The Dissertation Committee for Narayanan Krishnamurthy
certifies that this is the approved version of the following dissertation:

A Design Validation Methodology for High Performance Microprocessors

Committee:

Jacob A. Abraham, Supervisor

Nur A. Touba

Margarida F. Jacome

Doug Burger

Magdy S. Abadir

Andrew K. Martin

A Design Validation Methodology for High Performance Microprocessors

by

Narayanan Krishnamurthy, B.Tech (Hons.), M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2003

Dedicated to my family

Acknowledgments

I would like to thank my entire family for supporting me throughout my years in school. This would not have been possible without their sacrifice and unlimited patience. Sowmya has always been there for me and her continuous encouragement and pragmatism is what enabled me to go this distance. Sahitya's cute smiles and Sahana's Tamil/English babble had a therapeutic effect during my most stressful days. My sister and her family, and my parents have always egged me on and their optimism and confidence in me through the years was a key factor that enabled completion of this work.

My sincere thanks to Jacob for agreeing to supervise my research work, for poking holes when I thought I had it all, and for motivating me to strive for the best. Andy and Magdy have both been invaluable mentors throughout this process. I would like to thank Andy for his cooperation and technical assistance during the various stages of this research work. Thanks to Magdy for all the technical discussions and brainstorming done in his office, mostly on very short notice.

My friends Nelson and Ann have been motivating factors whenever I've felt down in the dumps and their uplifting words have encouraged me to stay focussed. Nelson's song-writing poetry and his sense of humour, Julian's guitar artwork, and Sofie's Hat gigs were the meditative equivalents that helped me stay the course. Robi and Sangeeta's support for me during my initial days at UT helped me get started. I would like to

thank Jayanta, Dinos, Ashok, Bogdan, Jing, Gary, and Nancy for their support and perspective during this endeavour. I would like to thank my parents-in-law for all their help and encouragement during the final stages of my dissertation.

My heartfelt thanks to the High Performance Methodology and Tools group and custom circuit designers at Motorola for giving me the opportunity to work with them during my doctoral stint. I would like to thank Hershey who I know was quietly encouraging me to get this done so that he could get his daily walks. Finally, thanks to my kid brother Jairam for wishing me well from heaven.

A Design Validation Methodology for High Performance Microprocessors

Publication No. _____

Narayanan Krishnamurthy, Ph.D.
The University of Texas at Austin, 2003

Supervisor: Jacob A. Abraham

The task of checking whether a circuit implementation satisfies an abstract specification, prior to manufacturing the circuit, is extremely important. This is because of the reliance on the abstract specification being predictive of silicon behavior. It is also important to know the exact conditions under which the prediction is guaranteed to be valid.

This dissertation delves into the fundamental bottlenecks and issues in model extraction and the inherent difficulties in verifying equivalence of transistor circuit implementations with respect to higher-level specifications. A novel implementation verification methodology that is based on symbolic simulation is presented. In addition, the dissertation presents the general theory of automatic constraint generation that is required for a sound verification strategy and proposes an enhanced implementation verification methodology to eliminate gate/switch-level full-chip simulations. The practical aspects of developing a tool to dovetail into this methodology is also presented.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Microprocessor design cycle times have drastically reduced and the pressure on companies to meet stringent time-to-market requirements has risen. Verification and design analysis are two major components of this cycle time. Any effort that improves the effectiveness of verification and the quality of the design is crucial to meeting customer deadlines and requirements.

Designs go through a series of phases during the design process before they result in a product. As designs grow in complexity, multi-phase verification is required before delivering the product to the market place. As we trace back from the final manufacturing stage towards previous phases of the design process, validation is performed at each of these phases. The design of a system is an iteration of *specification* and *implementation* steps, performed either top-down or bottom-up. The implementation at each level can be thought of as a specification for the next level. The *correctness* of an implementation is not an autonomous concept but is rather a relation between a specification and an implementation. Design validation techniques attempt to establish a mathematical relation between the two entities. There are a number of approaches to design validation. No matter how they are categorized, the ultimate objective of these different approaches is to ensure that the final product satisfies cus-

tomer requirements and does not fail during operation. The cost of failure is becoming unacceptably high and in some cases may even lead to loss of life or the disruption of economic and commercial activities.

Design validation techniques can be broadly categorized into simulation based approaches and formal techniques. Simulation is still the state-of-the-art technique for validating the correctness of a complete design. Due to increased complexity of the designs, validation using only traditional scalar simulation cannot be exhaustive and has proved to be ineffective in exposing the hard-to-find bugs. This is because of the combinatorial complexity of the number of states and input sequences possible for a non-trivial design. Due to the non-exhaustive finite set of patterns simulated, there have been numerous instances where errors have been discovered late in the design cycle. Sometimes, errors have occurred even after the commercial production and marketing of a product has taken place [98].

Formal techniques on the other hand do an exhaustive analysis of the design but can check only small designs *completely*. Formal verification is akin to a mathematical proof and therefore the consideration of *all possible cases* is implicit in a formal verification methodology [52][64][85]. It establishes a mathematical relation between two different representations of a system. The nature of this mathematical relation varies according to the kind of validation approach. Formal techniques, when they are applicable, can be used to establish universal properties about the design independent of any particular set of inputs. As the sizes and complexity of the designs keep growing, formal validation techniques suffer from the state explosion problem [86]. Unless, drastic innovations in data structures

and proof systems come about, validation methodologies purely based on formal methods are currently infeasible and uneconomical.

Symbolic simulation has proved to be an efficient technique, bridging the gap between traditional simulation and full-fledged formal verification. Original work on symbolic simulation was directed towards formal circuit verification and this is still the major application for symbolic simulation.

1.1 Scope of the Thesis

This dissertation presents a novel verification methodology based on symbolic simulation. We analyze the fundamental bottlenecks of some of the existing formalisms used for modeling behavioral aspects of the structure under investigation. We show how these drawbacks play a significant role in deciding the verification methodology and strategy. One of the goals of this dissertation is to educate and help the reader develop an appreciation of the nature and capabilities as well as the drawbacks of a symbolic simulation verification methodology. The work described here provides a uniform sound framework in which to carry out the verification.

This dissertation does not address the problem of whether the specification means what it is intended to mean (design verification) and is not about deriving a new formalism/logic or formal theory of verification. Formalism must not be confused with rigor and excessive preoccupation with formalism can obscure the simplicity of the ideas involved [55][88]. This dissertation brings in formalism only when it is certain that it adds precision and clarity to the arguments and results.

The principal contributions of this work are detailed below:

- We show how existing boolean equivalence checking technology is inadequate for self-timed circuits, and establish the duality of *time progress* in simulation and *states* in formal verification. We present an algorithm for generating delay state machine models that enable data-independent symbolic simulations.

- A rigorous symbolic simulation methodology for implementation verification is established with emphasis on custom logic and custom memory circuit structures. A satisfaction relation between a switch-level model (extracted from the circuit implementation) and a Register Transfer Language (RTL) model (extracted from the abstract specification) for *all modes* of operation (functional and non-functional) is established.

- A method for the automatic derivation of constraints is developed and its integration into the implementation verification framework is undertaken. A constraints guarantee methodology is developed and a rigorous methodology for dynamic circuit verification is illustrated.

- An analysis of the product machine approach versus the state mapping approach to custom memory verification in scan-based designs is made in terms of coverage and complexity. A new category of logic bugs is introduced in implementation verification that has traditionally been ignored.

- An enhanced equivalence checking flow is proposed to completely eliminate gate/switch-level simulations from the design flows.

The ultimate objective of this work is to establish a sound implementation verification methodology and to derive the specific technology requirements to enable that. Before we discuss the verification methodology, we discuss some of the existing verification techniques and formalisms that are commonly used in hardware verification.

1.2 Background and Related Work

Errors uncovered by different validation approaches are classified as design errors or implementation errors. Design errors occur when a specification itself has errors in it. Usually, the specification is at the behavioral level and is checked for correctness with respect to a set of properties. This is termed *design verification*. A correct specification is one for which the properties hold. Implementation errors are errors that occur in the realization (implementation) of the specification. Here, the specification is assumed to encode the correct set of behaviors of the design while the implementation is checked for errors. This is termed *implementation verification*. The work presented in this dissertation seeks to uncover implementation errors by establishing a mathematical relation of satisfaction between the specification and the implementation using symbolic simulation.

Any validation approach is prone to two types of errors. A *false-positive* error occurs when the validation technique predicts that an incorrect implementation is correct. A *false-negative* error occurs when the validation technique predicts that a correct implementation is incorrect. Clearly, false positives must be avoided (or else the methodology is useless) and false negatives must be minimized. The desire to produce more reliable

products free of design errors and the need to reduce time-to-market (for the same product quality) has resulted in a number of validation methodologies. A brief summary of these techniques is given below.

1.2.1 Simulation

The methodology used in the microprocessor and digital systems industry to validate designs has traditionally been simulation. Farms of workstations run simulations for months on the RTL and switch-level models (models that represent transistors as on-off switches). The product is taped out when the rate of discovering errors has dropped below a predetermined value.

Typically, the models are simulated using random and weighted random instruction sequences and their outputs and states are compared with the expected values obtained from the reference model. In case of a discrepancy, the model is then debugged to understand the cause of failure. Due to the large combination of possible initial state and input sequences, it is infeasible to exhaustively cover all cases in practice. This being said, even today, simulation is relied upon to catch the last (or thought to be) pretapeout bugs simply because there are no other techniques that can handle these large designs and that can fit into existing design methodologies seamlessly.

With the introduction of the ternary value X [47], it has been possible to simulate larger designs with fewer simulation vectors resulting in fewer simulation runs. Ternary valued logic simulators use X as an unknown digital value (either 0 or 1) to set up the initial state of the circuit prior to the start of simulation. If the simulation model is monotonic with

respect to the information ordering of the ternary values, a ternary valued simulator can verify circuit behavior for many possible input and initial state combinations [17]. If a simulation of a vector containing X's yields 0 or 1 on a node in the circuit, it is guaranteed that the value on the node will not change if the X's in the vector were replaced by any combination of 0's and 1's. Bryant proved the correctness of a static RAM design by logic simulation using ternary values [22]. For performance reasons, most ternary simulators are pessimistic and will produce an X even though it can be proved that the circuit produces a 0 or a 1 in all cases. As a result, a ternary logic verifier can produce many false-negatives and the debug and re-runs can be quite time-consuming. However, this does not compromise the rigor of the verification.

Automatic Test Pattern Generation (ATPG), SAT-based, and hybrid approaches have been used in the area of property verification [94][101] and combinational equivalence checking [15][50][95]. Because most ATPG and SAT tools work on gate-level models, they have not been applied to implementation verification of custom circuits.

1.2.2 Model Checking

Model checking is a verification technique in which properties are expressed in a temporal logic and the design is modeled as a state transition system or Kripke structure [67]. A design is verified against such a temporal logic specification by proving that the design is a model of the specification formula. The introduction of temporal logic model checking algorithms by Clarke and Emerson [35][36] allowed this approach to be automated. McMillan [86] developed the symbolic version and Coudert [39] developed

a symbolic model checker for a restricted class of formulas.

This approach relies on modeling the design as a state transition system and then checking to see if the formula holds on the transition system. The temporal logic specification describes the ordering of events in time without introducing time explicitly. These logics were originally developed by philosophers for investigating the way that time is used in natural language arguments [58]. Pnueli was one of the first to use temporal logic for reasoning about concurrent programs [97]. Temporal logics are often classified according to whether time is assumed to have a *linear* or a *branching* structure. Different logics characterizing various models of time can be found in [30].

This approach has several drawbacks. Most realistic systems are still too large to be model checked. This is primarily due to the size of the transition relations that need to be built. Moreover, for self-timed circuits, an erroneous transition relation would be built. To avoid this, the model checker would have to work with an extracted state transition system whose transitions occur according to the timing of the underlying switch-level model. Since the granularity of time in a switch-level model is finer than that of a higher-level description (or in other words, the number of distinct nodes in the circuit that can be treated as state variables is larger than the number of state variables in the higher-level description), there is an explosion in the number of states that the model checker has to contend with. In subsequent chapters, the dissertation delves further into the notion of modeling time in simulation as opposed to state space reachability analysis. Even though state explosion is a fundamental drawback of the model checking technique, it is extremely good at proving safety

(*nothing bad will ever happen*) and liveness (*something good will eventually happen*) properties of high-level descriptions of system behavior.

1.2.3 Theorem Proving

To be amenable to mathematical proof, both the specification of the system and the model of its actual behavior need to be stated mathematically. Certain formal procedures in a specific deductive system are then used to operate on the symbolic statements. Theorem proving works within a framework of logic, with axioms representing known truths about the behavior of hardware and theorems representing newly inferred properties of the system's behavior.

Two of the most common hardware verification approaches based on theorem proving are first-order logic using the ACL2 proof system [14] and higher-order logic using Gordon's HOL system [51] or other approaches such as PVS [90]. In the ACL2/Boyer-Moore system, sequential behavior is described by writing recursive functions. The function has parameters representing system state and time. Each recursive invocation of the function updates the state and time parameters. A proof of equivalence would involve showing the equivalence of two such functions in the specification and implementation respectively. Since the specification function may have less state, it would be a non-trivial proof procedure to establish this. Also, sometimes it is necessary to guide the prover through a series of lemmas which can demand intimate understanding of the theorem prover.

