhl, *The design and analysis of VLSI circuits*, Addison Wesley, 1985.

181. C. Seitz, "Self timed system," *Proc. of Caltech Conference on VLSI*, 1980.

182. T. Kirkpatrick and N. Clark, "PERT as an aid to logic design," *IBM Jour. of Research and Development*, pp. 135-141, Mar. 1966.

183. M. Wold, "Design verification and performance analysis," *Proc. of the 15th Design Automation Conference*, pp. 264-270, 1978.

184. T. McWilliams, "Verification of timing constraints on large digital systems," *Proc. of the 17th Design Automation Conference*, pp. 139-147, 1980.

185. T. Sasaki, A. Yamada, T. Aoyama, K. Hasegawa, S. Kato, and S. Sato, "Hierarchical design verification for large digital systems," *Proc. of the 18th Design Automation Conference*, pp. 105-112, June 1981.

186. R. Kamikawai, M. Yamada, and T. Chiba, "A critical path delay check system," *Proc. of the 18th Design Automation Conference*, pp. 118-123, June 1981.

187. E. Tamura, K. Okawa, and T. Nakano, "Path delay analysis for hierarchical building block layout system," *Proc. of the 20th Design Automation Conference*, pp. 403-410, June 1983.

188. N. Weiner, *Hummingbird: A timing analyzer for the Berkeley synthesis system*. Dept. of EECS, Univ. of California, Berkeley, Dec. 1987.

189. R. McGeer, "On the interaction of functional and timing behavior of combinational logic circuits," Ph.D Dissertation, University of California, Berkeley , CA, 1989.

190. D. Hodges and H. Jackson, *Analysis and design of digital integrated circuits*, McGraw-Hill, 1983.

191. Y. Kim, "Accurate timing verification for digital VLSI designs," Ph.D Dissertation, University of California, Berkeley , CA, Dec. 1988.

192. D. Brand and V. Iyengar, "Timing analysis using functional relationships," *ICCAD-86*, pp. 126-129, Santa Clara, CA, Nov. 1986.

193. J. Benkoski, E. Meersch, and H. De Man, "Efficient algorithms for solving the false path problem in timing verification," *ICCAD-87*, pp. 44-47, Santa Clara, CA, Nov. 1987.

194. R. B. Segal , in *BDSYN User's Manual*, University of California, Berkeley, 1987.

195. A. Church, "An unsolvable problem of number theory," *American Journal of Mathematics*, vol. 58, pp. 345-363, 1936.

196. A. M. Turing, "On computable numbers, with an application to the entschiedungs problem," *Proc. London Math. Soc.*, vol. 2, no. 42, pp. 230-265, 1936.

197. J. A. Robinson, "A machine oriented logic based on the resolution principle," *Journal of Assoc. Comput. Mach.*, vol. 12, pp. 23-41, 1965.

198. J. R. Slagle, "Automatic theorem proving with renamable and semantic resolution," *Journal of Assoc. Comput. Mach.*, vol. 14, pp. 687-697, 1967.

199. B. Meltzer, "Theorem-proving for computers: Some results on resolution and renaming," *Computer Journal*, vol. 8, pp. 341-343, 1966.

200. R. Kowalski and P. Hayes, "Semantic trees in automatic theorem proving," in *Machine Intelligence*, ed. B. Meltzer and D. Michie, vol. 4 , pp. 87-101, American Elsevier, New York, 1969.

201. R. S. Boyer, "Locking: A restriction of resolution," Ph. D. Dissertation, University of Texas, at Austin, Texas, 1971.

202. D. W. Loveland, "A linear format for resolution," *Proc. IRIA Symp. Automatic Demonstration, Versailles, France*, pp. 147-162, Springer Verlag, New York, 1970.

203. D. Luckham, "Refinements in resolution theory," *Proc. IRIA Symp. Automatic Demonstration, Versailles, France*, pp. 163-190, Springer Verlag, New York, 1970.

204. R. Anderson and W. W. Bledsoe, "A linear format for resolution with merging and a new technique for establishing completeness," *Journal of Assoc. Comput. Mach.*, vol. 17 , pp. 525-534, 1970.

205. R. Yates, B. Raphael, and T. Hart, "Resolution graphs," *Artificial Intelligence*, vol. 1, pp. 257-290, 1970.

206. R. Reiter, "Two results on ordering for resolution with merging and linear format," *Journal of Assoc. Comput. Mach.*, vol. 18 , pp. 630-646, 1971.

207. R. Kowalski and D. Keuhner, "Linear resolution with selection function," in *Metamathematics Unit*, Edinburgh University, Scotland, 1970.

208. L. Wos, D. Carson, and G. A. Robinson, "The unit preference strategy in theorem proving," *Proc. AFIPS 1964 Fall Joint Computer Conf.*, vol. 26, pp. 616-621, 1964.

209. C. L. Chang, "The unit proof and the input proof in theorem proving," *Journal of Assoc. Comput. Mach.*, vol. 17, pp. 698-707, 1970.

210. L. Wos, G. A. Robinson, and D. F. Carson, "Efficiency and completeness of the set of support strategy in theorem proving," *Journal of Assoc. Comput. Mach.*, vol. 12, pp. 536-541, 1965.

211. A. J. Cohn, "A proof of correctness of the Viper microprocessor: The first level," *Proc. of the Calgary Hardware Verification Workshop*, Calgary, Canada, Jan. 1987.

212. P. Lescanne, "Computer experiments with the REVE term rewriting system generator," *Proc. the Tenth ACM Symposium on the Principles of Programming Languages*, Austin, Texas, Jan. 1983.

213. G. Huet and J. M. Hullot, "Proof by induction in equational theories with constructors," *JCCS*, vol. 25, no. 2, 1982.

APPENDIX A

DESCRIPTIONS OF EXAMPLE CELLS

A.1 Exclusive-OR

```
(defcell xor
   (directive expand)
   (input (p bool) (q bool))
   (output (r bool))
   (let r (or (and (not p) q) (and p (not q)))))
```

A.2 Library Gates

```
;;;; library gates :
;;;;      inv1
;;;;      and2 and3 and4
;;;;      or2 or3 or4
;;;;      xor2
;;;;      nand2 nand3 nand4
;;;;      nor2 nor3 nor4

(defcell inv1
   (directive expand)
   (input (x bool))
   (output (y bool))
   (let y (delay 1 (not x))))

(defcell and2
   (directive expand)
   (input (x1 bool) (x2 bool))
   (output (y bool))
   (let y (delay 1.6 (and x1 x2))))

(defcell and3
   (directive expand)
   (input (x1 bool) (x2 bool) (x3 bool))
```

```
   (output (y bool))
   (let y (delay 2.0 (and x1 x2 x3))))

(defcell and4
   (directive expand)
   (input (x1 bool) (x2 bool) (x3 bool) (x4 bool))
   (output (y bool))
   (let y (delay 2.8 (and x1 x2 x3 x4))))

(defcell and5
   (directive expand)
   (input (x1 bool) (x2 bool) (x3 bool) (x4 bool) (x5 bool))
   (output (y bool))
   (let y (delay 4.0 (and x1 x2 x3 x4 x5))))

(defcell or2
   (directive expand)
   (input (x1 bool) (x2 bool))
   (output (y bool))
   (let y (delay 1.6 (or x1 x2))))

(defcell or3
   (directive expand)
   (input (x1 bool) (x2 bool) (x3 bool))
   (output (y bool))
   (let y (delay 2.0 (or x1 x2 x3))))

(defcell or4
   (directive expand)
   (input (x1 bool) (x2 bool) (x3 bool) (x4 bool))
   (output (y bool))
   (let y (delay 2.8 (or x1 x2 x3 x4))))

(defcell or5
   (directive expand)
   (input (x1 bool) (x2 bool) (x3 bool) (x4 bool) (x5 bool))
   (output (y bool))
   (let y (delay 4.0 (or x1 x2 x3 x4 x5))))

(defcell xor2
   (directive expand)
   (input (p bool) (q bool))
   (output (r bool))
   (let r (delay 1.6 (or (and (not p) q) (and p (not q))))))

(defcell nand2
   (directive expand)
```

```
    (input (x1 bool) (x2 bool))
    (output (y bool))
    (let y (delay 1.6 (not (and x1 x2)))))

(defcell nand3
    (directive expand)
    (input (x1 bool) (x2 bool) (x3 bool))
    (output (y bool))
    (let y (delay 2.0 (not (and x1 x2 x3)))))

(defcell nand4
    (directive expand)
    (input (x1 bool) (x2 bool) (x3 bool) (x4 bool))
    (output (y bool))
    (let y (delay 2.8 (not (and x1 x2 x3 x4)))))

(defcell nor2
    (directive expand)
    (input (x1 bool) (x2 bool))
    (output (y bool))
    (let y (delay 1.6 (not (or x1 x2)))))

(defcell nor3
    (directive expand)
    (input (x1 bool) (x2 bool) (x3 bool))
    (output (y bool))
    (let y (delay 2.0 (not (or x1 x2 x3)))))

(defcell nor4
    (directive expand)
    (input (x1 bool) (x2 bool) (x3 bool) (x4 bool))
    (output (y bool))
    (let y (delay 2.8 (not (or x1 x2 x3 x4)))))
```

A.3 Examples Used in Timing Verification

```
;;;;
;;;; Hitchcock's example
;;;;

(defcell hitchcock
    (input (x bool) (y bool))
    (output (z bool))
    (local (tmp bool))
    (let tmp (if (not y) (delay 5 x) (delay 100 x)))
```

```
    (let z (if y (delay 5 tmp) (delay 100 tmp))))

;;;;
;;;; Brand's examples
;;;;

(defcell brand1
    (input (a bool) (b bool))
    (output (d bool))
    (let d (or2 (and2 (inv1 a) b) b)))

(defcell brand2
    (input (a bool) (b bool))
    (output (d bool))
    (let d (and2 a (or2 (and2 (inv1 a) b) b))))

;;;;
;;;; Examples obtained from the instruction-fetch controller of SPUR
;;;;

;;;; Without optimization in the MIS-II session

(defcell fet.nop
  (input (memBusy_CV4<0> bool)(resetIB_CV3<0> bool)(flush_CV2<0> bool))
  (input (dataValid_C3<0> bool)(LatchWrKpsw_C2<0> bool))
  (input (LatchNotSpd_C1<0> bool)(ibMiss_C2<0> bool))
  (input (Fet_State_C1<2> bool))
  (input (Fet_State_C1<1> bool))
  (input (Fet_State_C1<0> bool))
  (output (FET_STATE_LOGIC<2> bool))
  (output (FET_STATE_LOGIC<1> bool))
  (output (FET_STATE_LOGIC<0> bool))
  (local ([448] bool)([451] bool)([481] bool)([455] bool)([316] bool))
  (local ([450] bool)([346] bool)([347] bool)([314] bool)([344] bool))
  (local ([345] bool)([313] bool)([453] bool)([341] bool)([321] bool))
  (local ([447] bool)([339] bool)([319] bool)([402] bool)([338] bool))
  (local ([318] bool)([335] bool)([336] bool)([317] bool)([464] bool))
  (local ([326] bool)([333] bool)([334] bool)([324] bool)([366] bool))
  (local ([332] bool)([323] bool)([329] bool)([330] bool)([322] bool))
  (local ([327] bool)([249] bool))
  (let [448] (inv1 Fet_State_C1<2> ))
  (let [451] (inv1 Fet_State_C1<0> ))
  (let [344] (nor4 dataValid_C3<0> [448] Fet_State_C1<1> [451] ))
  (let [481] (inv1 LatchWrKpsw_C2<0> ))
  (let [455] (nand3 [448] [481] LatchNotSpd_C1<0> ))
  (let [316] (nor2 flush_CV2<0> ibMiss_C2<0> ))
  (let [450] (inv1 Fet_State_C1<1> ))
```

```
 (let [346] (nor3 [455] [316] [450] ))
 (let [347] (nor2 Fet_State_C1<1> [448] ))
 (let [314] (nor2 [346] [347] ))
 (let [345] (nor2 Fet_State_C1<0> [314] ))
 (let [313] (nor2 [344] [345] ))
 (let FET_STATE_LOGIC<2> (nor2 resetIB_CV3<0> [313] ))
 (let [453] (inv1 LatchNotSpd_C1<0> ))
 (let [341] (nor4 flush_CV2<0> LatchWrKpsw_C2<0> ibMiss_C2<0> [453] ))
 (let [321] (nor3 [453] [341] LatchWrKpsw_C2<0> ))
 (let [335] (nor4 Fet_State_C1<2> [450] Fet_State_C1<0> [321] ))
 (let [402] (nor2 Fet_State_C1<2> [450] ))
 (let [447] (inv1 dataValid_C3<0> ))
 (let [339] (nor2 [447] [448] ))
 (let [319] (nor2 [339] [448] ))
 (let [338] (nor2 Fet_State_C1<1> [319] ))
 (let [318] (nor2 [402] [338] ))
 (let [336] (nor2 [318] [451] ))
 (let [317] (nor2 [335] [336] ))
 (let FET_STATE_LOGIC<1> (nor2 resetIB_CV3<0> [317] ))
 (let [464] (nand3 [448] [481] LatchNotSpd_C1<0> ))
 (let [326] (nor2 flush_CV2<0> ibMiss_C2<0> ))
 (let [333] (nor3 [464] [326] [450] ))
 (let [334] (nor2 Fet_State_C1<1> [448] ))
 (let [324] (nor2 [333] [334] ))
 (let [329] (nor3 [324] memBusy_CV4<0> Fet_State_C1<0> ))
 (let [366] (nor2 Fet_State_C1<1> [448] ))
 (let [332] (nor3 [450] LatchNotSpd_C1<0> Fet_State_C1<2> ))
 (let [323] (nor2 [366] [332] ))
 (let [330] (nor2 [323] [451] ))
 (let [322] (nor2 [329] [330] ))
 (let [327] (nor2 resetIB_CV3<0> [322] ))
 (let [249] (nor2 [327] resetIB_CV3<0> ))
 (let FET_STATE_LOGIC<0> (inv1 [249] )))

;;;; With optimization in the MIS-II session

(defcell fet.opt
 (input (memBusy_CV4<0> bool)(resetIB_CV3<0> bool)(flush_CV2<0> bool))
 (input (dataValid_C3<0> bool)(LatchWrKpsw_C2<0> bool))
 (input (LatchNotSpd_C1<0> bool)(ibMiss_C2<0> bool))
 (input (Fet_State_C1<2> bool))
 (input (Fet_State_C1<1> bool))
 (input (Fet_State_C1<0> bool))
 (output (FET_STATE_LOGIC<2> bool))
 (output (FET_STATE_LOGIC<1> bool))
 (output (FET_STATE_LOGIC<0> bool))
 (local ([141] bool)([140] bool)([215] bool)([217] bool)([90] bool))
```

```
(local ([143] bool)([77] bool)([86] bool)([223] bool)([201] bool))
(local ([219] bool)([221] bool)([154] bool)([199] bool)([197] bool))
(local ([147] bool)([149] bool)([82] bool)([83] bool)([78] bool))
(local ([145] bool)([79] bool)([80] bool)([2] bool))
(let [141] (inv1 Fet_State_C1<1> ))
(let [215] (nand2 [141] Fet_State_C1<2> ))
(let [140] (inv1 Fet_State_C1<2> ))
(let [217] (nand2 [140] Fet_State_C1<1> ))
(let [197] (nand2 [215] [217] ))
(let [154] (inv1 resetIB_CV3<0> ))
(let [90] (nor2 flush_CV2<0> ibMiss_C2<0> ))
(let [143] (inv1 LatchNotSpd_C1<0> ))
(let [77] (nor3 LatchWrKpsw_C2<0> [90] [143] ))
(let [86] (nor3 [77] Fet_State_C1<2> [141] ))
(let [219] (inv1 [86] ))
(let [223] (nand2 [141] dataValid_C3<0> ))
(let [201] (nand2 [223] Fet_State_C1<2> ))
(let [221] (nand2 [201] Fet_State_C1<0> ))
(let [199] (nand2 [219] [221] ))
(let [147] (nand2 [154] [199] ))
(let [149] (nand3 [197] [154] [147] ))
(let FET_STATE_LOGIC<2> (inv1 [149] ))
(let FET_STATE_LOGIC<1> (inv1 [147] ))
(let [82] (nor3 [147] LatchNotSpd_C1<0> [141] ))
(let [83] (nor2 Fet_State_C1<1> [140] ))
(let [78] (nor2 [82] [83] ))
(let [145] (inv1 Fet_State_C1<0> ))
(let [79] (nor2 [78] [145] ))
(let [80] (nor2 memBusy_CV4<0> [149] ))
(let [2] (nor3 resetIB_CV3<0> [79] [80] ))
(let FET_STATE_LOGIC<0> (inv1 [2] )))


;;;;
;;;; Examples obtained from the instruction decode logic of SPUR CPU
;;;;

;;;; Without optimization in the MIS-II session: gate count 151

(defcell opcode.nop
  (input (opcode_C3<25> bool)(opcode_C3<27> bool)(opcode_C3<28> bool))
  (input (opcode_C3<29> bool)(opcode_C3<30> bool)(opcode_C3<31> bool))
  (input (opcode_C3<26> bool)(fpuEn_C3<0> bool))
  (output (trapCall_V1<0> bool)(rdPC_V1<0> bool)(load_V1<0> bool))
  (output (cpuLd32_V1<0> bool)(cpuLoad_V1<0> bool)(cxr_V1<0> bool))
  (output (ldStExt_V1<0> bool)(subMode_V1<0> bool)(addSub_V1<0> bool))
  (output (addNt_V1<0> bool)(and_V1<0> bool)(or_V1<0> bool))
```

```
(output (xor_V1<0> bool)(extract_V1<0> bool)(insert_V1<0> bool))
(output (readTag_V1<0> bool)(writeTag_V1<0> bool)(shLeft_V1<0> bool))
(output (shRightA_V1<0> bool)(shRightL_V1<0> bool))
(output (fixnumOp_V1<0> bool)(regReg_V1<0> bool)(rdSpec_V1<0> bool))
(output (rdIns_V1<0> bool)(rdKpsw_V1<0> bool)(wrSpec_V1<0> bool))
(output (wrIns_V1<0> bool)(wrKpsw_V1<0> bool)(zeroRd_V1<0> bool))
(output (lowToUp_V1<0> bool)(callJump_V1<0> bool))
(output (userCall_V1<0> bool)(retTrap_V1<0> bool)(allRet_V1<0> bool))
(output (miss_V1<0> bool)(sync_V1<0> bool)(invIB_V1<0> bool))
(output (fpuLdSt_V1<0> bool)(fpuOper_V1<0> bool)(illegalOp_V1<0> bool))
(local ([835] bool)([836] bool)([842] bool)([838] bool)([1692] bool))
(local ([1616] bool)([1686] bool)([1688] bool)([1690] bool))
(local ([839] bool)([843] bool)([845] bool)([1612] bool)([1680] bool))
(local ([1682] bool)([1622] bool)([1684] bool)([1624] bool))
(local ([1676] bool)([1678] bool)([1620] bool)([2072] bool))
(local ([1674] bool)([752] bool)([1618] bool)([1542] bool))
(local ([1552] bool)([175] bool)([863] bool)([102] bool)([868] bool))
(local ([872] bool)([1670] bool)([1610] bool)([877] bool)([880] bool))
(local ([884] bool)([889] bool)([894] bool)([899] bool)([904] bool))
(local ([909] bool)([914] bool)([919] bool)([924] bool)([929] bool))
(local ([934] bool)([1666] bool)([1668] bool)([1632] bool))
(local ([1564] bool)([504] bool)([1568] bool)([492] bool))
(local ([1572] bool)([480] bool)([1576] bool)([961] bool)([456] bool))
(local ([1585] bool)([971] bool)([1839] bool)([975] bool)([979] bool))
(local ([1664] bool)([1638] bool)([1825] bool)([1662] bool))
(local ([396] bool)([1636] bool)([1588] bool)([1816] bool))
(local ([986] bool)([991] bool)([352] bool)([1594] bool)([340] bool))
(local ([1597] bool)([1660] bool)([306] bool)([1626] bool))
(local ([1555] bool)([1656] bool)([1658] bool)([1642] bool))
(local ([1654] bool)([280] bool)([1640] bool)([1603] bool))
(local ([156] bool)([157] bool)([130] bool)([154] bool)([155] bool))
(local ([1017] bool)([129] bool)([152] bool)([1557] bool)([219] bool))
(local ([246] bool)([1558] bool)([2158] bool)([1024] bool))
(local ([1031] bool)([1035] bool)([1043] bool)([2118] bool))