The HOL system is based on simple type theory and is a strongly-typed lambda calculus. The proof is based on manipulation of higher-order objects such as functions which operate on other functions. Time-varying

signals can be modeled as functions from integers to signal values. Predicates over signals can then be used to describe hardware elements. Behavior can then be described using logical predicates and proving correctness would involve showing that a conjunction of logical predicates (which describes the implementation) implies another conjunction of logical predicates (which describes the specification).

The main disadvantage of theorem proving is that complete automation is extremely hard. Automated theorem proving requires fairly general heuristics but many heuristics are domain-specific. The heuristics are imposed on some of the underlying proof mechanisms. While in a problem domain such as propositional tautology, complete automation is possible, even validity in first order logic is not decidable and may require an arbitrarily long search. As theorems get more complicated, the time that the theorem provers spend tends to grow exponentially and therefore the solution is typically to provide the theorem provers with heuristics. The development of good heuristics is a major area of research and requires much experience and insight. Though based on a foundation of mathematical rigor, there may be bugs in the inference rules which are hard to detect and may lead to false positives. Current hardware description languages such as Verilog and VHDL do not have formalized semantics and rely on the user to embed their semantics in the proof-system's logic. It is not clear whether such proof-systems can be automated and applied to problems of real practical significance and whether they can fit smoothly into existing design methodologies.

1.2.4 Comparison of State Machines

Another approach to verification is the comparison between the specification and the implementation state machines. Equivalence is checked by forming the product machine and then searching the state space for reachable states at which the outputs differ. Here, the equivalence criteria is input-output behavior with each machine starting in a known state. Work in this area has utilized symbolic representations such as BDD's [40][45][106]. Pixley relaxed the need for known starting states but the technique is capable of handling only small designs [96]. As a result of the state explosion of the resulting product state machine graph, this technique can only be used on relatively small designs. Also, if there are a number of initial states, then the search has to be performed for each one of them.

The shortcoming of this approach is the state explosion of the resulting product state machine graph. Non-determinism in the specification and implementation further exacerbates the state-explosion problem.

1.2.5 Language and Trace-based Theories

These techniques are based on inclusion relations between sets of traces or sets of strings, where the sets of traces or strings are used to model system behavior. Dill [46] developed a trace theory for the verification of speed-independent asynchronous circuits. Burch [28] generalized the mathematics of trace algebra and trace structures and introduced the idea of conservative approximations from continuous time models to discrete time models. Martin [81] introduced the trace-automata formalism and investigated its use as an abstraction to verify hybrid systems. Kurshan [77][78] uses language homomorphisms on automata to reduce the language con-

tainment verification problems to simpler problems. These techniques have primarily been applied to design verification of RTL and higher-level behavioral models. They have had limited success with transistor-level circuits for implementation verification.

1.2.6 Boolean equivalence checking

Boolean equivalence checking tools such as LEC (from Verplex), Formality (from Synopsys), Affirma (from Cadence Design Systems), Design VERIFYer (from Chrysalis), MET (from Motorola), and Verity (from IBM) are used routinely to verify equivalence between RTL, gate, and switch level models of standard library cells and custom designed circuits. These tools operate by extracting a boolean function from these descriptions and performing a logical equivalence check. This check is performed by comparing the boolean functions implemented by corresponding static cones of logic in the two circuit models that are being compared. It is the stable outputs of the logic cones that are compared. Singh *et al* [105] have done equivalence checking by extracting RTL models from transistor netlists and then doing the comparison. They rely on manipulating the simulation relation obtained to derive the stable behavior of the circuit. They also require the user to provide information about the clocking scheme employed in the transistor-level implementation.

However, as illustrated later in this dissertation, current boolean function extraction techniques that use functional composition to extract logic functions, or techniques that try to derive the stable behavior, will not suffice for a certain class of self-timed custom circuits.

1.2.7 Symbolic Simulation

Symbolic simulation is a simulation-based approach that combines traditional simulation with formal symbolic manipulation [5][6][7][23][24][38]. It has proved to be a practical and viable technique for validating behavioral, RTL and switch-level models [25][32][91][93][104]. The hybrid approach of combining formal techniques with simulation takes the best of both worlds. It combines switch-level model accuracy with formalized reasoning to infer properties about the system. It executes a model by evaluating the model for multiple data values in a single simulation run. By encoding a range of scalar values using symbolic variables, a symbolic simulator can compute what would require many runs of a traditional simulator [24]. Researchers at IBM [32][44] first introduced the term symbolic simulation to reason about properties of circuits described at the RTL-level. Their approach drew on techniques developed for reasoning about software by symbolic execution. Darringer [44] showed how to apply symbolic execution to combinational logic verification, by building a gate-level simulator and then simplifying the equations it implements. The D-algorithm [100] for test generation can be thought of as a form of symbolic simulation where the functions of the basic logic gates were extended to operate over an expanded set of values *0, 1, D and* $\bar{D}$.

The success of this early work was limited by the weakness of the symbolic manipulation methods. With the advent of BDD's [20], the technique became much more practical. Priam [80], Bull's industrial tool, performs equivalence checking by symbolic execution of two descriptions that have the same state encoding and same timing. Bryant's work [16][17][22][23][25] in the area of switch-level and memory symbolic

simulation and Seger's work on symbolic trajectory evaluation [103][104] renewed further interest in symbolic execution. Jain [61] examined the use of parametric Boolean formulas in symbolic simulation. Symbolic simulation has been used to verify embedded on-chip memories of commercial microprocessors [49][91][93]. Their work focussed on establishing that functional properties held for two different representations but no notion of completeness was established by their methodology. There has been work in the area of symbolic timing verification and development of techniques for symbolic delay modeling [66][82][83][84] as an alternative to static timing analysis. These techniques rely on a fairly simplistic Elmore delay model for delay computation and face the problem of event multiplication due to data-dependent event scheduling. Recently, generalized symbolic trajectory evaluation [120] was introduced for verifying ω-regular properties. In this dissertation, our main objective is in using symbolic simulation for developing an automated implementation verification methodology for custom circuits.

Chapter 2

Custom Logic Verification Methodology

A system may be specified at various levels of abstraction. These levels range from high-level abstract property specifications to fine-grained descriptions of circuit implementations in terms of differential equations. This dissertation addresses the implementation verification problem of proving that a custom logic transistor circuit is a correct implementation of a higher-level specification. This is critical to building a product since the implicit assumption is that the higher-level specification is predictive of the implementation behavior.

There are two main approaches for implementation verification. The first approach is concerned with performing combinational logic equivalence checking of the two circuits. The circuits are typically at the gate-level and the assumption is that the two circuits have identical state encodings. Some of the tools that do combinational equivalence checking are LEC (from Verplex), Formality (from Synopsys), Affirma (from Cadence Design Systems), Design VERIFYer (from Chrysalis), MET (from Motorola), and Verity (from IBM). The second approach involves specifying *desired* properties and proving that they are satisfied by the circuit implementation. Verification amounts to showing that each possible behavior of the implementation satisfies the specification property. Some of the verification techniques using the second approach are model checking,

theorem proving, and language containment. These techniques work well with gate-level models and at higher levels of abstraction and are not well suited for transistor circuits.

The symbolic simulation verification methodology described in this dissertation is an improvement over the above two approaches because

- A *complete* set of properties is automatically extracted from the behavioral specification.
- It can deal with self-timed circuit implementations where correct implementation behavior is dependent on dynamic transient states.
- The circuit implementation can have any arbitrary transistor structure and the methodology does not rely on pattern matching.
- The state encodings between the specification and the implementation may be different.
- It can deal with extra primary inputs in the implementation.
- The methodology allows selective switch-level model decomposition based on the property being checked and the ability to generate different switch-level models based on transistor resistance ratios and/or verilog strength attributes.
- Both the manufacturing test and functional modes of operation are checked thereby eliminating *crossover* bugs.
- The constraints modeling the environment are automatically extracted for verification and tied back into the verification methodology.

- Enables the elimination of verilog gate and switch-level functional simulations by having the ability to verify verilog switch-level semantics of the circuit implementation.

In the next section, we discuss some of the abstract models used for design analysis and for modeling hardware behavior. We present the case for why certain modeling paradigms are more suited to implementation behavior verification of custom and semi-custom logic circuits. We show how all these formalisms are based on a discrete time model and that the notion of simulation time, computation paths, and state sequences are equivalent formulations of computation runs or simulation progress. We introduce a state-based framework for modeling delays in circuit implementations. This allows the bridging of delay-based simulation approaches with traditional state space exploration techniques. We provide a systematic approach for integration into an event-driven simulation methodology for verifying custom logic implementations.

2.1 Model Abstraction and Simulation Semantics

Throughout this dissertation, we will be referring to the term *model*. By a *model*, we mean the mathematical abstractions of the specification and/or the implementation that provides a tractable and fairly accurate approximation of physical reality. Both simulation and formal verification rely on models to mimic and simulate real behavior of hardware designs. This section describes the formalisms and models commonly used to model hardware and establishes the duality between state-space formalisms and simulation time-progress semantics. Proof of the equivalence of two or more

formalisms has a compelling effect when these notions arise independently and from different application domains. We show why our verification methodology does not face the complexity issues that state-space exploration methods do. The following questions are raised and answered in subsequent sections.

- What is the basis for identifying certain elements in the hardware implementation as state and how does this affect any verification technique based on state reachability analysis?
- What does progress of time mean when verifying models derived from hardware implementations and what does time progress mean in our verification methodology?
- Can properties that involve time be cast in terms of properties about computation paths or state sequences?
- Can the same hardware circuit implementation lead to different finite-state discrete models thereby differentiating the properties that can be proved on them? In such a case, what are the kinds of properties that do not get proved using one vs the other?

Definition 2.1.1. Consider a finite set of elements C. Let $\mathcal{U}$ be a finite set of variables that range over the elements of the domain C. A valuation is a mapping $\mathcal{I} : \mathcal{U} \rightarrow C$ which assigns to each variable $u_i \in \mathcal{U}$, a value $c_i \in C$. Let $\mathcal{I}^{\mathcal{U}}$ be the set of all valuations and let $\mathcal{I}_j^{\mathcal{U}}$ denote the jth valuation such that $\mathcal{I}_j^{\mathcal{U}} \in \mathcal{I}^{\mathcal{U}}$. We will write $\mathcal{I}_j$ when the set of variables is clear from the context. If $f : C^n \rightarrow C$, then $f(\mathcal{I}_j) \in C \ : \mathcal{I}_j = \{u_0 \rightarrow c_0, u_1 \rightarrow c_1, \ldots, u_{n-1} \rightarrow c_{n-1}\}$ and for all values of i, c_i is in C.

If the domain of interest is binary values $\mathcal{B} = \{0, 1\}$ and $\mathcal{V}$ is a finite set of variables of type $\mathcal{B}$, then $\mathcal{I} : \mathcal{V} \rightarrow \mathcal{B}$ is a valuation which assigns to each variable $x \in \mathcal{V}$ the value 0 or 1. By the earlier definition, $\mathcal{I}^{\mathcal{V}}$ is then the set of all valuations and $\mathcal{I}_j^{\mathcal{V}}$ denotes the jth valuation such that $\mathcal{I}_j^{\mathcal{V}} \in \mathcal{I}^{\mathcal{V}}$. A Boolean function is a mapping $f : \mathcal{B}^n \rightarrow \mathcal{B}$ which assigns a result of type $\mathcal{B}$ to each valuation of all n variables. A variable $v \in \mathcal{V}$ is also known as a boolean variable. A boolean variable v_i with the value 0 is represented by $\bar{v_i}$ and a variable v_i with the value 1 is represented by v_i. A similar definition for propositional formulas exists if $\mathcal{B} = \{F, T\}$. Note that $\mathcal{I}^{\mathcal{V}}$ can also be viewed as representing all the entries in a truth table.

Definition 2.1.2. A finite-state machine or finite automaton M is a 6-tuple [65][89] $M = (I, O, S, \delta, \lambda, S_0)$, where $I(O)$ is the input (output) space, S is the state space, δ (λ) is the next state (output) relation and S_0 is the set of initial states. If $\mathcal{B}$ is the set $\{0, 1\}$, then

- State space or state alphabet S is a power of $\mathcal{B}$ that represents the states of the machine. If n is the number of state variables and $S_v = \{s_0, s_1, \ldots, s_{n-1}\}$ are the corresponding state variables, then $S = \mathcal{B}^n$.

- Input space or input alphabet I is a power of $\mathcal{B}$ that represents the input alphabet of the machine. If m is the number of inputs, and $I_v = \{i_0, i_1, \ldots, i_{m-1}\}$ are the corresponding input variables, then $I = \mathcal{B}^m$.

- Output space or output alphabet O is a power of $\mathcal{B}$ that represents the output alphabet of the machine. If k is the number of outputs,

and $O_v = \{o_0, o_1, \ldots, o_{k-1}\}$ are the corresponding output variables, then $O = \mathcal{B}^k$.

- Next-state function is $\delta : S \times I \rightarrow S$, where $\delta = \{\delta_0, \delta_1, \ldots, \delta_{n-1}\}$ and $\delta_i : S \times I \rightarrow \mathcal{B}$ is the next-state transition function of the state variable s_i. For non-deterministic machines, δ is a relation and can be defined as a set-valued function $\delta : S \times I \rightarrow \mathcal{P}(S)$. In the general case, δ is a relation from powers of $\mathcal{B}$ to powers of $\mathcal{B}$.

- Output function is $\lambda : S \times I \rightarrow O$, where $\lambda = \{\lambda_0, \lambda_1, \ldots, \lambda_{k-1}\}$ and $\lambda_i : S \times I \rightarrow \mathcal{B}$ is the output function associated with the variable o_i. A machine possessing this property of the output function is known as a Mealy machine [87]. When the output function λ is defined as $\lambda : S \rightarrow O$, and the output is determined only by the current state i.e., the inputs are not considered, then the machine is known as a Moore machine. In the general case, λ is a relation from powers of $\mathcal{B}$ to powers of $\mathcal{B}$.

- $S_0 \subseteq S$ represents the set of initial states of the machine

A Moore or Mealy state machine can be considered as an automaton where all the machine states are accepting. The terms automata and machines will be used interchangeably throughout this dissertation unless a distinction is warranted in the context of the discussion. See Figure 2.1 for a Mealy and Moore machine representation of a sequential circuit. This representation of a circuit is based on the Huffman model of a sequential circuit where a circuit is partitioned into combinational logic and state elements [57]. Every Mealy machine has a Moore machine counterpart and there is an established procedure for transforming a Mealy machine into

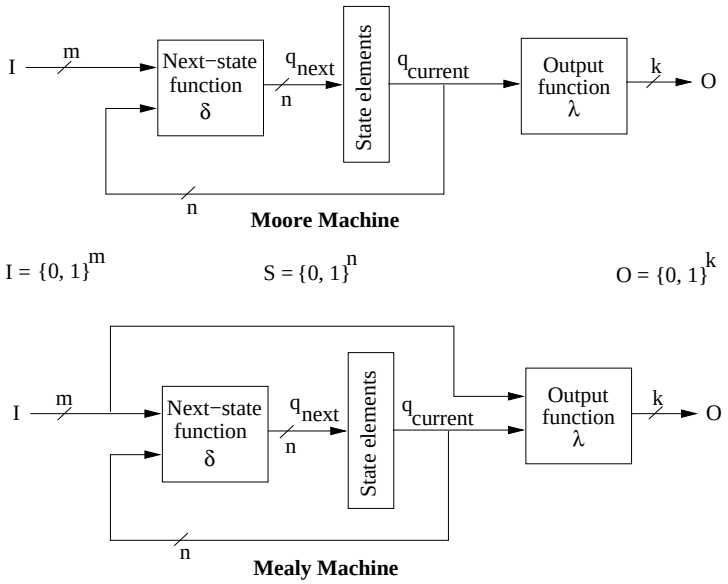


Figure 2.1: A Moore and Mealy representation of a sequential circuit

a corresponding Moore machine having the same terminal input-output behavior [55][65]. In the automata theoretic framework, this would mean that for every Mealy automaton, there exists a corresponding Moore automata that recognizes exactly the same set of input sequences. It is easier to determine whether or not a Moore machine accepts an input sequence by computing which state the sequence leaves the machine in. Since the outputs are directly associated with the states, we will assume the existence of a Moore machine for our discussions, without any loss of generality.

Machine M in Definition 2.1.2 can also be viewed as a function from finite input sequences to finite output sequences. The behavior of a finite-state machine is described as a sequence of events that occur at discrete instants of time. These machines do not have infinite storage capabilities and their past histories can affect their future behavior in only

a finite number of ways. These models can only distinguish between a finite number of classes of input histories thereby leading to equivalence classes of input histories. Each equivalence class is an *internal state* of the machine.

Definition 2.1.3. A machine internal state $q_i \in S$ is represented by a unique valuation function $\mathfrak{I}_{q_i} : S_v \rightarrow \mathcal{B}$. Any state of the machine is binary encoded into some valuation of the state variables of the model. The encoding from the set of machine states to the set of valuations is injective. For n state variables, there exists exactly 2^n valuations $\{q_0, q_1, \ldots, q_{(2^n-1)}\}$. Two states q_i and q_j are equal if they are represented by the same valuation, i.e. $\mathfrak{I}_{q_i} = \mathfrak{I}_{q_j}$.

Definition 2.1.4. Let $S_c \subseteq S_v$. Using Definition 2.1.3, the partial valuation function derived from the valuation function $\mathfrak{I}_{q_i}$ with respect to the restricting set S_c is given by

$$q_i \downarrow_{S_c} = \{y \mid y = \mathfrak{I}_{q_i}(x) \text{ and } x \in S_c\}$$

We term this partial function $q_i \downarrow_{S_c}$ the *projected* machine state q_i with respect to a set of variables S_c. Given a set of states $Q = \{q_0, q_1, \ldots, q_{n-1}\}$, the projected state set is $Q \downarrow_{S_c} = \{q_0 \downarrow_{S_c}, q_1 \downarrow_{S_c}, \ldots, q_{n-1} \downarrow_{S_c}\}$.

Corollary 2.1.1. *If q_i and q_j represent two machine states, then*

$$\forall i, j.\ q_i \equiv q_j \Rightarrow q_i \downarrow_{S_c} \equiv q_j \downarrow_{S_c}$$

but the converse relation does not hold

$$\forall i, j.\ q_i \downarrow_{S_c} \equiv q_j \downarrow_{S_c} \not\Rightarrow q_i \equiv q_j$$

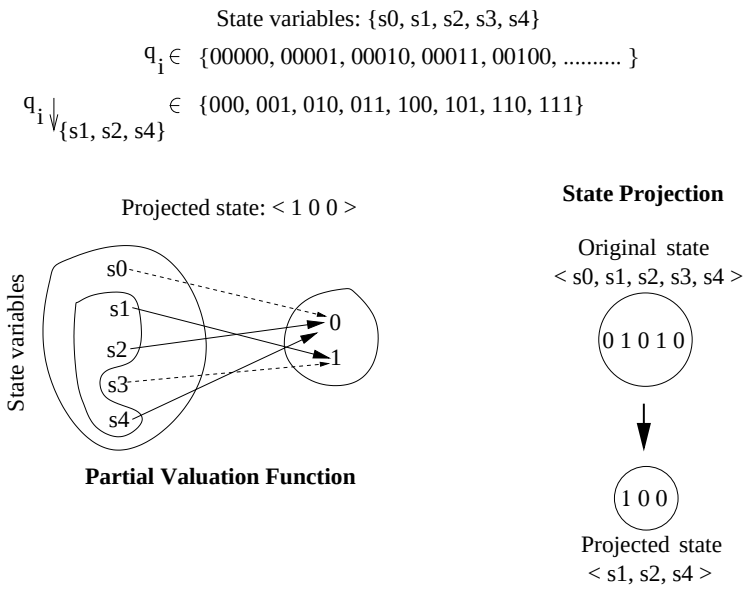


Figure 2.2: Projection of states

Assume a fixed order in the state variable specification. We will use the tuple notation $< s_0, s_1, \ldots >$ to represent a specific machine state q_i. See Figure 2.2 for an illustration of projected states.

Definition 2.1.5. The *smoothing* or existential quantification $\exists v$ of a boolean variable from a boolean function f is defined as $\exists v.\, f = f\,|_{\bar{v}} \vee f\,|_{v}$. For a vector of variables $\vec{w} = [w_0, w_1, \ldots, w_{n-1}]$, the existential quantification $\exists \vec{w}\, f$ is defined as $\exists w_0 \exists w_1 \ldots \exists w_{n-1} f$, where n is the length of the vector $\vec{w}$. Existential quantification of a variable can be viewed as removing the variable from further consideration. The *consensus* or universal quantification $\forall v$ of a boolean variable from a boolean function f is defined as $\forall v.\, f = f\,|_{\bar{v}} \wedge f\,|_{v}$. For a vector of variables $\vec{w}$, the universal quantification $\forall \vec{w}\, f$ is defined as $\forall w_0 \forall w_1 \ldots \forall w_{n-1} f$. The universal quantification of a variable can be viewed as representing the component of the function that

is independent of that variable.

Definition 2.1.6. Given a next-state function $\delta : S \times I \rightarrow S$ for a machine M, the next state relation or *transition relation* predicate associated with δ is $R_\delta : S \times I \times S \rightarrow \mathcal{B}$.

$$R_\delta(q_i, x, q_j) = \begin{cases} T & , \quad \text{if } q_j = \delta(q_i, x) \\ F & , \quad \text{otherwise} \end{cases}$$

A machine state q_j is said to be a successor of a machine state q_i if and only if $\exists x \in I.\ q_j = \delta(q_i, x)$. To get rid of the input values in R_δ, we carry out the existential quantification of all the input variables I_v from R_δ. We then obtain the *input-free* transition relation predicate $R(q_i, q_j) : S \times S \rightarrow \mathcal{B}$ for the machine M. The input-free transition relation predicate $R(q_i, q_j)$ is satisfied when state q_j is a possible successor of state q_i and is unsatisfied otherwise.

Definition 2.1.7. A state path or state sequence $\hat{q}$ is a possibly infinite sequence of machine states $(q_0, q_1, \ldots, q_{n-1}, \ldots)$ such that $\forall i \geq 0,\ R(q_i, q_{i+1}) = T$. For a finite state path, $\hat{q} = (q_0, q_1, \ldots, q_{n-1})$, where n is the length of the path. We use $\hat{q}^i$ to denote the suffix of $\hat{q}$ starting at q_i. Each element q_i in the sequence is associated with a time instant t_i. A state path or state sequence is also called a *trajectory*.

Definition 2.1.8. A *projected state path* or *projected state sequence* $\hat{q} \downarrow_V$ with respect to a set of variables V is a possibly infinite sequence of machine states $(q_0 \downarrow_V, q_1 \downarrow_V, \ldots, q_{n-1} \downarrow_V, \ldots)$ such that $\forall i \geq 0,\ R(q_i, q_{i+1}) = T$. For a finite projected state path, $\hat{q} \downarrow_V = (q_0 \downarrow_V, q_1 \downarrow_V, \ldots, q_{n-1} \downarrow_V)$, where n is the length of the path.

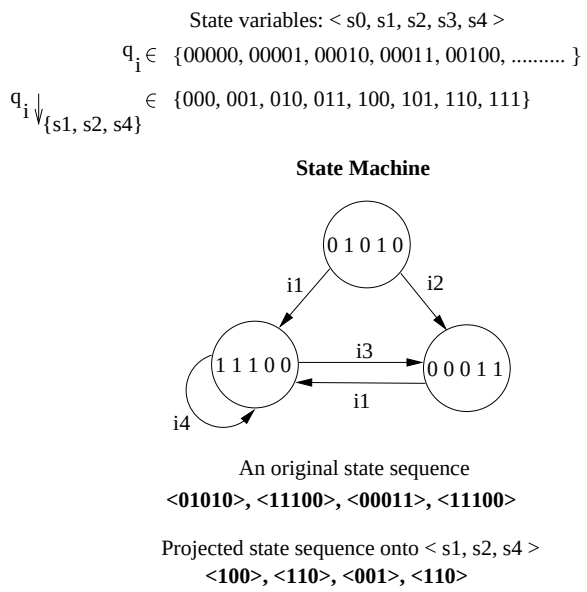


Figure 2.3: Projected state sequences

See Figure 2.3 for an illustration of projected states and projected state paths. The projected state sequence $(100, 110, 001, 110)$ is derived by projecting the original state sequence $(01010, 11100, 00011, 11100)$ onto the set of variables $\{s_1, s_2, s_4\}$. In general, if behavior is to be preserved, then only state sequences can be projected and not the state transition graph as a whole. This is because additional behavior can be derived from the newly projected machine model due to the introduction of additional transition edges from and to the projected states. Using Corollary 2.1.1, if two machine states q_i and q_j map to the same projected state $q_p = q_i \downarrow_V = q_j \downarrow_V$, then all outgoing transitions from q_i and q_j will now need to originate from the projected state q_p. All incoming transitions into q_i and q_j will need to be redirected into q_p to preserve the same behavior. However, this leads to the introduction of additional transitions and possibly non-deterministic transitions. In Figure 2.4, the projected state machine on the

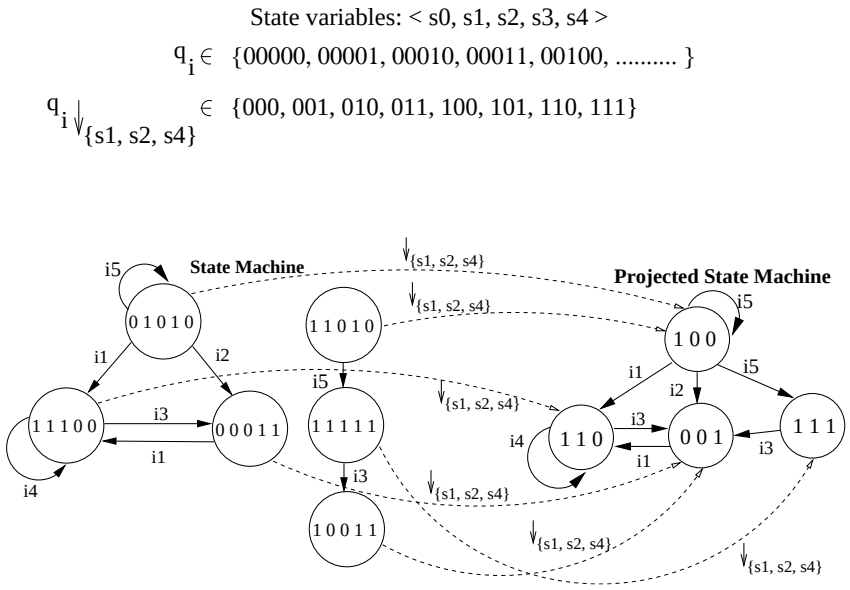


Figure 2.4: Additional behavior induced by projection

right hand side of the figure has a non-deterministic transition on input i_5 in projected state $< s_1 = 1, s_2 = 0, s_4 = 0 >$. Also, a new state sequence $(100, 111, 001, 110)$ is now a possible behavior which cannot be generated by the original state machine. This implies that only state sequences can be projected. Thus the transition relation cannot generally be projected without introducing new behaviors.