(local ([2123] bool)([2128] bool)([2133] bool)([2148] bool))
(let [835] (inv1 opcode_C3<25> ))
(let [842] (inv1 opcode_C3<29> ))
(let [838] (inv1 opcode_C3<27> ))
(let [843] (inv1 opcode_C3<30> ))
(let [839] (inv1 opcode_C3<28> ))
(let [845] (inv1 opcode_C3<31> ))
(let [836] (inv1 opcode_C3<26> ))
(let [1686] (nand4 [835] opcode_C3<27> [836] opcode_C3<29> ))
(let [1688] (nand3 [836] [842] opcode_C3<27> ))
(let [1692] (nand2 [838] opcode_C3<29> ))
(let [1616] (nand2 [1692] opcode_C3<29> ))
(let [1690] (nand2 [1616] opcode_C3<26> ))
```

```
(let [1612] (nand3 [1686] [1688] [1690] ))
(let [1542] (nand4 [839] [843] [845] [1612] ))
(let [752] (nor3 opcode_C3<31> opcode_C3<30> [839] ))
(let [2072] (nand2 [838] opcode_C3<26> ))
(let [1680] (nand2 [838] opcode_C3<25> ))
(let [1682] (nand2 [835] opcode_C3<27> ))
(let [1622] (nand2 [1680] [1682] ))
(let [1676] (nand2 [1622] opcode_C3<29> ))
(let [1684] (nand2 [835] opcode_C3<27> ))
(let [1624] (nand2 [1684] opcode_C3<27> ))
(let [1678] (nand2 [842] [1624] ))
(let [1620] (nand2 [1676] [1678] ))
(let [1674] (nand2 [836] [1620] ))
(let [1618] (nand2 [2072] [1674] ))
(let [1552] (nand3 [752] [1618] fpuEn_C3<0> ))
(let load_V1<0> (nand2 [1542] [1552] ))
(let [863] (nand3 [843] [839] opcode_C3<27> ))
(let [175] (nor3 opcode_C3<26> opcode_C3<25> [842] ))
(let [102] (nor2 [175] [842] ))
(let cpuLd32_V1<0> (nor3 [863] opcode_C3<31> [102] ))
(let cpuLoad_V1<0> (inv1 [1542] ))
(let [868] (nand4 [838] opcode_C3<25> [839] [843] ))
(let cxr_V1<0> (nor3 [868] opcode_C3<31> [836] ))
(let [872] (nand4 opcode_C3<25> opcode_C3<27> [839] [842] ))
(let ldStExt_V1<0> (nor3 [872] opcode_C3<31> [836] ))
(let [1670] (nand2 [842] opcode_C3<25> ))
(let [1610] (nand2 [842] [1670] ))
(let [877] (nand3 [1610] [836] opcode_C3<31> ))
(let subMode_V1<0> (nor4 [877] opcode_C3<30> opcode_C3<27> opcode_C3<28> ))
(let [880] (nand4 [838] [839] [842] [843] ))
(let addSub_V1<0> (nor3 [880] opcode_C3<26> [845] ))
(let [884] (nand4 [835] [836] [838] opcode_C3<28> ))
(let addNt_V1<0> (nor4 [884] [845] opcode_C3<29> opcode_C3<30> ))
(let [889] (nand4 [835] opcode_C3<26> [838] [839] ))
(let and_V1<0> (nor4 [889] [845] opcode_C3<29> opcode_C3<30> ))
(let [894] (nand4 opcode_C3<25> opcode_C3<26> [838] [839] ))
(let or_V1<0> (nor4 [894] [845] opcode_C3<29> opcode_C3<30> ))
(let [899] (nand4 [835] [836] [839] opcode_C3<27> ))
(let xor_V1<0> (nor4 [899] [845] opcode_C3<29> opcode_C3<30> ))
(let [904] (nand4 [836] opcode_C3<25> opcode_C3<27> opcode_C3<28> ))
(let extract_V1<0> (nor4 [904] [845] opcode_C3<29> opcode_C3<30> ))
(let [909] (nand4 opcode_C3<25> opcode_C3<26> opcode_C3<27> opcode_C3<28> ))
(let insert_V1<0> (nor4 [909] [845] opcode_C3<29> opcode_C3<30> ))
(let [914] (nand4 [835] [836] opcode_C3<27> opcode_C3<28> ))
(let readTag_V1<0> (nor4 [914] [845] opcode_C3<29> opcode_C3<30> ))
(let [919] (nand4 [835] opcode_C3<26> opcode_C3<27> opcode_C3<28> ))
(let writeTag_V1<0> (nor4 [919] [845] opcode_C3<29> opcode_C3<30> ))
```

```
(let [924] (nand4 [836] opcode_C3<25> [839] opcode_C3<27> ))
(let shLeft_V1<0> (nor4 [924] [845] opcode_C3<29> opcode_C3<30> ))
(let [929] (nand4 [835] opcode_C3<26> [839] opcode_C3<27> ))
(let shRightA_V1<0> (nor4 [929] [845] opcode_C3<29> opcode_C3<30> ))
(let [934] (nand4 opcode_C3<25> opcode_C3<26> [839] opcode_C3<27> ))
(let shRightL_V1<0> (nor4 [934] [845] opcode_C3<29> opcode_C3<30> ))
(let fixnumOp_V1<0> (nor4 opcode_C3<28> opcode_C3<29> opcode_C3<30> [845] ))
(let [1666] (nand4 [835] [838] [836] opcode_C3<28> ))
(let [1668] (nand2 opcode_C3<27> opcode_C3<28> ))
(let [1632] (nand3 [1666] [1668] opcode_C3<28> ))
(let [1564] (nand4 [842] [843] [1632] opcode_C3<31> ))
(let regReg_V1<0> (inv1 [1564] ))
(let [504] (nor4 opcode_C3<26> [835] opcode_C3<28> [838] ))
(let [1568] (nand4 [504] opcode_C3<31> [843] opcode_C3<29> ))
(let rdSpec_V1<0> (inv1 [1568] ))
(let [492] (nor4 [835] [836] opcode_C3<28> [838] ))
(let [1572] (nand4 [492] opcode_C3<31> [843] opcode_C3<29> ))
(let rdIns_V1<0> (inv1 [1572] ))
(let [480] (nor4 opcode_C3<25> [836] opcode_C3<28> [838] ))
(let [1576] (nand4 [480] opcode_C3<31> [843] opcode_C3<29> ))
(let rdKpsw_V1<0> (inv1 [1576] ))
(let [961] (nand4 [836] opcode_C3<25> opcode_C3<27> opcode_C3<28> ))
(let wrSpec_V1<0> (nor4 [961] [845] opcode_C3<30> [842] ))
(let [456] (nor4 [835] [836] opcode_C3<27> opcode_C3<28> ))
(let [1585] (nand4 [456] opcode_C3<31> [843] opcode_C3<29> ))
(let wrIns_V1<0> (inv1 [1585] ))
(let [971] (nand4 [835] opcode_C3<26> opcode_C3<27> opcode_C3<28> ))
(let wrKpsw_V1<0> (nor4 [971] [845] opcode_C3<30> [842] ))
(let [975] (nand3 [1572] [1564] [1568] ))
(let [1839] (nand4 [839] opcode_C3<29> opcode_C3<30> opcode_C3<31> ))
(let [979] (nand3 [1839] [1576] [1542] ))
(let zeroRd_V1<0> (nor2 [975] [979] ))
(let [396] (nor3 opcode_C3<30> [839] [842] ))
(let [1825] (nand2 [836] opcode_C3<25> ))
(let [1664] (nand2 opcode_C3<26> opcode_C3<27> ))
(let [1638] (nand2 [1664] opcode_C3<27> ))
(let [1662] (nand2 [835] [1638] ))
(let [1636] (nand2 [1825] [1662] ))
(let [1588] (nand3 [396] [1636] opcode_C3<31> ))
(let lowToUp_V1<0> (inv1 [1588] ))
(let [1816] (nand3 opcode_C3<31> opcode_C3<29> opcode_C3<30> ))
(let callJump_V1<0> (inv1 [1816] ))
(let userCall_V1<0> (inv1 [1839] ))
(let [986] (nand4 [836] opcode_C3<25> [838] opcode_C3<28> ))
(let retTrap_V1<0> (nor4 [986] [845] opcode_C3<30> [842] ))
(let [991] (nand4 [838] opcode_C3<28> [843] opcode_C3<29> ))
(let allRet_V1<0> (nor3 [991] opcode_C3<26> [845] ))
```

```
(let [352] (nor4 opcode_C3<25> opcode_C3<26> opcode_C3<27> [839] ))
(let [1594] (nand4 [352] opcode_C3<31> [842] opcode_C3<30> ))
(let miss_V1<0> (inv1 [1594] ))
(let [340] (nor4 [835] [836] [838] [839] ))
(let [1597] (nand4 [340] [845] opcode_C3<29> opcode_C3<30> ))
(let sync_V1<0> (inv1 [1597] ))
(let [306] (nor3 [843] opcode_C3<29> [839] ))
(let [1660] (nand3 [836] [835] opcode_C3<27> ))
(let [1626] (nand2 [1660] opcode_C3<27> ))
(let [1555] (nand4 [306] [1626] [845] fpuEn_C3<0> ))
(let fpuLdSt_V1<0> (nand2 [1552] [1555] ))
(let [280] (nor3 opcode_C3<31> [842] [843] ))
(let [1656] (nand2 [838] [836] ))
(let [1658] (nand2 opcode_C3<26> opcode_C3<27> ))
(let [1642] (nand2 [1656] [1658] ))
(let [1654] (nand2 [1642] opcode_C3<28> ))
(let [1640] (nand2 [1654] opcode_C3<28> ))
(let [1603] (nand3 [280] [1640] fpuEn_C3<0> ))
(let fpuOper_V1<0> (inv1 [1603] ))
(let [1017] (nand3 opcode_C3<30> [839] [842] ))
(let [154] (nor3 opcode_C3<26> opcode_C3<25> [838] ))
(let [156] (nor2 opcode_C3<25> opcode_C3<27> ))
(let [157] (nor2 [835] [838] ))
(let [130] (nor2 [156] [157] ))
(let [155] (nor2 [130] [836] ))
(let [129] (nor2 [154] [155] ))
(let [152] (nor3 [1017] opcode_C3<31> [129] ))
(let [1557] (inv1 [1555] ))
(let [219] (nor3 load_V1<0> [152] [1557] ))
(let [1024] (nand4 [1603] [219] [1564] [1588] ))
(let [246] (nor4 opcode_C3<27> opcode_C3<28> opcode_C3<30> [842] ))
(let [1558] (nand3 [246] [836] opcode_C3<31> ))
(let [1031] (nand4 [1816] [1558] [1568] [1572] ))
(let [1035] (nand4 [1576] [1585] [1594] [1597] ))
(let [1043] (inv1 [2158] ))
(let illegalOp_V1<0> (nor4 [1024] [1031] [1035] [1043] ))
(let [2118] (nand4 [836] opcode_C3<25> [838] [839] ))
(let [2123] (nand4 [836] opcode_C3<25> [838] opcode_C3<28> ))
(let [2128] (nand4 opcode_C3<25> opcode_C3<26> [838] opcode_C3<28> ))
(let [2133] (nand4 [835] [836] [839] opcode_C3<27> ))
(let trapCall_V1<0> (nor4 [2118] [845] opcode_C3<29> [843] ))
(let invIB_V1<0> (nor4 [2123] [845] opcode_C3<29> [843] ))
(let [2148] (nor4 [2128] [845] opcode_C3<29> opcode_C3<30> ))
(let rdPC_V1<0> (nor4 [2133] [845] opcode_C3<30> [842] ))
(let [2158] (nor4 invIB_V1<0> rdPC_V1<0> trapCall_V1<0> [2148] )))
```

```
;;;; With optimization in the MIS-II session: gate count 132

(defcell opcode.opt
  (input (opcode_C3<25> bool)(opcode_C3<27> bool)(opcode_C3<28> bool))
  (input (opcode_C3<29> bool)(opcode_C3<30> bool)(opcode_C3<31> bool))
  (input (opcode_C3<26> bool)(fpuEn_C3<0> bool))
  (output (trapCall_V1<0> bool)(rdPC_V1<0> bool)(load_V1<0> bool))