Definition 2.1.9. Let M be a Moore machine and let $\hat{x} = (x_0, x_1, \ldots, x_n)$ be a sequence over the input alphabet I (input string) such that $\hat{x} \in I^\star$. Let $\hat{y} = (y_0, y_1, \ldots, y_n)$ be a sequence over the output alphabet O (output string) such that $\hat{y} \in O^\star$. Let $\ddot{z} = (x_0 y_0, x_1 y_1, \ldots, x_n y_n)$ be the finite input-output sequence such that $\ddot{z} \in \{I \times O\}^\star$. M *simulates* $\ddot{z}$ if a state sequence

$\hat{q} = (q_0, q_1, \ldots, q_n)$ exists in S that satisfies the following conditions

$$\begin{aligned} q_0 &\in S_0 \\ \delta(q_i, x_i) &= q_{i+1} \qquad for\ i = 0, \ldots, (n-1) \\ \lambda(q_i) &= y_i \qquad for\ i = 0, \ldots, n \end{aligned}$$

If $\ddot{w}$ is an input-output sequence, the notation $M \models \ddot{w}$ means that M simulates $\ddot{w}$ and that $\ddot{w}$ is a possible behavior that could be exhibited by M.

Definition 2.1.10. Let E_v be the set of externally visible variables, i.e. the input and output variables, $\{e_0, e_1, \ldots, e_{p-1}\}$ associated with a Moore Machine M.

$$E_v = I_v \cup O_v = \{i_0, i_1, \ldots, i_{m-1}, o_0, o_1, \ldots, o_{k-1}\}$$

The number of externally visible variables for M is $p = m + k$.

Definition 2.1.11. A labeled state transition graph or system, also known as a Kripke or temporal structure [58][67][68], is a 4-tuple $K = (S, R, L, S_0)$ that consists of

- A finite set of states S
- A transition relation, $R \subseteq S \times S$ such that $\forall q \in S,\ \exists q' \in S,\ (q, q') \in R$
  (i.e. R is total)
- A labeling function $L : S \to \mathcal{P}(\mathcal{V})$ with $\mathcal{V}$ being the set of propositional variables (atomic formulas)

- A set of initial states $S_0 \subseteq S$

Kripke structures are the underlying models used in state reachability analysis and temporal logic model checking. Model checking is a verification technique that is based on state-reachability analysis of the underlying Kripke structure. The algorithm is based on computing fixpoints of predicate transformers that are obtained from the transition relation [29][37][86]. The transition relation is expressed as a boolean formula in terms of two sets of state variables, one set encoding the old state and the other encoding the next state. The fixpoints are sets of states that represent various temporal properties of the system. The existence of the fixpoints and its uniqueness is based on Tarski's theorem [109].

In model checking, specifications are expressed in a propositional temporal logic and circuit implementations and protocols are modeled as Kripke structures. Model checking automatically checks if the Kripke structure is a *model* for the specification. Temporal logic formulas, obtained from specifications, describe properties of computation trees that are derived from a Kripke structure K. A state is designated as the initial state from the set S_0 and then the structure is unwound into an infinite tree with the designated state at the root as shown in Figure 2.5. Beginning in this state, the temporal structure is traversed according to the successor states given by R. Temporal logics are often classified according to whether time is assumed to have a linear or a branching structure. For the purposes of this dissertation, this distinction is not relevant.

Definition 2.1.12. A state path or state sequence in a Kripke structure K is a finite sequence of states $\hat{\pi} = (q_0, q_1, \ldots, q_{n-1})$ or an infinite sequence of

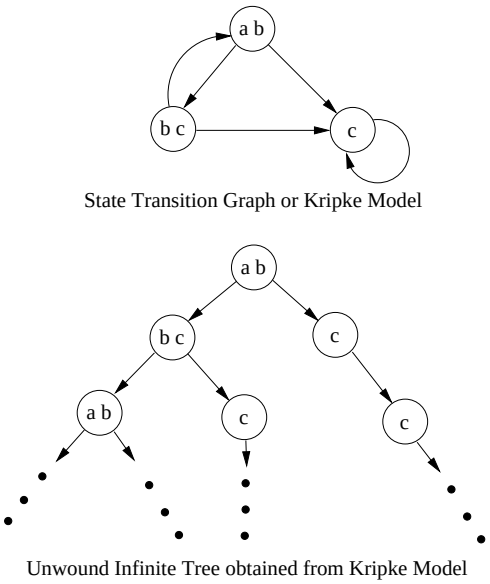


Figure 2.5: Computation trees from Kripke structures

states, $\hat{\pi} = (q_0, q_1, \dots)$ such that $\forall i \geq 0, (q_i, q_{i+1}) \in R$. We use $\hat{\pi}^i$ to denote the suffix of $\hat{\pi}$ starting at q_i. A *permissible* state sequence $\hat{\pi}_p^i$ additionally satisfies the restriction that $q_0 \in S_0$. A Kripke structure K simulates a sequence $\hat{\pi}$ i.e. $K \models \hat{\pi}$ if and only if $\hat{\pi}$ is a permissible state sequence. We consider only permissible state sequences $\hat{\pi}_p$.

Corollary 2.1.2. *Let S_v^K be the set of state variables given the state set S of Kripke structure K. The size n of the state variable set is $n = \left|S_v^K\right| = \lceil \log_2 |S| \rceil$.*

$$S_v^K = \{k_0, k_1, \dots, k_{n-1}\}$$

Proposition 2.1.3. *Any finite state Moore machine* $M = (I, O, S, \delta, \lambda, S_0)$ *can be converted to an equivalent temporal Kripke structure* $\acute{K} = (\acute{S}, \acute{R}, \acute{L}, \acute{S_0})$ *such that*

$$\begin{aligned} \forall \ddot{u} \in \{I \times O\}^{\star},\ M \models \ddot{u} \ &\Rightarrow\ \exists \hat{\pi}_p : K \models \hat{\pi}_p \text{ and } \hat{\pi}_p \downarrow_{(I_v \cup O_v)} \equiv \ddot{u} \\ \forall \hat{\pi}_p \downarrow_{(I_v \cup O_v)} \ &\Rightarrow\ \exists \ddot{u} \in \{I \times O\}^{\star},\ M \models \ddot{u} \end{aligned}$$

Proof: Construct $\acute{K}$ as follows

$$\begin{aligned} \acute{S} \subseteq I \times S \times O \quad &: \forall (\acute{\imath}, \acute{q}, \acute{o}) \in \acute{S},\ \lambda(\acute{q}) = \acute{o} \\ \acute{R} \subseteq \acute{S} \times \acute{S} \quad &: \forall \acute{q_i}, \acute{q_j} \in \acute{S},\ (\acute{q_i}, \acute{q_j}) \in \acute{R} \Rightarrow R_\delta(\acute{q_i} \downarrow_{S_v}, \acute{q_i} \downarrow_{I_v}, \acute{q_j} \downarrow_{S_v}) \equiv T \\ \acute{L} : \acute{S} \rightarrow \mathcal{P}(E_v) \quad &: \forall \acute{q} \in \acute{S},\ \acute{L}(\acute{q}) = \{e_i \,|e_i = 1\} \\ \acute{S_0} \subseteq \acute{S} \quad &: \forall \acute{q} \in \acute{S},\ \acute{q} \downarrow_{S_v} \in S_0 \Rightarrow \acute{q} \in \acute{S_0} \end{aligned}$$

This procedure considers the externally visible inputs and outputs of the Moore machine M as additional state variables in the Kripke structure $\acute{K}$. Each state $\acute{q_i}$ in the Kripke structure $\acute{K}$ is now a unique valuation $\Im_{\acute{q_i}} : (I_v \cup S_v \cup O_v) \rightarrow \mathcal{B}$. The state space of the Kripke structure is 2^{n+m+k}. Note that each state of the original Moore machine is now modeled by many states in the Kripke structure. This is because the inputs and outputs are now considered as additional state variables of the model. $\acute{R}$ implicitly captures the non-deterministic nature of changing inputs as part of extra non-deterministic transitions between states in the Kripke structure. Given a state $\acute{q_i}$ in $\acute{K}$, $\acute{L}(\acute{q_i})$ identifies all the input and output variables that are 1 in $\acute{q_i}$. This labeling function can also be viewed as labeling each state with a set of atomic propositions about inputs and outputs that are true in that state. If this set of true atomic propositions is E_v^1, then the set of

externally visible signals that are 0 is easily obtained by $E_v^0 = E_v - E_v^1$. The set of initial states $\acute{S_0}$ is the set of states $\acute{q} \in \acute{S}$ such that $\acute{q} \downarrow_{S_v} \in S_0$.

Going from a Moore machine model to a Kripke structure enables the utilization of a variety of state-space exploration techniques for reachability and state-based analysis of circuits. Similar definitions have been used in the context of asynchronous circuits [26][114] and model checking [75][79]. The fundamental thread linking all these different modeling techniques is the classification of the domain elements into *state and non-state variables*.

2.2 State reachability analysis on circuit implementations

In this section, we show that there are properties that can be proved to hold or not hold based on different temporal structures K derived from a given circuit implementation. We then show how delays play an important role in state-space exploration and show how they can be statically modeled as part of the Kripke structure. Consider a self-timed circuit implementation as shown in Figure 2.6. It has an input A and an output B that is latched. Let the initial state of the circuit be such that $B = 0$. Let us ask the following question.

- Can the circuit ever get to a state such that $B = 1$?

We can also phrase this as a CTL property in model checking as $EF\ (B == 1)$. With the selection of the node B as the only state variable in the model, the property fails. This is because the Kripke structure extracted from the circuit does not have a state transition that goes to a state where B is a

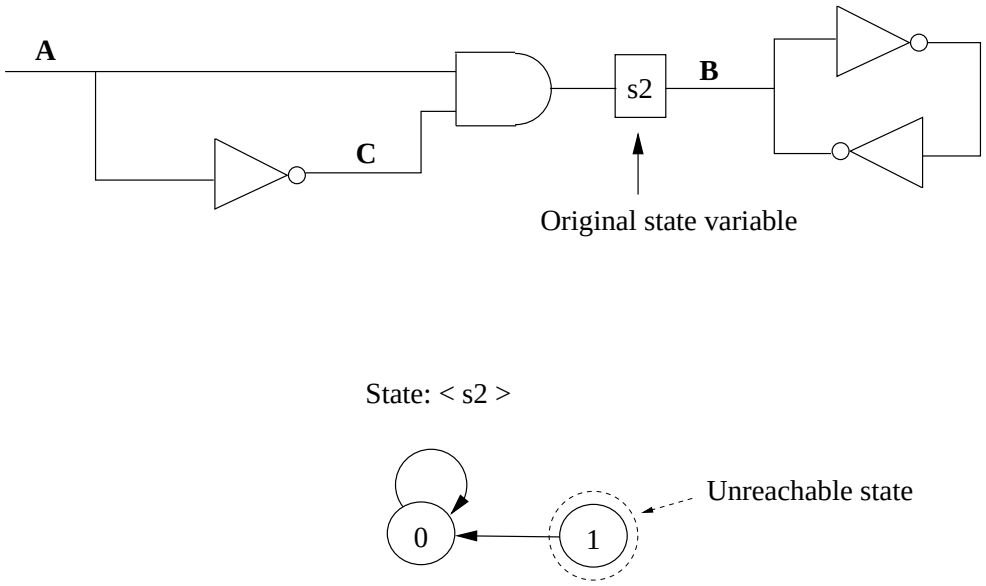


Figure 2.6: Pulse Generating Circuit

1. Now, consider the same circuit but with an additional node A being added to the state variable set as shown in Figure 2.7. Let the initial state of the circuit now be $A = 0, B = 0$. The state transition graph that is now extracted from the circuit has a total of 4 states. The property still fails to hold since the states where B is 1 is unreachable. Finally, consider the same circuit implementation but with the additional intermediate node C added to the state variable set as shown in Figure 2.8. Let the initial state of the circuit be $A = 0, B = 0, C = 1$. This time the property holds and an example of a state sequence that satisfies this property is $(001, 101, 110, 000, 001)$.