  (output (cpuLd32_V1<0> bool)(cpuLoad_V1<0> bool)(cxr_V1<0> bool))
  (output (ldStExt_V1<0> bool)(subMode_V1<0> bool)(addSub_V1<0> bool))
  (output (addNt_V1<0> bool)(and_V1<0> bool)(or_V1<0> bool))
  (output (xor_V1<0> bool)(extract_V1<0> bool)(insert_V1<0> bool))
  (output (readTag_V1<0> bool)(writeTag_V1<0> bool)(shLeft_V1<0> bool))
  (output (shRightA_V1<0> bool)(shRightL_V1<0> bool))
  (output (fixnumOp_V1<0> bool)(regReg_V1<0> bool)(rdSpec_V1<0> bool))
  (output (rdIns_V1<0> bool)(rdKpsw_V1<0> bool)(wrSpec_V1<0> bool))
  (output (wrIns_V1<0> bool)(wrKpsw_V1<0> bool)(zeroRd_V1<0> bool))
  (output (lowToUp_V1<0> bool)(callJump_V1<0> bool))
  (output (userCall_V1<0> bool)(retTrap_V1<0> bool)(allRet_V1<0> bool))
  (output (miss_V1<0> bool)(sync_V1<0> bool)(invIB_V1<0> bool))
  (output (fpuLdSt_V1<0> bool)(fpuOper_V1<0> bool))
  (output (illegalOp_V1<0> bool))
  (local ([1216] bool)([502] bool)([495] bool)([505] bool)([492] bool))
  (local ([52] bool)([494] bool)([499] bool)([815] bool)([1212] bool))
  (local ([491] bool)([509] bool)([64] bool)([507] bool)([498] bool))
  (local ([1010] bool)([454] bool)([1068] bool)([584] bool)([782] bool))
  (local ([517] bool)([1064] bool)([1066] bool)([1022] bool))
  (local ([1220] bool)([1062] bool)([497] bool)([1020] bool))
  (local ([792] bool)([519] bool)([415] bool)([589] bool)([522] bool))
  (local ([592] bool)([525] bool)([527] bool)([501] bool)([529] bool))
  (local ([531] bool)([533] bool)([166] bool)([167] bool))
  (local (regReg_State<0>0.1 bool)([806] bool)([809] bool)([812] bool))
  (local ([540] bool)([548] bool)([546] bool)([550] bool)([552] bool))
  (local ([163] bool)([1258] bool)([511] bool)([62] bool)([848] bool))
  (local ([1243] bool)([795] bool)([1052] bool)([500] bool))
  (local ([1028] bool)([836] bool)([1048] bool)([1050] bool))
  (local ([1026] bool)([851] bool)([844] bool)([1044] bool))
  (local ([1046] bool)([1018] bool)([568] bool)([244] bool)([236] bool))
  (local ([223] bool)([840] bool)([1040] bool)([1042] bool))
  (local ([1016] bool)([1036] bool)([1038] bool)([1014] bool))
  (local ([1034] bool)([1030] bool)([1032] bool)([1012] bool))
  (local ([583] bool)([327] bool)([1221] bool)([1251] bool))
  (local ([1228] bool)([294] bool)([1257] bool))
  (let [498] (inv1 opcode_C3<27> ))
  (let [499] (inv1 opcode_C3<26> ))
  (let [501] (inv1 [1216] ))
  (let [502] (nand3 [1216] opcode_C3<25> opcode_C3<30> ))
  (let trapCall_V1<0> (nor3 [502] opcode_C3<28> opcode_C3<29> ))
```

```
(let [495] (inv1 opcode_C3<30> ))
(let [505] (nand3 opcode_C3<31> [495] opcode_C3<27> ))
(let [492] (inv1 opcode_C3<29> ))
(let [52] (nor3 [505] opcode_C3<28> [492] ))
(let [494] (inv1 opcode_C3<25> ))
(let [815] (nand3 [52] [494] [499] ))
(let rdPC_V1<0> (inv1 [815] ))
(let [497] (inv1 opcode_C3<31> ))
(let [507] (inv1 [1212] ))
(let [491] (inv1 opcode_C3<28> ))
(let [509] (nand2 [491] [495] ))
(let [64] (nor3 [509] opcode_C3<27> opcode_C3<31> ))
(let [1068] (nand2 opcode_C3<26> [64] ))
(let [1010] (nand2 [507] opcode_C3<29> ))
(let [454] (nor3 [509] opcode_C3<31> [498] ))
(let [584] (nand2 [1010] [454] ))
(let cpuLoad_V1<0> (nand2 [1068] [584] ))
(let [517] (nand2 [492] opcode_C3<28> ))
(let [782] (inv1 cpuLoad_V1<0> ))
(let [1064] (nand2 [782] opcode_C3<26> ))
(let [1066] (nand2 opcode_C3<28> opcode_C3<25> ))
(let [1022] (nand3 [517] [1064] [1066] ))
(let [1062] (nand2 [498] [1022] ))
(let [1020] (nand2 [1220] [1062] ))
(let [792] (nand4 [495] [497] [1020] fpuEn_C3<0> ))
(let load_V1<0> (nand2 [792] [782] ))
(let cpuLd32_V1<0> (inv1 [584] ))
(let [519] (nand2 opcode_C3<25> opcode_C3<26> ))
(let cxr_V1<0> (nor4 [519] [509] opcode_C3<27> opcode_C3<31> ))
(let [415] (nor3 [519] opcode_C3<31> [498] ))
(let [589] (nand3 [415] [491] [492] ))
(let ldStExt_V1<0> (inv1 [589] ))
(let [522] (nand2 [494] [492] ))
(let [592] (nand4 [491] [495] [522] [1216] ))
(let subMode_V1<0> (inv1 [592] ))
(let [525] (inv1 fixnumOp_V1<0> ))
(let addSub_V1<0> (nor3 [525] opcode_C3<27> opcode_C3<26> ))
(let [527] (nand3 opcode_C3<28> [494] [492] ))
(let addNt_V1<0> (nor3 [527] opcode_C3<30> [501] ))
(let [529] (nand2 [494] opcode_C3<26> ))
(let and_V1<0> (nor3 [529] opcode_C3<27> [525] ))
(let or_V1<0> (nor3 [519] opcode_C3<27> [525] ))
(let [531] (nand2 opcode_C3<27> fixnumOp_V1<0> ))
(let xor_V1<0> (nor2 [507] [531] ))
(let [533] (nand2 [499] opcode_C3<25> ))
(let extract_V1<0> (nor3 [533] [505] [517] ))
(let insert_V1<0> (nor3 [517] [505] [519] ))
```

```
(let readTag_V1<0> (nor3 [527] opcode_C3<26> [505] ))
(let writeTag_V1<0> (nor3 [527] [499] [505] ))
(let shLeft_V1<0> (nor2 [531] [533] ))
(let shRightA_V1<0> (nor2 [529] [531] ))
(let shRightL_V1<0> (nor2 [519] [531] ))
(let [166] (nor4 opcode_C3<30> [497] opcode_C3<26> [522] ))
(let [167] (nor2 opcode_C3<29> [505] ))
(let regReg_State<0>0.1 (nor3 fixnumOp_V1<0> [166] [167] ))
(let regReg_V1<0> (inv1 regReg_State<0>0.1 ))
(let [806] (nand3 [52] [499] opcode_C3<25> ))
(let rdSpec_V1<0> (inv1 [806] ))
(let [809] (nand3 [52] opcode_C3<25> opcode_C3<26> ))
(let rdIns_V1<0> (inv1 [809] ))
(let [812] (nand3 [52] [494] opcode_C3<26> ))
(let rdKpsw_V1<0> (inv1 [812] ))
(let [540] (nand2 opcode_C3<28> opcode_C3<29> ))
(let wrSpec_V1<0> (nor3 [533] [505] [540] ))
(let [327] (nor2 [509] [519] ))
(let wrKpsw_V1<0> (nor3 [540] [505] [529] ))
(let [546] (nand2 regReg_State<0>0.1 [782] ))
(let [548] (inv1 callJump_V1<0> ))
(let userCall_V1<0> (nor2 opcode_C3<28> [548] ))
(let [550] (nand3 [812] [806] [809] ))
(let zeroRd_V1<0> (nor3 [546] userCall_V1<0> [550] ))
(let [552] (nand2 [548] opcode_C3<31> ))
(let [163] (nor3 [552] [529] [540] ))
(let [294] (nor2 [163] wrSpec_V1<0> ))
(let lowToUp_V1<0> (inv1 [1258] ))
(let retTrap_V1<0> (nor4 opcode_C3<30> [494] [501] [540] ))
(let [840] (inv1 miss_V1<0> ))
(let [511] (nand2 [497] opcode_C3<27> ))
(let sync_V1<0> (nor4 [495] [519] [511] [540] ))
(let [844] (inv1 sync_V1<0> ))
(let [62] (nor3 [501] [494] [495] ))
(let [848] (nand3 [62] [492] opcode_C3<28> ))
(let invIB_V1<0> (inv1 [848] ))
(let [795] (inv1 [1243] ))
(let fpuLdSt_V1<0> (nand2 [792] [795] ))
(let [1052] (nand2 opcode_C3<27> opcode_C3<26> ))
(let [500] (nand2 [498] [499] ))
(let [1028] (nand2 [1052] [500] ))
(let [1048] (nand2 [548] [1028] ))
(let [836] (nand4 [491] opcode_C3<31> opcode_C3<29> opcode_C3<30> ))
(let [1050] (nand2 [491] [836] ))
(let [1026] (nand2 [1048] [1050] ))
(let [851] (nand4 opcode_C3<29> opcode_C3<30> [1026] fpuEn_C3<0> ))
(let fpuOper_V1<0> (inv1 [851] ))
```

```
(let [1044] (nand3 [844] [589] opcode_C3<25> ))
(let [1046] (nand3 opcode_C3<27> [494] opcode_C3<26> ))
(let [1018] (nand4 [1044] [1046] [500] opcode_C3<30> ))
(let [1034] (nand2 [497] [1018] ))
(let [568] (nand2 [499] regReg_State<0>0.1 ))
(let [244] (nor3 [568] opcode_C3<29> [497] ))
(let [236] (nor2 trapCall_V1<0> miss_V1<0> ))
(let [1030] (nand3 [244] [848] [236] ))
(let [1036] (nand3 opcode_C3<26> opcode_C3<30> opcode_C3<31> ))
(let [223] (nor3 [550] wrIns_V1<0> sync_V1<0> ))
(let [1040] (nand4 [223] [815] [592] opcode_C3<29> ))
(let [1042] (nand4 [494] opcode_C3<28> [840] regReg_State<0>0.1 ))
(let [1016] (nand2 [1040] [1042] ))
(let [1038] (nand2 [1016] [1258] ))
(let [1014] (nand2 [1036] [1038] ))
(let [1032] (nand2 [548] [1014] ))
(let [1012] (nand3 [1034] [1030] [1032] ))
(let [583] (nand2 [851] [1012] ))
(let illegalOp_V1<0> (nor3 [583] [1243] load_V1<0> ))
(let [1212] (nor2 opcode_C3<25> opcode_C3<26> ))
(let [1216] (nor3 [497] opcode_C3<27> opcode_C3<26> ))
(let [1220] (nand3 [1212] [782] opcode_C3<27> ))
(let [1221] (inv1 [327] ))
(let wrIns_V1<0> (nor4 [1221] [497] opcode_C3<27> [492] ))
(let [1228] (nor2 [1212] [498] ))
(let fixnumOp_V1<0> (nor4 opcode_C3<28> opcode_C3<30> opcode_C3<29> [497] ))
(let allRet_V1<0> (nor4 [491] [492] callJump_V1<0> [501] ))
(let [1243] (nor4 [1251] [1228] opcode_C3<29> [491] ))
(let callJump_V1<0> (nor3 [497] [492] [495] ))
(let [1251] (nand3 fpuEn_C3<0> [497] opcode_C3<30> ))
(let miss_V1<0> (nor3 [527] [495] [501] ))
(let [1257] (inv1 [294] ))
(let [1258] (nor2 allRet_V1<0> [1257] )))
```