Relying solely on the logic specification for state variable identification or the classical interpretation of only feedback paths in the circuit implementation as state variables of the model can lead to incorrect model

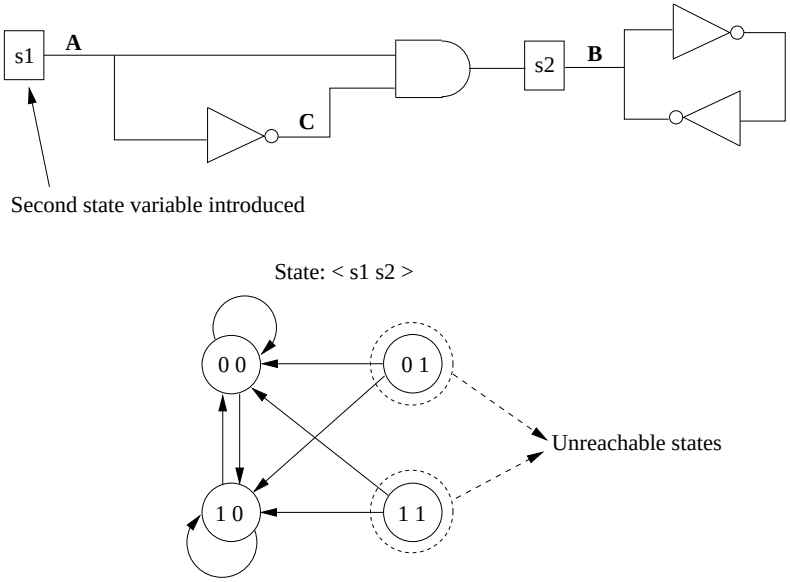


Figure 2.7: Second state variable at A introduced

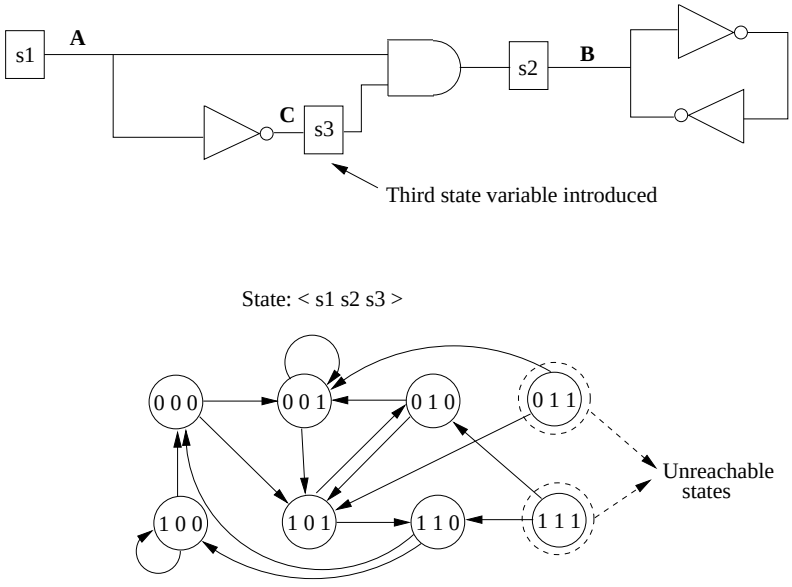


Figure 2.8: Third state variable at C introduced

generation and the impossibility of proving that certain properties hold on the implementation. This is because it may be necessary to select other additional nodes as state variables in the circuit that are not part of a feedback path or cycle but are still required to predict the correct behavior of the circuit. For proving the correctness of a broader class of implementations using state reachability techniques such as model checking or language containment, it is necessary to apriori identify the additional state variables in the implementation to complement all the specification-based state variables and the feedback state variable set.

Proposition 2.2.1. *Given a set of state variables s_{spec} modeled in a specification as an abstraction of the implementation, there exist circuit structures in the implementation such that it is necessary to augment s_{spec} with an additional set of state variables s_{imp} from the implementation to adequately model the correct behavior of the implementation and be able to prove properties on it.*

Consider a generalization of the earlier circuit structures shown in Figure 2.9. Each of the circuits has a stable output that is either 0 or 1. Such circuit structures in the implementation give rise to transient states. If the property to be proved depends on these transient states, then atleast one or more nodes in the fanin to these circuit structures must be selected as state variables. Implementation state variables s_{imp} are shown under each circuit structure in Figure 2.9. For circuit implementations ranging in the millions of transistors, trying to manually identify these state variables can be very tedious. We present a method for identifying such state variables automatically.

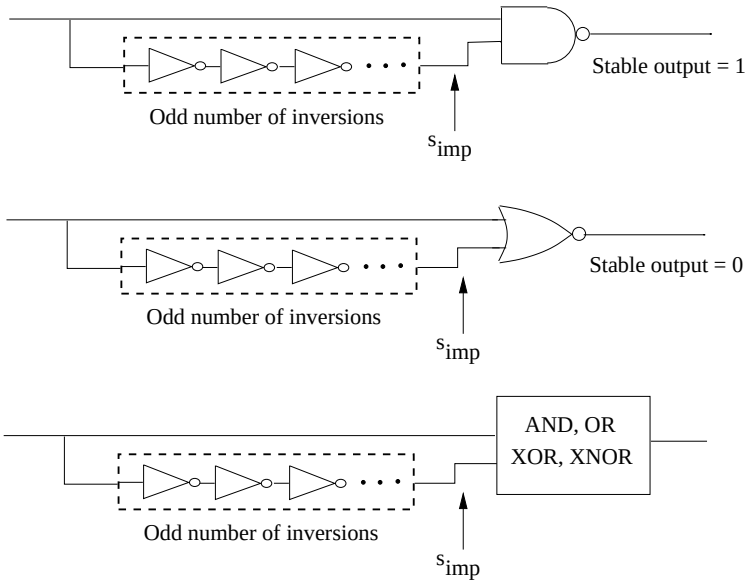


Figure 2.9: Common circuit structures for pulse generation

Definition 2.2.1. If f is a function from V to W, the image of V under f is

$$\mathrm{Img}(f, V) = \{y \,|\exists x \in V.\ y = f(x)\}$$

This can also be written as

$$\mathrm{Img}(f, V) = \{f(x) \,|x \in V\}$$

If $Z \subseteq V$, we also define

$$\mathrm{Img}(f, Z) = \{y \,|\exists x \in Z.\ y = f(x)\}$$

Definition 2.2.2. Let $f : V \to W$ and $g : Y \to Z$, with $\mathrm{Img}(f, V) \subseteq Y$, then the function composition of g with f is denoted by $g \circ f$, and is defined by

$$\forall x \in V,\ (g \circ f)(x) = g(f(x))$$

If $g(y)$ is the original function, then $g(f(x))$ is computed by the substitution of variable y in $g(y)$ by $y = f(x)$.

The next state function $\delta : S \times I \rightarrow S$ in Definition 2.1.2 is comprised of a number of state transition functions $\delta_i : S \times I \rightarrow \mathcal{B}$ for each state variable s_i in the design. Given $\delta_i : \mathcal{B}^{n+m} \rightarrow \mathcal{B}$, there always exists a decomposition δ_i into t boolean functions (t can be viewed as the number of levels of logic in a combinational cone) $[\delta_i^0, \delta_i^1, \ldots, \delta_i^{t-1}]$ and there exist values of $r_0, r_1, \ldots, r_{t-1}$ such that $r_0 = m + n$ and

$$\begin{aligned}
\delta_i^0 &: \mathcal{B}^{r_0} \rightarrow \mathcal{B}^{r_1} \\
\delta_i^1 &: \mathcal{B}^{r_1} \rightarrow \mathcal{B}^{r_2} \\
\delta_i^2 &: \mathcal{B}^{r_2} \rightarrow \mathcal{B}^{r_3} \\
&\vdots \\
\delta_i^{t-1} &: \mathcal{B}^{r_{t-1}} \rightarrow \mathcal{B}
\end{aligned}$$

and

$$\left(\delta_i^{t-1} \circ \delta_i^{t-2} \circ \ldots \delta_i^1 \circ \delta_i^0\right)(x) \equiv \delta_i(x) \text{ where } x \in S \times I$$

The number of potential internal state variables will not exceed $r_1 + r_2 + \ldots + r_{t-1}$. Since the composition operator $\circ$ is associative, the individual

δ_i^t functions can be composed in many different ways

$$(\underbrace{\delta_i^{t-1} \circ \delta_i^{t-2} \circ \ldots \delta_i^1 \circ \delta_i^0}_{\delta_i})(x)$$

$$(\underbrace{\delta_i^{t-1}}_{\delta_{s_{n+1}}} \circ \underbrace{\delta_i^{t-2} \circ \ldots \delta_i^1 \circ \delta_i^0}_{\delta_{s_{n+2}}})(x)$$

$$(\underbrace{\delta_i^{t-1} \circ \delta_i^{t-2}}_{\delta_{s_{n+1}}} \circ \underbrace{\ldots \delta_i^1 \circ \delta_i^0}_{\delta_{s_{n+2}}})(x)$$

$$\vdots$$

$$(\underbrace{\delta_i^{t-1} \circ \delta_i^{t-2} \circ \ldots \delta_i^1}_{\delta_{s_{n+1}}} \circ \underbrace{\delta_i^0}_{\delta_{s_{n+2}}})(x)$$

$$(\underbrace{\delta_i^{t-1} \circ \delta_i^{t-2}}_{\delta_{s_{n+1}}} \circ \ldots \underbrace{\delta_i^1 \circ \delta_i^0}_{\delta_{s_{n+(t/2)}}})(x)$$

$$\vdots$$

$$(\underbrace{\delta_i^{t-1}}_{\delta_{s_{n+1}}} \circ \underbrace{\delta_i^{t-2}}_{\delta_{s_{n+2}}} \circ \ldots \underbrace{\delta_i^1}_{\delta_{s_{n+(t-1)}}} \circ \underbrace{\delta_i^0}_{s_{n+t}})(x)$$

The next state function δ_i for state variable s_i can be viewed as the composition of δ_i^t functions where each δ_i^t function's range is the domain of function δ_i^{t+1}. By using function composition to compute δ_i, we are abstracting the state transitions by not considering the intermediate state variables. This is exactly why the circuits in Figures 2.6, 2.7, and 2.9 result in a boolean function that is either 0 or 1.

Proposition 2.2.2. *Every variable substitution $v = f(x)$ during the operation of function composition potentially eliminates the variable v as a state variable $s_i \in S_v$. In this context, the functional composition operator performs the function of time abstraction and therefore may also be referred to as the time abstraction operator.*

If any variable substitution $v = f(x)$ during the operation of function composition results in the original function becoming a constant 0 or a constant 1, then v must necessarily be considered a state variable s_{imp}. For example, in Figure 2.8, the function associated with node B is $\delta_{s_2} = \delta_{s_1} \wedge \delta_{s_3}$ and the function associated with node C is $\delta_{s_3} = \overline{\delta_{s_1}}$. Composing node B and node C functions, we obtain the new function

$$\begin{aligned} \delta_{s_2} \circ \delta_{s_3} &= \delta_{s_1} \wedge \delta_{s_3} \\ &= \delta_{s_1} \wedge \overline{\delta_{s_1}} \\ &= 0 \end{aligned}$$

The substitution of function δ_{s_3} for variable s_3 resulted in the function becoming 0 and therefore s_3 must be considered a state variable for model extraction. Using this algorithm, the output of the first inverter in each of the circuits in Figure 2.9 can be automatically identified as state variables. This algorithm is sufficient to identify state variables in similar self-timed structures provided a levelized functional composition technique is adopted. Levelized functional composition is where variables that are at the highest level (i.e. variables representing nodes that are farthest from the primary inputs) are first substituted before proceeding with variable substitutions at lower levels. Substitution of constants 0 or 1 during function composition is not considered a variable substitution and therefore this eliminates false identification of gate outputs as state variables whose inputs may be tied to a controlling value.

Proposition 2.2.3. *Given a logic specification and a corresponding circuit implementation for a design D, let $s_D = s_{spec} \cup s_{imp}$ be the complete set of state variables associated with D. Both the next-state function δ and*

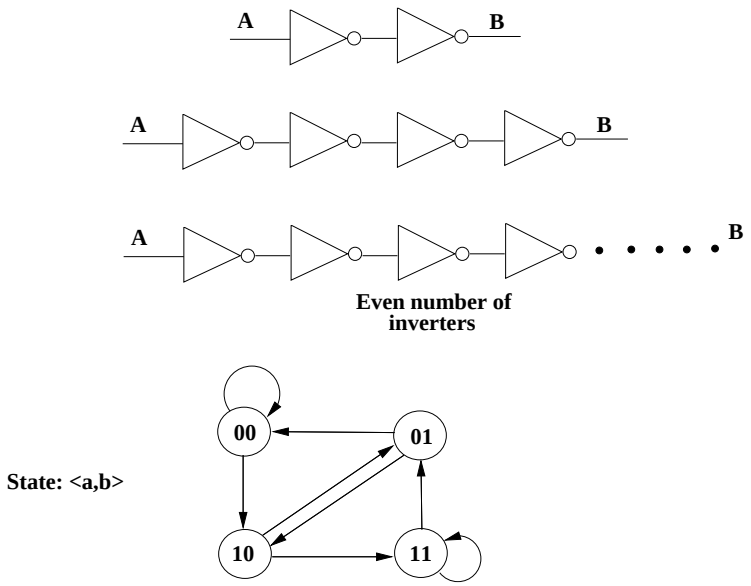


Figure 2.10: State transition model for a family of buffer circuits

the corresponding transition relation R for design D's implementation are completely determined by the state variable set s_D.