A.4 Examples Used in Functional Verification

```
;;;; 1-bit binary full-adder

(defcell adder-1d
  (input (a bool) (b bool) (cin bool))
  (output (sum bool) (cout bool))

  (let sum (xor2 cin (xor2 a b)))
  (let cout (or2 (and2 a b) (and2 cin (xor2 a b)))))
```

```
(defcell r-c1d
  (input (a1 bool) (b1 bool) (cin bool))
  (output (s1 bool) (cout bool))
  (let (s1 cout) (adder-1d a1 b1 cin)))

;;;; 2-bit adder: ripple-carry implementation

(defcell r-c2d
  (input (x1 bool) (x2 bool) (y1 bool) (y2 bool) (cin bool))
  (output (s1 bool) (s2 bool) (cout bool))
  (local (c1 bool))
  (let (s1 c1) (adder-1d x1 y1 cin))
  (let (s2 cout) (adder-1d x2 y2 c1)))

;;;; 2-bit adder: carry-look-ahead implementation

(defcell l-a2d
  (input (x1 bool) (x2 bool) (y1 bool) (y2 bool) (cin bool))
  (output (s1 bool) (s2 bool) (cout bool))
  (local (g1 bool) (g2 bool) (p1 bool) (p2 bool) (c1 bool))
  (let g1 (and2 x1 y1))
  (let g2 (and2 x2 y2))
  (let p1 (or2 x1 y1))
  (let p2 (or2 x2 y2))
  (let c1 (or2 g1 (and2 p1 cin)))
  (let cout (or3 g2 (and2 p2 g1) (and3 p2 p1 cin)))
  (let s1 (xor2 cin (xor2 x1 y1)))
  (let s2 (xor2 c1 (xor2 x2 y2))))

;;;; 3-bit adder: ripple-carry implementation

(defcell r-c3d
  (input (x1 bool) (x2 bool) (x3 bool)
         (y1 bool) (y2 bool) (y3 bool) (cin bool))
  (output (s1 bool) (s2 bool) (s3 bool) (cout bool))
  (local (c1 bool) (c2 bool))
  (let (s1 c1) (adder-1d x1 y1 cin))
  (let (s2 c2) (adder-1d x2 y2 c1))
  (let (s3 cout) (adder-1d x3 y3 c2)))

;;;; 3-bit adder: carry-look-ahead implementation

(defcell l-a3d
  (input (x1 bool) (x2 bool) (x3 bool)
         (y1 bool) (y2 bool) (y3 bool) (cin bool))
  (output (s1 bool) (s2 bool) (s3 bool) (cout bool))
  (local (g1 bool) (g2 bool) (g3 bool)
```

```
        (p1 bool) (p2 bool) (p3 bool) (c1 bool) (c2 bool))
  (let g1 (and2 x1 y1))
  (let g2 (and2 x2 y2))
  (let g3 (and2 x3 y3))
  (let p1 (or2 x1 y1))
  (let p2 (or2 x2 y2))
  (let p3 (or2 x3 y3))
  (let c1 (or2 g1 (and2 p1 cin)))
  (let c2 (or3 g2 (and2 p2 g1) (and3 p2 p1 cin)))
  (let cout (or4 g3 (and2 p3 g2) (and3 p3 p2 g1) (and4 p3 p2 p1 cin)))
  (let s1 (xor2 cin (xor2 x1 y1)))
  (let s2 (xor2 c1 (xor2 x2 y2)))
  (let s3 (xor2 c2 (xor2 x3 y3))))

;;;; 4-bit adder: ripple-carry implementation

(defcell r-c4d
  (input (a1 bool) (a2 bool) (a3 bool) (a4 bool)
         (b1 bool) (b2 bool) (b3 bool) (b4 bool) (cin bool))
  (output (s1 bool) (s2 bool) (s3 bool) (s4 bool) (cout bool))
  (local (c1 bool) (c2 bool) (c3 bool))
  (let (s1 c1) (adder-1d a1 b1 cin))
  (let (s2 c2) (adder-1d a2 b2 c1))
  (let (s3 c3) (adder-1d a3 b3 c2))
  (let (s4 cout) (adder-1d a4 b4 c3)))

;;;; 4-bit adder: carry-look-ahead implementation

(defcell l-a4d
  (input (a1 bool) (a2 bool) (a3 bool) (a4 bool)
         (b1 bool) (b2 bool) (b3 bool) (b4 bool) (cin bool))
  (output (s1 bool) (s2 bool) (s3 bool) (s4 bool) (cout bool))
  (local (g1 bool) (g2 bool) (g3 bool) (g4 bool)
         (p1 bool) (p2 bool) (p3 bool) (p4 bool)
         (c1 bool) (c2 bool) (c3 bool))
  (let g1 (and2 a1 b1))
  (let g2 (and2 a2 b2))
  (let g3 (and2 a3 b3))
  (let g4 (and2 a4 b4))
  (let p1 (or2 a1 b1))
  (let p2 (or2 a2 b2))
  (let p3 (or2 a3 b3))
  (let p4 (or2 a4 b4))
  (let c1 (or2 g1 (and2 p1 cin)))
  (let c2 (or3 g2 (and2 p2 g1) (and3 p2 p1 cin)))
  (let c3 (or4 g3 (and2 p3 g2) (and3 p3 p2 g1) (and4 p3 p2 p1 cin)))
  (let cout (or5 g4 (and2 p4 g3) (and3 p4 p3 g2) (and4 p4 p3 p2 g1)
```

```
                    (and5 p4 p3 p2 p1 cin)))
  (let s1 (xor2 cin (xor2 a1 b1)))
  (let s2 (xor2 c1 (xor2 a2 b2)))
  (let s3 (xor2 c2 (xor2 a3 b3)))
  (let s4 (xor2 c3 (xor2 a4 b4))))

;;;; 4-bit adder: incorrect implementation

(defcell wrong4d
  (input (a1 bool) (a2 bool) (a3 bool) (a4 bool)
         (b1 bool) (b2 bool) (b3 bool) (b4 bool) (cin bool))
  (output (s1 bool) (s2 bool) (s3 bool) (s4 bool) (cout bool))
  (local (g1 bool) (g2 bool) (g3 bool) (g4 bool)
         (p1 bool) (p2 bool) (p3 bool) (p4 bool)
         (c1 bool) (c2 bool) (c3 bool))
  (let g1 (and2 a1 b1))
  (let g2 (and2 a2 b2))
  (let g3 (and2 a3 b3))
  (let g4 (and2 a4 b4))
  (let p1 (or2 a1 b1))
  (let p2 (or2 a2 b2))
  (let p3 (or2 a3 b3))
  (let p4 (or2 a4 b4))
  (let c1 (or2 g1 (and2 p1 cin)))
  (let c2 (or3 g2 (and2 p2 g1) (and3 p2 p1 cin)))
  (let c3 (or4 g3 (and2 p3 g2) (and3 p3 p2 g1) (and4 p3 p2 p1 cin)))
  (let cout (and5 g4 (and2 p4 g3) (and3 p4 p3 g2) (and4 p4 p3 p2 g1)
                  (and5 p4 p3 p2 p1 cin)))
  (let s1 (xor2 cin (xor2 a1 b1)))
  (let s2 (xor2 c1 (xor2 a2 b2)))
  (let s3 (xor2 c2 (xor2 a3 b3)))
  (let s4 (xor2 c3 (xor2 a4 b4))))

;;;;
;;;; implementation of 2-bit alu
;;;;

;;;; 1-bit alu: implemented using MIS-II

(defcell alubit
  (input (a<0> bool) (b<0> bool) (cin<0> bool) (and_op<0> bool))
  (input (add_op<0> bool) (comp_op<0> bool) (pass_op<0> bool))
  (output (sum<0> bool) (cout<0> bool))
  (local ([120] bool)([121] bool)([162] bool)([174] bool)([176] bool))
  (local ([73] bool)([118] bool)([119] bool)([80] bool)([72] bool))
  (local ([78] bool)([71] bool)([75] bool)([76] bool)([168] bool))
  (local ([89] bool))
```

```
 (let [120] (inv1 a<0> ))
 (let [168] (nand2 [120] comp_op<0> ))
 (let [73] (nor3 cin<0> a<0> b<0> ))
 (let [121] (inv1 b<0> ))
 (let [162] (nand2 [120] [121] ))
 (let [174] (nand2 [162] cin<0> ))
 (let [176] (nand2 a<0> b<0> ))
 (let cout<0> (nand2 [174] [176] ))
 (let [118] (inv1 add_op<0> ))
 (let [75] (nor3 [73] cout<0> [118] ))
 (let [119] (inv1 cin<0> ))
 (let [80] (nor2 [118] [119] ))
 (let [72] (nor2 [80] and_op<0> ))
 (let [78] (nor2 [72] [121] ))
 (let [71] (nor2 [78] pass_op<0> ))
 (let [76] (nor2 [71] [120] ))
 (let [89] (nor2 [75] [76] ))
 (let sum<0> (nand2 [168] [89] )))

;;;; decode logic for alu: implemented using MIS-II

(defcell op_decode
  (input (op<1> bool)(op<0> bool))
  (output (and_op<0> bool)(add_op<0> bool))
  (output (comp_op<0> bool)(pass_op<0> bool))
  (local ([268] bool)([269] bool))
  (let [268] (inv1 op<1> ))
  (let and_op<0> (nor2 op<0> [268] ))
  (let [269] (inv1 op<0> ))
  (let add_op<0> (nor2 [268] [269] ))
  (let comp_op<0> (nor2 op<1> [269] ))
  (let pass_op<0> (nor2 op<1> op<0> )))

;;;; 2-bit alu: implemented using MIS-II

(defcell alu2bit
  (input (a<1> bool) (a<0> bool))
  (input (b<1> bool) (b<0> bool))
  (input (c<0> bool) (op<1> bool) (op<0> bool))
  (output (s<1> bool) (s<0> bool) (c<2> bool))
  (local (c<1> bool))
  (local (and_op bool) (add_op bool) (comp_op bool) (pass_op bool))

  (let (and_op add_op comp_op pass_op) (op_decode op<1> op<0>))
  (let (s<0> c<1>)
      (alubit a<0> b<0> c<0> and_op add_op comp_op pass_op))
  (let (s<1> c<2>)
```

```
      (alubit a<1> b<1> c<1> and_op add_op comp_op pass_op)))

;;;; 1-bit alu: obtained from BDS description

(defcell alubit.bds
  (input (a bool) (b bool) (cin bool))
  (input (and_op bool) (add_op bool) (comp_op bool) (pass_op bool))
  (output (sum bool) (cout bool))
  (local (p bool) (g bool))

  (let g (and a b))
  (let p (xor a b))

  (let cout (or g (and p cin)))

  (let sum (or (and pass_op a) (and comp_op (not a))
               (and and_op g) (and add_op (xor p cin)))))

;;;; decode logic for alu: obtained from BDS description

(defcell op_decode.bds
  (input (op<1> bool)(op<0> bool))
  (output (and_op bool)(add_op bool)(comp_op bool)(pass_op bool))

  (let pass_op (and (not op<1>) (not op<0>)))
  (let comp_op (and (not op<1>) op<0>))
  (let and_op (and op<1> (not op<0>)))
  (let add_op (and op<1> op<0>)))

;;;; type definition of 2-bit vector

(deftype bool2 (struct (<1> bool) (<0> bool)))

;;;; type conversion function: nat to bool2

(defcell nat-to-bool2
  (input (n nat))
  (output (y bool2))
  (local (tmp nat))
  (let tmp (if (< n 4) n (- n 4)))
  (let y (case tmp
               (0 (bool2-cons false false))
               (1 (bool2-cons false true))
               (2 (bool2-cons true false))
               (3 (bool2-cons true true)))))

;;;; type conversion function: bool2 to nat
```

```
(defcell bool2-to-nat
  (input (x bool2))
  (output (y nat))
  (local (b1 bool) (b0 bool))
  (let b1 (bool2-<1> x))
  (let b0 (bool2-<0> x))
  (let y (case (b1 b0)
               ((false false) 0)
               ((false true) 1)
               ((true false) 2)
               ((true true) 3))))

;;;; 2-bit alu: obtained from BDS description

(defcell alu2bit.bds
  (input (a<1> bool) (a<0> bool))
  (input (b<1> bool) (b<0> bool))
  (input (c<0> bool) (op<1> bool) (op<0> bool))
  (output (s<1> bool) (s<0> bool) (c<2> bool))
  (local (tmp nat) (tmp-bool2 bool2))
  (let tmp (+ (bool2-to-nat (bool2-cons a<1> a<0>))
              (bool2-to-nat (bool2-cons b<1> b<0>))
               (if c<0> 1 0)))
  (let c<2> (if (< tmp 4) false true))
  (let tmp-bool2 (nat-to-bool2 tmp))
  (let (s<1> s<0>)
       (case (op<1> op<0>)
             ((false false) (values a<1> a<0>))
             ((false true) (values (not a<1>) (not a<0>)))
             ((true false) (values (and a<1> b<1>) (and a<0> b<0>)))
             ((true true) (values (bool2-<1> tmp-bool2)
                                  (bool2-<0> tmp-bool2))))))
;;;;
;;;; sequential circuit examples
;;;;

;;;; top-level cell

(defcell iufsm
  (input
        (phi1 sync)
        (phi2 sync)
        (phi3 sync)
        (phi4 sync)
        (random sync)
        (p1 bool)
        (p2 bool)
```

```
        (p3 bool)
        (p4 bool)
        (iuPre_C4 bool)
        (iuEn_C4 bool)
      (reset_L_C1 bool)
      (notSpd_C4 bool)
      (dataValid_C3 bool)
      (cacheBusy_C3 bool)
      (invIB_C1 bool)
      (trapReq_C2 bool)
      (load_Ex bool)
      (store_Ex bool)
      (lowToUp_Ex bool)
      (invIB_Ex bool)
       (wrKpsw_Ex bool)
      (blockMiss_C2 bool)
      (ibMiss_C2 bool))
(state
       (FET_STATE_LOGIC<2> bool)
       (FET_STATE_LOGIC<1> bool)
       (FET_STATE_LOGIC<0> bool))
(state
       (PF_STATE_LOGIC<2> bool)
       (PF_STATE_LOGIC<1> bool)
       (PF_STATE_LOGIC<0> bool))
(state
       (FET_State_C4<2> bool)
       (FET_State_C4<1> bool)
       (FET_State_C4<0> bool)
       (PF_State_C4<2> bool)
       (PF_State_C4<1> bool)
       (PF_State_C4<0> bool)
       (resetIB_CV3 bool)
       (memBusy_CV4 bool)
       (busSBusy_C2 bool))
(output
       (random_clk bool)
       (random_clk_L bool)
       (ldIBfetPC_1 bool)
       (readIB_2 bool)
       (invalidate_2 bool)
      (bypass_2 bool)
      (ibToBusI_3 bool)
      (missToBusI_3 bool)
      (trapToBusI_3 bool)
      (readToBusI_3 bool)
      (ibTagWr_4 bool)
```

```
      (ibWrite_4 bool)
      (readML_4 bool)
      (fetPCToBusS_4 bool)
       (ldIBrefPC_4 bool)
       (ibINCToBusS_4 bool)
       (notFetPending_C1 bool)
      (fetch_C3 bool)
      (preFetch_C3 bool)
       (invalidBlock_C1 bool))
(local
       (flush_CV2 bool)
       (LatchWrKpsw_C2 bool)
       (LatchNotSpd_C1 bool)
       (FET_State_C1<2> bool)
       (FET_State_C1<1> bool)
       (FET_State_C1<0> bool)
       (PF_State_C1<2> bool)
       (PF_State_C1<1> bool)
       (PF_State_C1<0> bool)
       (startingPF_CV3 bool)
       (LatchIuEn_C1 bool)
       (LatchIuPre_C1 bool)
       (LatchReset_C2 bool)
       (t1 bool)
       (t2 bool)
       (t3 bool)
       (t4 bool)
       (t5 bool)
       (t6 bool)
       (t7 bool)
       (t8 bool)
       (t9 bool)
       )
;; fetfsm
(let (FET_STATE_LOGIC<2> FET_STATE_LOGIC<1> FET_STATE_LOGIC<0>
       FET_State_C4<2> FET_State_C4<1> FET_State_C4<0>
      FET_State_C1<2> FET_State_C1<1> FET_State_C1<0>)
     (fetfsm FET_STATE_LOGIC<2> FET_STATE_LOGIC<1> FET_STATE_LOGIC<0>
             phi1 phi4 memBusy_CV4 resetIB_CV3 flush_CV2
             dataValid_C3 LatchWrKpsw_C2 LatchNotSpd_C1 ibMiss_C2))

;; p1 and its boundary latches
(let (t1 t2 ldIBfetPC_1)
     (p1cell p1 FET_State_C4<2> FET_State_C4<1> FET_State_C4<0>
             notSpd_C4 blockMiss_C2 LatchWrKpsw_C2 flush_CV2))
(let notFetPending_C1 (prev phi1 t1))
(let invalidBlock_C1 (prev phi1 t2))
```

```
;; p2 and its boundary latches
(let (t3 t4 invalidate_2 bypass_2 readIB_2)
    (p2cell p2 FET_State_C4<2> FET_State_C4<1> FET_State_C4<0>
            load_Ex store_Ex lowToUp_Ex invIB_Ex invIB_C1
            LatchNotSpd_C1))
(let flush_CV2 (prev phi2 t3))
(let busSBusy_C2 (prev phi2 t4))

;; p3 and its boundary latches
(let (t5 t6 t7 t8 trapToBusI_3 readToBusI_3 missToBusI_3 ibToBusI_3)
    (p3cell p3 FET_State_C4<2> FET_State_C4<1> FET_State_C4<0>
            PF_State_C4<2> PF_State_C4<1> PF_State_C4<0>
            LatchNotSpd_C1 ibMiss_C2 flush_CV2 reset_L_C1 trapReq_C2
            LatchWrKpsw_C2 busSBusy_c2))
(let startingPF_CV3 (prev phi3 t5))
(let resetIB_CV3 (prev phi3 t6))
(let fetch_C3 (prev phi3 t7))
(let preFetch_C3 (prev phi3 t8))

;; p4
(let (ibWrite_4 ibTagWr_4 fetPCToBusS_4 readML_4 ldIBrefPC_4 ibINCToBusS_4)
    (p4cell p4 PF_State_C1<2> PF_State_C1<1> PF_State_C1<0>
            FET_State_C1<2> FET_State_C1<1> FET_State_C1<0>
            flush_CV2 dataValid_C3 cacheBusy_C3 busSBusy_C2
            resetIB_CV3 LatchWrKpsw_C2 LatchNotSpd_C1 blockMiss_C2
            ibMiss_C2 startingPF_CV3))

;; pffsm
(let (PF_STATE_LOGIC<2> PF_STATE_LOGIC<1> PF_STATE_LOGIC<0>
      PF_State_C4<2> PF_State_C4<1> PF_State_C4<0>
     PF_State_C1<2> PF_State_C1<1> PF_State_C1<0>)
    (pffsm  PF_STATE_LOGIC<2> PF_STATE_LOGIC<1> PF_STATE_LOGIC<0>
            phi1 phi4 memBusy_CV4 resetIB_CV3 flush_CV2 LatchIuEn_C1
            LatchIuPre_C1 LatchWrKpsw_C2 startingPF_CV3))