2.2.1 Time progress semantics determined by state variable selection

Consider the circuits shown in Figure 2.10. There are two state variables $G_v = \{a, b\}$ associated with the nodes A and node B for each of the circuits. The state transition graph is shown in Figure 2.10. This state transition graph adequately describes the behavior of a family of such circuits where the number of inverters is even and finite. A valid behavior of the circuit is the state sequence $\hat{u} = (00, 10, 11)$ The projected state sequence onto the nodes A and B is $\ddot{u} = \hat{u} \downarrow_{G_v} = (00, 10, 11)$ and is the same as $\hat{u}$.

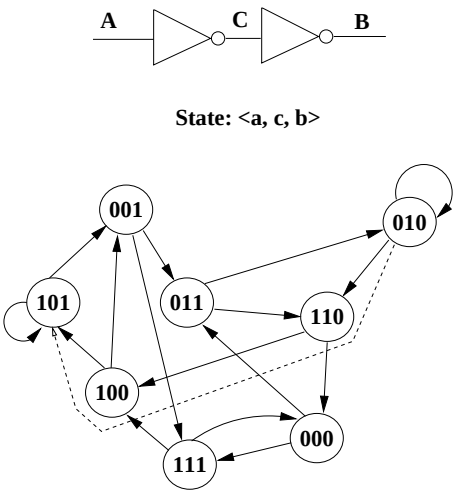


Figure 2.11: 3-state variable transition model

Now consider the top circuit in Figure 2.10 where another state variable has been identified as shown in Figure 2.11. The state variable set is now $G'_v = \{a, b, c\}$ that is associated with nodes A, B, and C respectively. The derived state transition graph is shown in Figure 2.11. Consider the state sequence $\hat{w} = (010, 110, 100, 101)$ as depicted by the dotted line. The projected state sequence with respect to $G_v = \{a, b\}$ is $\ddot{w} = \hat{w} \downarrow_{G_v} = (00, 10, 10, 11)$ and is shown next to the state transition graph in Figure 2.11. The state 10 now occurs twice in succession in $\ddot{w}$ as compared to occurring only once in $\ddot{u}$. Note that the path length from initial state 00 to the final state of 11 is one more than that of the path length in Figure 2.10.

Consider the same circuit with a fourth state variable selected in addition to the earlier three. The state variables are $G''_v = \{a, b, c, d\}$. Note

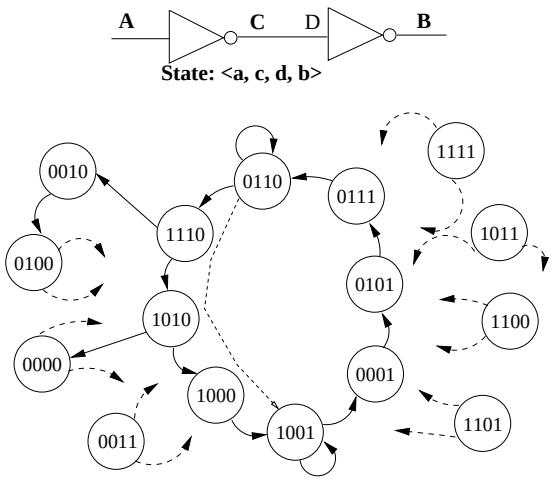


Figure 2.12: 4-state variable transition model

that c and d are two state variables for the very same node C. The reason for this selection will become clear later in this section. Consider the corresponding state transition graph with the newly selected state variable set shown in Figure 2.12. Only a subset of the state transition relation is shown. Each state has two outgoing transitions due to the non-deterministic nature of input A. Consider the state sequence $\hat{z} = (0110, 1110, 1010, 1000, 1001)$ as depicted by the dotted line in Figure 2.12. The projected state sequence is $\ddot{z} = \hat{z} \downarrow_{G_v} = (00, 10, 10, 10, 11)$. The path length from the initial state 00 to the final state 11 in $\ddot{z} = \hat{z} \downarrow_{G_v}$ is is two more than that of the path length in $\ddot{u} = \hat{u} \downarrow_{G_v}$. This state sequence is exactly identical to the input-output behavior $\ddot{z}$ we would see in an event-driven simulator if the node C were annotated with 2 units of transport delay. If we interpret each state variable as modeling a unit delay in simulation, then the path length between two states q_i and q_j

in the state transition graph gives the number of simulation time ticks or delay units it will take to go from q_i to q_j. The state variables selected completely determine the state transitions and thereby the *time progress* simulation semantics.

Let us ask the question *"Does the circuit implementation in Figure 2.12 satisfy the specification in Figure 2.10 assuming the initial states to be either* $A = 0, B = 0$ *or* $A = 1, B = 1$*?"*. To prove that the implementation satisfies the specification using scalar simulation, we would simulate a 1 at the input A in initial state 00 and then observe and hope that the output B becomes a 1. Similarly, starting in the initial state 11, we would simulate a 0 at the input A and observe and hope that the output B becomes a 0. However, it is critical that we do not sample output B too early to come to our conclusion. For example, the projected state sequence (with time instance annotations) for the circuit in Figure 2.12 is $\ddot{z} = \hat{z} \downarrow_{G_v} = (t_0 = 00, t_1 = 10, t_2 = 10, t_3 = 10, t_4 = 11, t_5 = 11, \ldots)$. If we sample the output B at any time instance $t_i < t_4$, we would conclude that the implementation fails to meet the specification. However, our conclusion would be wrong leading to a false negative. If the buffer implementation had 100 inverters in series with a different number of state variables assigned to each intermediate node, the output B must now be sampled much later in time. If the implementation in Figure 2.12 was an inverter (faulty implementation) instead of a buffer, both our simulations would give us the wrong output at B and we would conclude that the implementation violates the specification. This conclusion would be valid even if we sampled the output B at a time instance $t_i < t_4$. However, this does not mean that the output sampling time instance was correctly

chosen. There are circuit structures shown later in this dissertation where false positives can occur as a result of the wrong sampling times. Any verification or analysis algorithm must know when to sample the output to produce a valid conclusion. The problem of when to sample the outputs is completely dependent on the state variable selection and model extracted from the circuit implementation.

Any integer-valued transport delay d can be modeled *statically* with d state variables. This allows the simulation time flow mechanism using timing wheels in event-driven simulation to be captured as transition edges in the state transition graph. Therefore one method of modeling time *statically* is to introduce additional state variables in the model. However, every addition of a state variable doubles the state space of the state transition graph. Other delay models such as rise/fall or inertial delays can similarly be modeled as finite Moore machines. Different methods have been devised for modeling delays in simulation and most formal verification techniques do not model them. Here, we present a general technique for modeling transport and inertial delays such that we can move from the state-space domain to the time domain and vice-versa seamlessly.

2.2.2 Transport Delay Semantics

Any integer valued transport delay can be modeled as a finite Moore machine. Examples of a 1-unit, 2-unit, and 3-unit delay finite automaton are shown in Figure 2.13. The state space of the transport delay automaton increases exponentially with the delay d_t. This is because the automaton requires 2^{d_t} states to remember an event that happened exactly d_t time units earlier. The transport delay Moore automatons next-state function

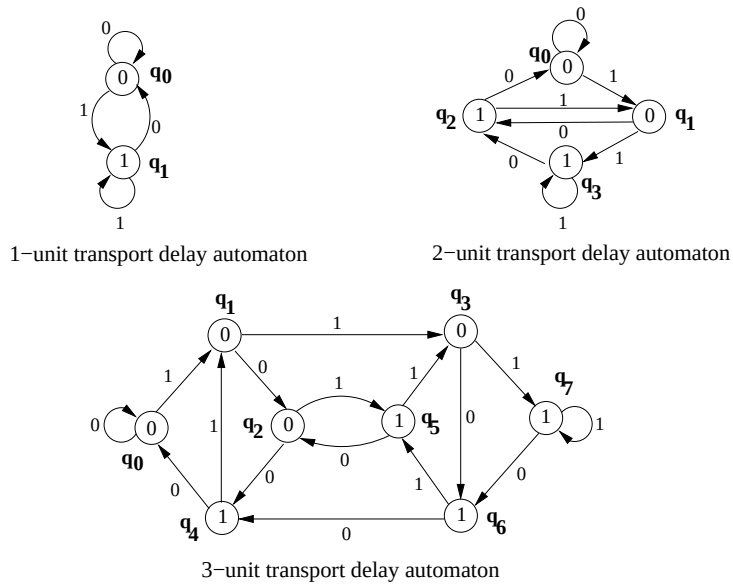


Figure 2.13: Transport Delay Moore Automatons

I\S	$\mathbf{q}_0$	$\mathbf{q}_1$	$\mathbf{q}_2$	$\mathbf{q}_3$	$\mathbf{q}_4$	$\mathbf{q}_5$	$\mathbf{q}_6$	$\mathbf{q}_7$
0	q_0	q_2	q_4	q_6	q_0	q_2	q_4	q_6
1	q_1	q_3	q_5	q_7	q_1	q_3	q_5	q_7

Table 2.1: 3-unit transport delay state table

and the output function table for a 3-unit transport delay are shown in Tables 2.1 and 2.2 respectively. Given an integer-valued transport delay d_t, the corresponding Moore machine can automatically be derived as follows.

$$
\begin{aligned}
I = &\quad \{0,1\} \qquad \text{where I is the delay's input alphabet} \\
O = &\quad \{0,1\} \qquad \text{where I is the delay's output alphabet} \\
S = &\quad \{q_0, q_1, \ldots, q_{\left(2^{d_t}-1\right)}\} \\
\delta(q_i, x) = &\quad q_{\,\left((2i+x) \bmod (2^{d_t})\right)} \qquad \text{for} \quad i = 0 \ldots \left(2^{d_t}-1\right) \\
\lambda(q_i) = &\quad \lfloor\, (i/2^{d_t-1}) \,\rfloor \qquad\quad \text{for} \quad i = 0 \ldots \left(2^{d_t}-1\right)
\end{aligned}
$$

I\S	$\mathbf{q}_0$	$\mathbf{q}_1$	$\mathbf{q}_2$	$\mathbf{q}_3$	$\mathbf{q}_4$	$\mathbf{q}_5$	$\mathbf{q}_6$	$\mathbf{q}_7$
0	0	0	0	0	1	1	1	1
1	0	0	0	0	1	1	1	1

Table 2.2: 3-unit transport delay output table

I\S	$\mathbf{q}_0$	$\mathbf{q}_1$	$\mathbf{q}_2$	$\mathbf{q}_3$	$\mathbf{q}_4$	$\mathbf{q}_5$
0	q_0	q_3	q_0	q_5	q_0	q_0
1	q_2	q_1	q_4	q_1	q_1	q_1

Table 2.3: 3-unit inertial delay state table

If the transport delay d_t is comprised of different rise d_R and fall d_F transport delays, then the delay tuple $d_t = (d_R, d_F)$ acts as an inertial delay even though d_R and d_F are transport delays. This is because there exists a possible assignment of numbers to d_R and d_F such that a pair of events at the input to the delay could cause an output transition effect of the latter event to occur earlier than the output transition effect of the first event. If $d_R \neq d_F$, then the transport delays (d_R, d_F) must be modeled as rise and fall inertial delays. Moore machine models for inertial delays are discussed in the next section.

2.2.3 Inertial Delay Semantics

Inertial delays model some physical phenomena more realistically and are used to filter out glitches or spikes during simulation. An inertial delay can be modeled by a Moore machine. Inertial delay automatons, for a d_i-unit inertial delay where $1 \leq d_i \leq 4$, are shown in Figure 2.14. The inertial delay Moore automatons next-state function and output function for a 3-unit inertial delay are shown in Tables 2.3 and 2.4 respectively.