;; random logic and its boundary latches
(let (t9 random_clk random_clk_L)
    (racell p1 cacheBusy_C3 dataValid_C3 busSBusy_C2 LatchWrKpsw_C2
            LatchReset_C2))
(let memBusy_CV4 (prev phi4 t9))

;; isolated latches
(let LatchIuPre_C1 (prev random iuPre_C4))
(let LatchIuEn_C1 (prev random iuEn_C4))
(let LatchNotSpd_C1 (prev phi1 notSpd_C4))
(let LatchWrKpsw_C2 (prev phi2 WrKpsw_Ex)))
```

```
(defcell fetfsm
 (input
       (phi1 sync)
       (phi4 sync)
       (memBusy_CV4<0> bool)
       (resetIB_CV3<0> bool)
       (flush_CV2<0> bool)
       (dataValid_C3<0> bool)
       (LatchWrKpsw_C2<0> bool)
       (LatchNotSpd_C1<0> bool)
       (ibMiss_C2<0> bool))
 (state
       (FET_STATE_LOGIC<2> bool)
       (FET_STATE_LOGIC<1> bool)
       (FET_STATE_LOGIC<0> bool))
 (output
       (FET_State_C4<2> bool)
       (FET_State_C4<1> bool)
       (FET_State_C4<0> bool)
       (FET_State_C1<2> bool)
       (FET_State_C1<1> bool)
       (FET_State_C1<0> bool))
 (local
       ([89] bool)([142] bool)([140] bool)([221] bool)([199] bool)
       ([139] bool)([81] bool)([82] bool)([77] bool)([144] bool)
       ([147] bool)([78] bool)([79] bool)([2] bool)([267] bool)
       ([264] bool)([76] bool)([254] bool)([257] bool)([219] bool)
       ([261] bool)([266] bool))
 (let [140] (inv1 FET_State_C1<1> ))
 (let [139] (inv1 FET_State_C1<2> ))
 (let [89] (nor2 flush_CV2<0> ibMiss_C2<0> ))
 (let [142] (inv1 LatchNotSpd_C1<0> ))
 (let [76] (nor3 LatchWrKpsw_C2<0> [89] [142] ))
 (let [221] (nand2 [140] dataValid_C3<0> ))
 (let [199] (nand2 [221] FET_State_C1<2> ))
 (let [219] (nand2 [199] FET_State_C1<0> ))
 (let [147] (inv1 FET_STATE_LOGIC<2> ))
 (let [81] (nor3 [140] LatchNotSpd_C1<0> FET_State_C1<2> ))
 (let [82] (nor2 FET_State_C1<1> [139] ))
 (let [77] (nor2 [81] [82] ))
 (let [144] (inv1 FET_State_C1<0> ))
 (let [78] (nor2 [77] [144] ))
 (let [79] (nor2 memBusy_CV4<0> [147] ))
 (let [2] (nor3 resetIB_CV3<0> [78] [79] ))
 (let FET_STATE_LOGIC<0> (inv1 [2] ))
 (let FET_STATE_LOGIC<1> (nor2 resetIB_CV3<0> [267] ))
 (let FET_STATE_LOGIC<2> (nor3 [264] resetIB_CV3<0> FET_STATE_LOGIC<1> ))
```

```
 (let [254] (nor2 FET_State_C1<1> [139] ))
 (let [257] (nor2 FET_State_C1<2> [140] ))
 (let [261] (nor3 [76] FET_State_C1<2> [140] ))
 (let [264] (nor2 [254] [257] ))
 (let [266] (inv1 [219] ))
 (let [267] (nor2 [261] [266] ))
 (let FET_State_C4<2> (prev phi4 FET_STATE_LOGIC<2>))
 (let FET_State_C4<1> (prev phi4 FET_STATE_LOGIC<1>))
 (let FET_State_C4<0> (prev phi4 FET_STATE_LOGIC<0>))
 (let FET_State_C1<2> (prev phi1 FET_State_C4<2>))
 (let FET_State_C1<1> (prev phi1 FET_State_C4<1>))
 (let FET_State_C1<0> (prev phi1 FET_State_C4<0>)))

(defcell p1cell
 (input
       (phi1<0> bool)
       (FET_State_C4<2> bool)
       (FET_State_C4<1> bool)
       (FET_State_C4<0> bool)
       (notSpd_C4<0> bool)
       (blockMiss_C2<0> bool)
       (LatchWrKpsw_C2<0> bool)
       (flush_CV2<0> bool))
 (output
       (notFetPending_V1<0> bool)
       (invalidBlock_V1<0> bool)
       (ldIBfetPC_1<0> bool))
 (local
       ([358] bool)
       ([335] bool)
       ([360] bool)
       ([357] bool)
       ([365] bool))
 (let [358] (inv1 FET_State_C4<1> ))
 (let notFetPending_V1<0> (nand3 FET_State_C4<2> [358] FET_State_C4<0> ))
 (let [335] (nor2 blockMiss_C2<0> flush_CV2<0> ))
 (let [360] (inv1 FET_State_C4<2> ))
 (let invalidBlock_V1<0> (nor3 [335] FET_State_C4<1> [360] ))
 (let [357] (inv1 FET_State_C4<0> ))
 (let [365] (nand3 [357] phi1<0> notSpd_C4<0> ))
 (let ldIBfetPC_1<0> (nor4 [365] LatchWrKpsw_C2<0> FET_State_C4<2> [358] )))

(defcell p2cell
 (input
       (phi2<0> bool)
       (FET_State_C4<2> bool)
       (FET_State_C4<1> bool)
```

```
        (FET_State_C4<0> bool)
        (load_Ex<0> bool)
        (store_Ex<0> bool)
        (lowToUp_Ex<0> bool)
        (invIB_Ex<0> bool)
        (invIB_C1<0> bool)
        (LatchNotSpd_C1<0> bool))
 (output
        (flush_V2<0> bool)
        (busSBusy_V2<0> bool)
        (invalidate_2<0> bool)
        (bypass_2<0> bool)
        (readIB_2<0> bool))
 (local
        (state_flush_CV2<0>1.1 bool)
        ([419] bool)
        ([540] bool)
        ([542] bool)
        ([520] bool)
        ([546] bool)
        ([543] bool))
 (let state_flush_CV2<0>1.1 (nor2 invIB_Ex<0> invIB_C1<0> ))
 (let flush_V2<0> (inv1 state_flush_CV2<0>1.1 ))
 (let [419] (nor3 lowToUp_Ex<0> load_Ex<0> store_Ex<0> ))
 (let busSBusy_V2<0> (inv1 [419] ))
 (let [540] (nand3 flush_V2<0> phi2<0> LatchNotSpd_C1<0> ))
 (let [542] (inv1 [540] ))
 (let invalidate_2<0> [542] )
 (let bypass_2<0> [542] )
 (let [520] (inv1 FET_State_C4<2> ))
 (let [546] (nand3 FET_State_C4<0> [520] FET_State_C4<1> ))
 (let [543] (nand3 [546] phi2<0> LatchNotSpd_C1<0> ))
 (let readIB_2<0> (inv1 [543] )))

(defcell p3cell
 (input
        (phi3<0> bool)
        (FET_State_C4<2> bool)
        (FET_State_C4<1> bool)
        (FET_State_C4<0> bool)
        (PF_State_C4<2> bool)
        (PF_State_C4<1> bool)
        (PF_State_C4<0> bool)
        (LatchNotSpd_C1<0> bool)
        (ibMiss_C2<0> bool)
        (flush_CV2<0> bool)
        (reset_L_C1<0> bool)
```

```
        (trapReq_C2<0> bool)
        (LatchWrKpsw_C2<0> bool)
        (busSBusy_C2<0> bool))
(output
        (startingPF_V3<0> bool)
        (resetIB_V3<0> bool)
        (fetch_V3<0> bool)
        (preFetch_V3<0> bool)
        (trapToBusI_3<0> bool)
        (readToBusI_3<0> bool)
        (missToBusI_3<0> bool)
        (ibToBusI_3<0> bool))
(local
        ([574] bool)
        ([735] bool)
        ([748] bool)
        ([743] bool)
        ([886] bool)
        ([888] bool)
        ([819] bool)
        ([747] bool)
        ([657] bool)
        ([658] bool)
        ([647] bool)
        ([757] bool)
        ([933] bool)
        ([655] bool)
        ([656] bool)
        ([648] bool)
        ([832] bool)
        ([765] bool)
        ([749] bool)
        ([821] bool)
        ([874] bool)
        ([876] bool)
        ([868] bool)
        ([870] bool)
        ([872] bool)
        ([866] bool)
        ([934] bool))
 (let [574] (nor2 ibMiss_C2<0> flush_CV2<0> ))
 (let [735] (nor3 [574] FET_State_C4<2> FET_State_C4<0> ))
 (let [886] (nand3 [735] FET_State_C4<1> LatchNotSpd_C1<0> ))
 (let [748] (inv1 FET_State_C4<0> ))
 (let [743] (inv1 FET_State_C4<1> ))
 (let [888] (nand3 [748] [743] FET_State_C4<2> ))
 (let startingPF_V3<0> (nand2 [886] [888] ))
```

```
 (let [819] (inv1 trapReq_C2<0> ))
 (let resetIB_V3<0> (nand2 [819] reset_L_C1<0> ))
 (let [657] (nor4 FET_State_C4<2> [743] LatchWrKpsw_C2<0> [574] ))
 (let [747] (inv1 FET_State_C4<2> ))
 (let [658] (nor2 FET_State_C4<1> [747] ))
 (let [647] (nor2 [657] [658] ))
 (let fetch_V3<0> (nor3 [647] FET_State_C4<0> busSBusy_C2<0> ))