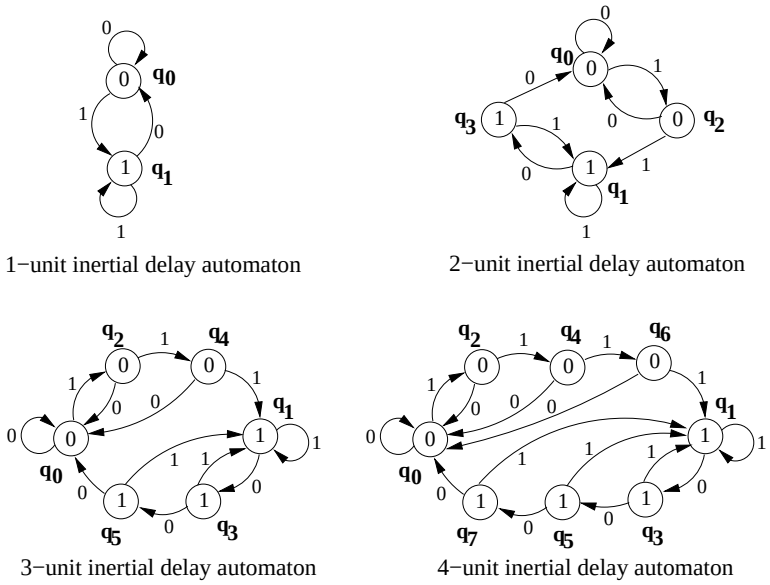


Figure 2.14: Inertial Delay Moore Automatons

$\mathbf{I \backslash S}$	$\mathbf{q}_0$	$\mathbf{q}_1$	$\mathbf{q}_2$	$\mathbf{q}_3$	$\mathbf{q}_4$	$\mathbf{q}_5$
0	0	1	0	1	0	1
1	0	1	0	1	0	1

Table 2.4: 3-unit inertial delay output table

The state space of the inertial delay automaton increases linearly with the inertial delay d_i as compared to the exponential relationship for transport delays. Each d_i-unit inertial delay automaton has exactly two more states than the previous $(d_i - 1)$-unit delay automaton. This drastic reduction in the state space from an exponential state space in the case of transport delays to a linear state space in the case of inertial delays is because the automaton does not have to remember every event in the past. It only has to remember past events, separated in time, that are longer than d_i time units. If the time elapse between a pair of consecutive events is less than d_i time units, the automaton can forget the first event component of the event pair as soon as the second component of the event pair occurs. If the rise and fall delays are different, then $d_i = (d_R, d_F)$ where d_R is the delay for the rising transition and d_F is the delay for the falling transition.

Consider the delay automatons for a $(d_R = 2, d_F = 3)$-units and $(d_R = 2, d_F = 3)$-units inertial delays as shown in the top half of Figure 2.15. In both the automatons, the state sequence corresponding to a rising edge from 0 to 1 proceeds along the top half of the automaton, while the state sequence corresponding to a falling edge from 1 to 0 proceeds along the bottom half of the automaton. The general inertial delay Moore machine for an arbitrary rising and falling inertial delay is shown in the bottom half of Figure 2.15. The cardinality of the total state space of an inertial delay Moore machine is $|S| = d_R + d_F$. Given a specific rise/fall inertial delay $d_i = (d_R, d_F)$, the corresponding Moore automaton can be

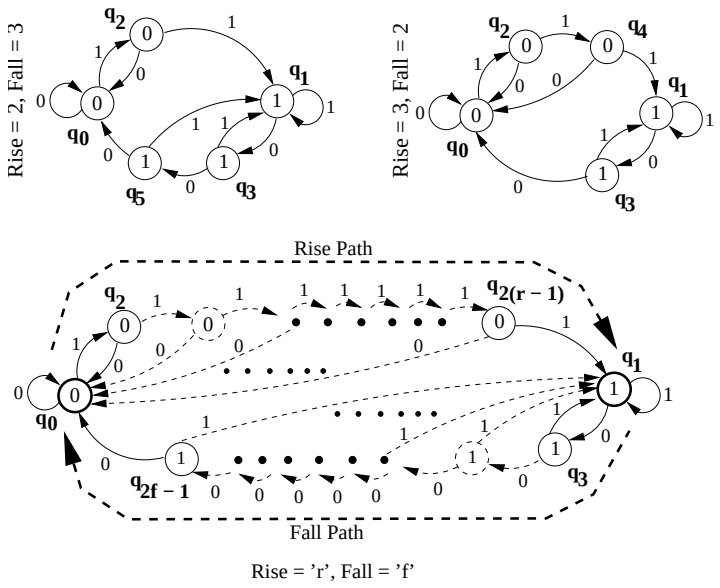


Figure 2.15: Rise and Fall Inertial Delay Moore Automatons

derived automatically as follows

$$
\begin{aligned}
I = &\quad \{0,1\} \qquad \text{where I is the delay's input alphabet} \\
O = &\quad \{0,1\} \qquad \text{where I is the delay's output alphabet} \\
S = &\quad \{q_0, q_2, q_4, \ldots, q_{(2\times(d_R-1))}\} \cup \{q_1, q_3, q_5, \ldots, q_{(2\times d_F)-1}\}
\end{aligned}
$$

$$
\delta(q_i, x) = \begin{cases}
q_0 & , \text{ if } i \text{ is even and } x = 0 \\
q_1 & , \text{ if } i \text{ is odd and } x = 1 \\
q_{\,((i+2) \bmod ((2\times d_F)+1))} & , \text{ if } i \text{ is odd and } x = 0 \\
q_{\,((i+2) \bmod ((2\times d_R)-1))} & , \text{ if } i \text{ is even and } x = 1
\end{cases}
$$

$$
\lambda(q_i) = \begin{cases}
0 & , \text{ if } i \text{ is even} \\
1 & , \text{ if } i \text{ is odd}
\end{cases}
$$

If n is the number of state variables, m is the number of inputs, k is the number of outputs, p is the number of combinational nodes, and d_t^a is the average transport delay per combinational node, then the total state

space of the circuit is $2^{n+m+k+\,(p\times d_t^a)}$. If inertial delays are considered and d_i^a is the average inertial delay per node, then the total state space of the circuit is $2^{(n+m+k+\,(p\times(\lceil \log_2(2\times d_i^a)\rceil)))}$

Delay attribution on nodes in a circuit for simulating time progress in event-driven simulation is equivalent to selecting the nodes as state variables in state reachability analysis. Some design methodologies model hardware at the behavioral level with behavioral delays to get a more accurate view of the events in their simulation [42]. The IEEE Verilog 1364-2001 standard has incorporated a number of new constructs to enable accurate modeling of timing for very deep sub-micron designs. With timing being modeled at the behavioral level and/or the requirement for doing race or hazard analysis earlier in the design cycle, any verification or analysis technique based on state-space exploration, will require the modeling of delays as additional state variables for correct state transition model extraction. This requirement clearly exacerbates the state explosion problem. This is how symbolic simulation is able to use the simulation time flow mechanism to avoid modeling delays as explicit state spaces. In the next section, we will present techniques that allow us to model delays without explicitly modeling them as part of the state transition graph.

2.2.4 Delay synthesis

Consider the buffer example discussed earlier and shown again in Figure 2.16. Assume that we would like to simulate the buffer with an inertial delay of $(d_R = 2, d_F = 1)$ units on node C. The inertial delay automaton for a $(d_R = 2, d_F = 1)$ delay is shown below the buffer. Let us now synthesize this inertial delay Moore machine into a combinational and

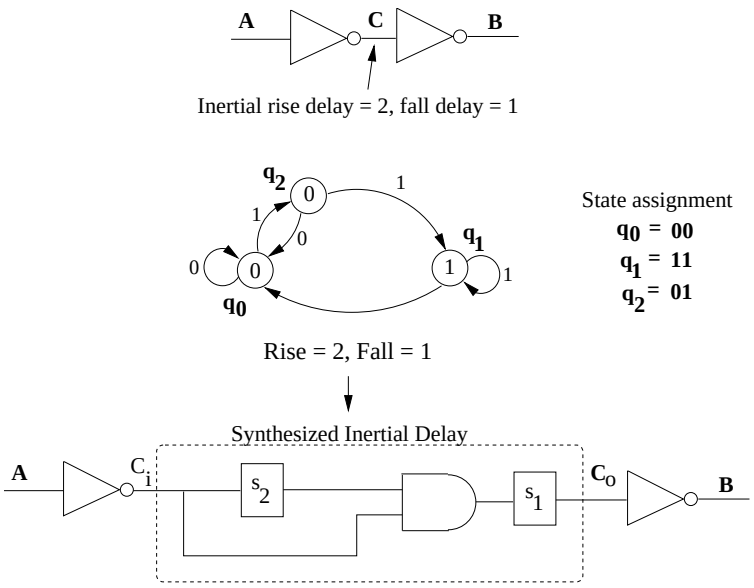


Figure 2.16: Synthesized logic for an inertial delay

i\ s_1s_2	**00**	**01**	**11**	**10**
0	0	0	0	0
1	0	1	1	0

Table 2.5: s_1's next state table

latch-based circuit using standard synthesis algorithms. Its corresponding synthesized logic is shown below the automaton and is inserted into the original buffer at node C. The output of the first inverter feeds into the delay logic input C_i and the output C_o of the delay logic feeds into the input of the second inverter. The state assignment used for synthesis is shown on the right hand side of Figure 2.16. The synthesized delay logic has exactly one input and one output. The logic was synthesized using the state assignment in Figure 2.16 and state transition functions in Tables 2.5, 2.6 and 2.7. All the entries for state **10** in the tables are dont-cares and

$\mathbf{i \backslash\ s_1 s_2}$	**00**	**01**	**11**	**10**
0	0	0	0	0
1	1	1	1	1

Table 2.6: s_2's next state table

$\mathbf{i \backslash\ s_1 s_2}$	**00**	**01**	**11**	**10**
0	0	0	1	1
1	0	0	1	1

Table 2.7: Output C_o's function table

so they have been assigned values to simplify the synthesized logic. The behavior of the *delay-latches* s_1 and s_2 is such that whenever there is a change in the input to a *delay-latch*, the latch gets updated in the next simulation time tick. Assuming the value domain of all the signals to be $\mathcal{B} = \{0, 1\}$, four different simulations of the delay synthesized buffer are depicted in Figure 2.17. The top two simulation waveforms depict the rising edge of C_i being delayed by 2 delay units and the falling edge being delayed by 1 delay unit. The bottom left simulation trace shows a pulse of 1-unit width applied to C_i. The pulse gets suppressed as it passes through the delay-synthesized logic.

Consider a ternary-valued simulation of the delay synthesized buffer as shown in Figure 2.18. Transitions $X \rightarrow 1$ and $0 \rightarrow X$ are considered rising transitions whereas transitions $X \rightarrow 0$ and $1 \rightarrow X$ are falling transitions. Here again the bottom left waveform shows the inertial suppression (due to the rising transition delay) of an input pulse that goes to X. With the value domain as $\mathcal{T} = \{0, 1, X\}$ and the interpretation of the ternary value X as either 0 or 1, the ternary simulations, in terms of the delay se-

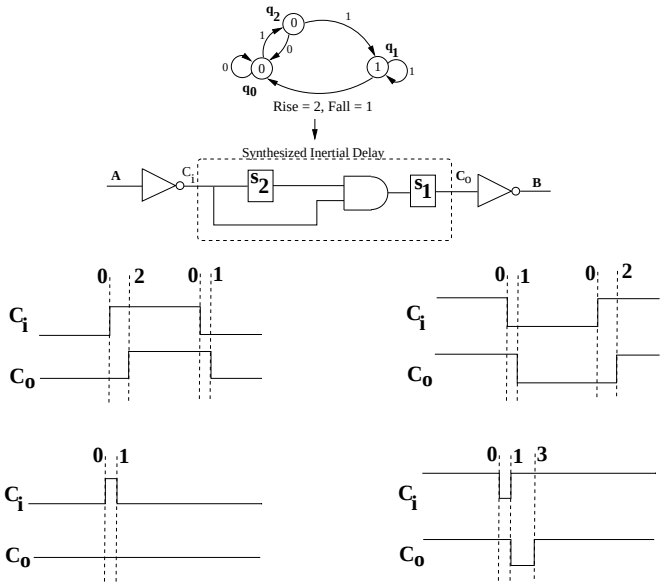


Figure 2.17: Binary-valued simulation

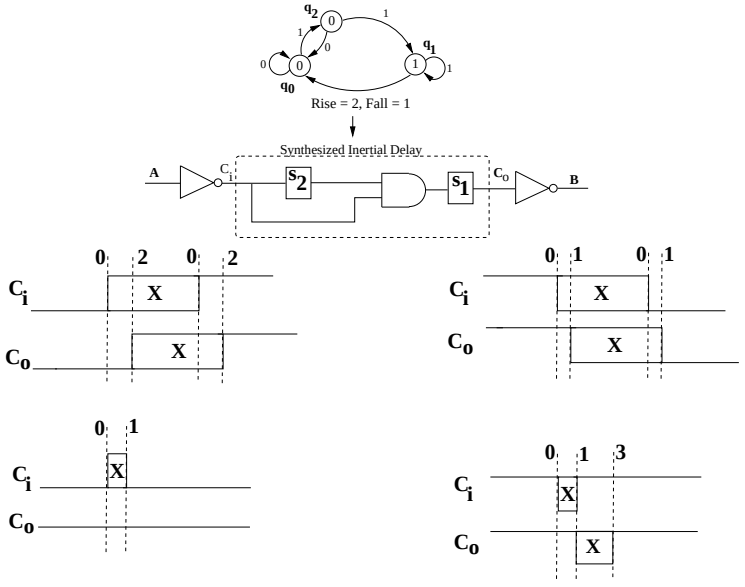


Figure 2.18: Ternary-valued simulation

mantics, turn out to be exactly the same as the binary-valued simulations.