 (let [757] (inv1 PF_State_C4<2> ))
 (let [655] (nor3 PF_State_C4<0> PF_State_C4<1> [757] ))
 (let [656] (inv1 [933] ))
 (let [648] (nor2 [655] [656] ))
 (let preFetch_V3<0>
     (nor4 LatchWrKpsw_C2<0> busSBusy_C2<0> [648] startingPF_V3<0> ))
 (let [832] (nand2 resetIB_V3<0> phi3<0> ))
 (let trapToBusI_3<0> (inv1 [832] ))
 (let [765] (nand3 phi3<0> [819] reset_L_C1<0> ))
 (let readToBusI_3<0> (nor4 FET_State_C4<2> FET_State_C4<1> [748] [765] ))
 (let [749] (nand2 [747] [748] ))
 (let [870] (nand3 [749] FET_State_C4<1> LatchNotSpd_C1<0> ))
 (let [821] (inv1 LatchWrKpsw_C2<0> ))
 (let [874] (nand3 [574] [821] FET_State_C4<1> ))
 (let [876] (nand2 [743] LatchNotSpd_C1<0> ))
 (let [868] (nand2 [874] [876] ))
 (let [872] (nand3 [868] [747] [748] ))
 (let [866] (nand2 [870] [872] ))
 (let missToBusI_3<0> (nor3 [765] readToBusI_3<0> ibToBusI_3<0> ))
 (let [933] (nand3 PF_State_C4<0> [757] PF_State_C4<1> ))
 (let [934] (inv1 [866] ))
 (let ibToBusI_3<0> (nor2 [934] [765] )))

(defcell p4cell
 (input
       (phi4<0> bool)
       (PF_State_C1<2> bool)
       (PF_State_C1<1> bool)
       (PF_State_C1<0> bool)
       (FET_State_C1<2> bool)
       (FET_State_C1<1> bool)
       (FET_State_C1<0> bool)
       (flush_CV2<0> bool)
       (dataValid_C3<0> bool)
       (cacheBusy_C3<0> bool)
       (busSBusy_C2<0> bool)
       (resetIB_CV3<0> bool)
       (LatchWrKpsw_C2<0> bool)
       (LatchNotSpd_C1<0> bool)
       (blockMiss_C2<0> bool)
```

```
        (ibMiss_C2<0> bool)
        (startingPF_CV3<0> bool))
(output
        (ibWrite_4<0> bool)
        (ibTagWr_4<0> bool)
        (fetPCToBusS_4<0> bool)
        (readML_4<0> bool)
        (ldIBrefPC_4<0> bool)
        (ibINCToBusS_4<0> bool))
(local
        ([1285] bool)
        ([1031] bool)
        ([1020] bool)
        ([1112] bool)
        ([1116] bool)
        ([1021] bool)
        ([1218] bool)
        ([1120] bool)
        ([1123] bool)
        ([1023] bool)
        ([1114] bool)
        ([1029] bool)
        ([1030] bool)
        ([1106] bool)
        ([1228] bool)
        ([1179] bool)
        ([1022] bool)
        ([1127] bool)
        ([1131] bool)
        ([1293] bool)
        ([1108] bool)
        ([1300] bool)
        ([1109] bool)
        ([1289] bool)
        ([1297] bool)
        ([1307] bool))
(let [1109] (inv1 PF_State_C1<1> ))
(let [1106] (inv1 dataValid_C3<0> ))
(let [1031] (inv1 [1285] ))
(let [1112] (inv1 FET_State_C1<2> ))
(let [1020] (nor2 [1031] readML_4<0> ))
(let ibWrite_4<0> (nor2 flush_CV2<0> [1020] ))
(let [1116] (inv1 FET_State_C1<0> ))
(let [1120] (nand4 [1112] phi4<0> [1116] FET_State_C1<1> ))
(let [1021] (nor2 flush_CV2<0> blockMiss_C2<0> ))
(let [1218] (inv1 [1021] ))
(let [1123] (nand2 [1218] LatchNotSpd_C1<0> ))
```

```
 (let ibTagWr_4<0> (nor4 [1120] [1123] resetIB_CV3<0> LatchWrKpsw_C2<0> ))
 (let [1023] (nor2 flush_CV2<0> ibMiss_C2<0> ))
 (let [1114] (inv1 FET_State_C1<1> ))
 (let [1029] (nor3 [1023] FET_State_C1<2> [1114] ))
 (let [1030] (nor2 FET_State_C1<1> [1112] ))
 (let [1022] (nor2 [1029] [1030] ))
 (let [1228] (nand2 [1106] cacheBusy_C3<0> ))
 (let [1179] (inv1 busSBusy_C2<0> ))
 (let [1127] (nand3 [1228] [1179] phi4<0> ))
 (let fetPCToBusS_4<0> (nor3 [1022] FET_State_C1<0> [1127] ))
 (let [1108] (inv1 PF_State_C1<2> ))
 (let [1131] (inv1 ldIBrefPC_4<0> ))
 (let ibINCToBusS_4<0>
     (nor4 flush_CV2<0> resetIB_CV3<0> startingPF_CV3<0> [1131] ))
 (let [1285] (nand3 [1293] phi4<0> dataValid_C3<0> ))
 (let [1289] (nand3 [1114] phi4<0> FET_State_C1<2> ))
 (let [1293] (nor3 PF_State_C1<0> PF_State_C1<1> [1108] ))
 (let [1297] (nor3 [1300] PF_State_C1<2> [1109] ))
 (let [1300] (nor2 PF_State_C1<0> startingPF_CV3<0> ))
 (let readML_4<0> (nor3 [1289] [1106] [1116] ))
 (let [1307] (nor2 [1297] [1293] ))
 (let ldIBrefPC_4<0> (nor2 [1307] [1127] )))

(defcell racell
 (input
       (phi1<0> bool)
       (cacheBusy_C3<0> bool)
       (dataValid_C3<0> bool)
       (busSBusy_C2<0> bool)
       (LatchWrKpsw_C2<0> bool)
       (LatchReset_C2<0> bool))
 (output
       (memBusy_V4<0> bool)
       (random_clk<0> bool)
       (random_clk_L<0> bool))
 (local ([1644] bool)([1657] bool)([1646] bool)([1641] bool))
 (let [1644] (inv1 dataValid_C3<0> ))
 (let [1657] (nand2 [1644] cacheBusy_C3<0> ))
 (let [1646] (inv1 busSBusy_C2<0> ))
 (let memBusy_V4<0> (nand2 [1657] [1646] ))
 (let random_clk_L<0> (nor2 LatchWrKpsw_C2<0> LatchReset_C2<0> ))
 (let [1641] (inv1 phi1<0> ))
 (let random_clk<0> (nor2 random_clk_L<0> [1641] )))

(defcell pffsm
 (input
       (phi1 sync)
```

```
        (phi4 sync)
        (memBusy_CV4<0> bool)
        (resetIB_CV3<0> bool)
        (flush_CV2<0> bool)
        (LatchIuEn_C1<0> bool)
        (LatchIuPre_C1<0> bool)
        (LatchWrKpsw_C2<0> bool)
        (startingPF_CV3<0> bool))
(state
        (PF_STATE_LOGIC<2> bool)
        (PF_STATE_LOGIC<1> bool)
        (PF_STATE_LOGIC<0> bool))
(output
        (PF_State_C4<2> bool)
        (PF_State_C4<1> bool)
        (PF_State_C4<0> bool)
        (PF_State_C1<2> bool)
        (PF_State_C1<1> bool)
        (PF_State_C1<0> bool))
(local
        ([1448] bool)([1447] bool)([1449] bool)([1526] bool)
        ([1502] bool)([1516] bool)([1433] bool)([1452] bool)
        ([1476] bool)([1458] bool)([1450] bool)([1445] bool)
        ([1387] bool)([1388] bool)([1313] bool)([1514] bool)
        ([1399] bool)([1518] bool)([1520] bool))
(let [1448] (inv1 PF_State_C1<2> ))
(let [1526] (nand3 PF_State_C1<0> [1448] PF_State_C1<1> ))
(let [1447] (inv1 PF_State_C1<0> ))
(let [1449] (inv1 PF_State_C1<1> ))
(let [1502] (nand3 [1447] [1449] PF_State_C1<2> ))
(let [1516] (nand2 [1526] [1502] ))
(let [1433] (nor2 flush_CV2<0> LatchWrKpsw_C2<0> ))
(let [1452] (nand2 [1516] [1433] ))
(let PF_STATE_LOGIC<2>
    (nor4 memBusy_CV4<0> resetIB_CV3<0> startingPF_CV3<0> [1452] ))
(let [1476] (inv1 LatchWrKpsw_C2<0> ))
(let [1458] (nand3 [1476] LatchIuEn_C1<0> LatchIuPre_C1<0> ))
(let [1450] (nand2 [1448] PF_State_C1<1> ))
(let [1445] (inv1 startingPF_CV3<0> ))
(let [1387] (nor4 [1458] [1450] PF_State_C1<0> [1445] ))
(let [1388] (nor2 PF_STATE_LOGIC<2> [1452] ))
(let [1313] (nor3 resetIB_CV3<0> [1387] [1388] ))
(let PF_STATE_LOGIC<0> (inv1 [1313] ))
(let [1518] (nand4 [1448] [1449] [1313] PF_State_C1<0> ))
(let [1514] (nand2 [1502] [1450] ))
(let [1399] (nor2 resetIB_CV3<0> PF_STATE_LOGIC<2> ))
(let [1520] (nand2 [1514] [1399] ))
```

```
(let PF_STATE_LOGIC<1> (nand2 [1518] [1520] ))
(let PF_State_C4<2> (prev phi4 PF_STATE_LOGIC<2>))
(let PF_State_C4<1> (prev phi4 PF_STATE_LOGIC<1>))
(let PF_State_C4<0> (prev phi4 PF_STATE_LOGIC<0>))
(let PF_State_C1<2> (prev phi1 PF_State_C4<2>))
(let PF_State_C1<1> (prev phi1 PF_State_C4<1>))
(let PF_State_C1<0> (prev phi1 PF_State_C4<0>)))
```