The delay-latches implementing the synthesized delay logic do not have any clocks associated with them. We can view a delay-latch element as having an implicit clock that enables the delay-latch as soon as its input changes. The state of the delay-latch is always updated at simulation time tick t_{i+1} whenever there is an event on its input at time t_i. By modeling both transport delays and inertial delays as Moore machines, we can now synthesize the delays into logic using boolean gates and delay-latches. Seger et al. [102] presented an inertial delay circuit model construction for ternary symbolic race analysis in terms of the dual-rail encoding. The number of unit-delay elements used in the delay circuit was linear with respect to the size of the inertial delay. This construction is sub-optimal with respect to the number of delay-latches and we can do much better. The inertial-delay synthesis algorithm presented here constructs a delay circuit that is logarithmic with respect to the size of the inertial delay. Simulation results using the new construction are shown later in this dissertation.

By embedding the delay logic into the circuit implementation, we can use existing logic simulation algorithms to perform a delay-based simulation. Depending on the logic simulation algorithm, the synthesized logic could be built into the extracted model itself thereby not having to modify the circuit implementation. This also enables the model generation process to directly leverage advancements and research in the logic synthesis field for modeling delays in simulation. In the next section, we will show how the delay-synthesized logic and its inclusion as part of the circuit model enables efficient data-independent event-driven symbolic simulators to be derived.

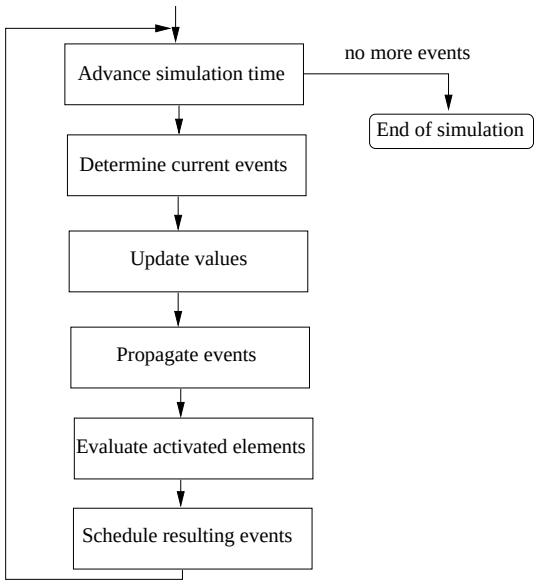


Figure 2.19: Event-driven simulation flow

2.3 Event-driven simulation

The foremost accuracy requirements for simulation are adequate structural detail, a sufficient repertoire of state variables, and relative timing accuracy. Event-driven simulators use an event-evaluation algorithm and a simulation time-flow mechanism to compute and schedule events such that they occur in the correct temporal order. The input stimuli and output responses are represented as sequences of events. Events scheduled to occur in the future relative to the current simulated time are stored in an event queue. A standard event-driven simulation flow chart [4] is shown in Figure 2.19. The event queue could be implemented as an array of event lists such that

- Each entry in the array is indexed by the simulation time t. Each slot in the array is a *simulation tick*. This array is also known as the

timing wheel.

- Each event list is a list of events of the type (i, v_i^f).

An entry (i, v_i^f) in an event list indexed by t_j indicates that at time t_j, the value of *element* i is scheduled to be set to v_i^f. v_i^p denotes the previously scheduled value for element i and t_i^p denotes the time that v_i^p was scheduled.

An event-driven simulation algorithm relies on the structure of the circuit for simulation. The structure is completely determined by the objects in the circuit denoted as elements and how they are connected. An element i is the fundamental object in the circuit model and typically has a next-state function associated with it. What constitutes an element is a fundamental tradeoff resulting in different extracted models from the circuit implementation. The issue of what constitutes sufficient inter-connective detail or structure has been the subject to debate among researchers and simulation experts. There is agreement, however, that structural detail is important and necessary [2][113]. The general event-driven simulation algorithm is shown below.

Algorithm1: Event driven Inertial Delay Simulation Algorithm

while (timing wheel is not empty)

1. begin
2. t_c = next simulation time in the timing wheel
3. foreach event (i, v_i^f) pending at the current time t_c
4. begin
5. $v_i^c = v_i^f$
6. foreach element j activated by the event v_i^c

7. begin
8. v_j^f = evaluate element j using v_i^c `/* Compute element's output */`
9. while ($v_j^f \neq v_j^p$) then `/* new value is not equal to previous value */`
10. begin
11. $d = (v_j^f == 1)\ ?\ d_R\ :\ d_F$ `/* Rise or Fall delay? */`
12. $t_n = t_c + d$
13. if ($t_n - t_j^p > d$) then `/* Check for inertial delay criteria */`
14. begin
15. schedule (j, v_j^f) for simulation time t_n
16. $t_j^p = t_n$
17. $v_j^p = v_j^f$
18. end
19. else `/* Pulse is suppressed */`
20. begin
21. remove event v_j^p from timing wheel at time t_j^p
22. if (additional pending events for element j) then
23. begin
24. v_j^p = value of last pending event for j
25. t_j^p = time of last pending event for j
26. end
27. else
28. begin
29. $v_j^p = v_j^c$
30. $t_j^p = t_c$
31. end
32. end
33. end

34. end

35. end

36. end

The event-driven algorithm presented here is a one-pass strategy [112]. There are two-pass strategy algorithms where the events are first retrieved in the first pass and the element evaluations are then performed in the second pass. This is done to avoid the repeated evaluations of elements that have multiple input events. There are quite a few variations and optimizations on the algorithms and data structures [4][115]. For the purposes of this dissertation, the above algorithm is sufficient for elucidation of the key principles.

The element values (v_i^c, v_i^f, v_i^p, ...) could range over the boolean domain $\mathcal{B} = \{0, 1\}$, or the ternary logic domain $\mathcal{T} = \{0, 1, X\}$ or any other multi-valued algebraic domain [27][47][53]. Since each signal in the circuit can now take on multiple logic values, this requires modification of the simulation algorithm. To incorporate a multi-valued logic simulation algorithm into the above simulation procedure, the following steps need to be taken.

Step 1: Line 8 computes the output of an element by calling an element evaluation procedure. A multi-valued algebra must be defined and the element evaluation procedure modified to compute a multi-valued function. It is important that the domain of the multi-valued algebra satisfy closure properties with respect to its binary and unary operators.

Step 2: Line 9 incorporates the test that compares the current value of an element's output with the future value to be scheduled. The *equality* operator must be defined as a boolean predicate for the multi-valued logic system so that comparisons between values in the domain can be made. The equality test must be efficient as this is the basis for determining and processing events.

Step 3: Line 11 incorporates the tests for selection of the appropriate rise or fall delay values for scheduling. The rising and falling transitions must be defined with respect to the underlying set of logic values. Then line 11 in the algorithm must be modified to include the appropriate tests.

Consider a binary-valued simulator that simulates values over the domain $\mathcal{B}$. Now consider the domain of element values to be the set $\mathcal{B}^f = \{f : \mathcal{B}^n \rightarrow \mathcal{B}\}$, where $\mathcal{B}^f$ is the set of all functions mapping a set of n boolean variables to the set $\mathcal{B}$. This value domain is now a set of boolean functions. By following the recipe described in Steps 1 to 3 for this new domain, we can implement a simulator over the abstracted domain of boolean functions. The scalar event-driven binary-valued simulation algorithm is now a boolean function event-driven simulator.

What does it mean for signal values in the circuit to range over this domain? How do we modify the simulation algorithms to operate over these new function domains and still be able to model delays correctly? We will address these questions in the next section.

2.3.1 Data-Independent Event-Driven Simulation

Given a multi-valued logic domain D and a set of variables V_D of cardinality n that range over the domain D, we can generate a function domain consisting of all functions mapping the the set of V_D variables to the set D, i.e.

$$\{f : D^n \rightarrow D\}$$

This process of abstracting from a multi-valued logic domain to one of functions over these values forms the basis of multi-valued symbolic simulation. All the algebraic properties that hold in the original domain also hold on the abstract one.

In the earlier event-scheduling algorithm described in Algorithm 1, Line 11 incorporated the checks for rising and falling transitions of events. To modify Algorithm 1 for multi-valued logic simulation using inertial delays, the rising and falling transitions must be well defined for the domain. This is possible only if the multi-valued logic domain is given an interpretation in terms of the binary domain $\mathcal{B} = \{0, 1\}$. For example, in the ternary domain $\{0, 1, X\}$, X transitioning to a 1 is considered a rising transition and X transitioning to a 0 is considered a falling transition. This is because of the interpretation of X as a 0 or a 1. For X transitioning to a 1, if X is interpreted as a 1, there is no rising transition, but when X is interpreted as a 0, a rising transition occurs. For the symbolic domain, the tests for rising and falling transitions are even more complex since the transitions are now dependent on valuations of the multi-valued logic variables. This implies that the event simulation algorithm is now data-dependent and scheduling of events is now a non-trivial task.

To eliminate this data-dependency test from the event scheduling algorithm, we modify the circuit model to now include the synthesized delay logic as part of the circuit model. Consider the boolean symbolic simulation of the buffer in Figure 2.20 . Three different symbolic simula-

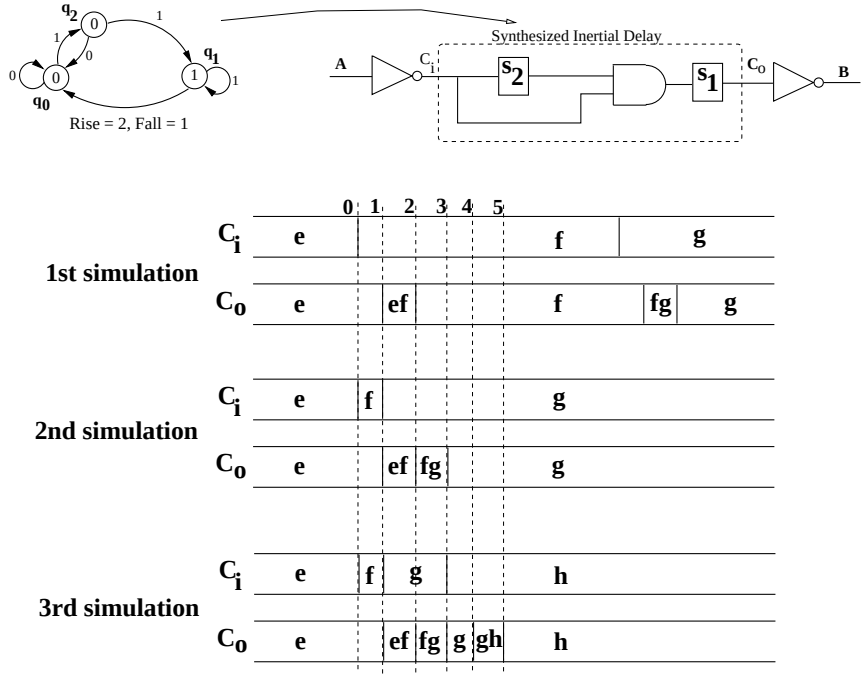


Figure 2.20: Boolean Symbolic Simulation

tions of the synthesized delay logic with boolean variables e, f, g, h have been depicted. In the first simulation, there are two symbolic transitions $e \rightarrow f$ and $f \rightarrow g$. Consider one valuation of the boolean variables where $e = 0, f = 1$ and another valuation where $e = 1, f = 0$. The simulated waveforms at the output C_o is exactly as what was observed in the top left and top right of the binary valued simulations in Figure 2.17. In the third simulation, there are three symbolic transitions $e \rightarrow f$, $f \rightarrow g$, and $g \rightarrow h$. Consider the valuation $e = 0, f = 1, g = 0, h = 1$. By substituting these

valuations into the boolean expressions shown for C_o, it is observed that the only transition that occurs is at time t_5 which is a rising transition. This is because this specific valuation causes a pulse to occur between t_0 and t_1 at C_i which gets suppressed by the synthesized delay logic. Now consider a second valuation $e = 0, f = 0, g = 1, h = 0$, which generates an input waveform at C_i such that a rising transition occurs at t_1 and a falling transition occurs at t_3. This input waveform now causes a rising transition on C_o at t_3 and a falling transition at t_4 which is consistent with the inertial delay of $(t_R = 2, t_F = 1)$.

A key property of the modified delay-synthesized circuit is that the same delay logic has preserved its semantics across the binary-valued logic domain, the ternary-valued logic domain, and the symbolic boolean function domain. Assuming a modified circuit model with delay logic instantiated wherever delays are required, we can now modify Algorithm 1 to be a multi-domain event-driven symbolic simulation algorithm This will allow simulating both the multi-valued and symbolic domains while preserving the delay semantics. The modified event-driven symbolic simulation algorithm is shown below.

Algorithm2: Modified Event driven Inertial Delay Simulation Algorithm

while (timing wheel is not empty)

1. begin
2. t_c = next simulation time in the timing wheel
3. for every event (i, v_i^f) pending at the current time t_c
4. begin
5. $v_i^c = v_i^f$
6. for every element j activated by the event v_i^c

7. begin
8. v_j^f = evaluate element j using v_i^c `/* Compute element's output */`
9. while ($v_j^f \neq v_j^p$) then
10. begin
12. $t_n = t_c$+(Is j a delay-latch) ? 1 : 0 `/* State or Comb. element? */`
15. schedule (j, v_j^f) for simulation time t_n
17. $v_j^p = v_j^f$
33. end
34. end
35. end
36. end

The delay-latches in the delay synthesized logic update to a new value whenever there is an event on their inputs. We can also view a delay-latch as having a clock that enables it precisely at the time when its input changes. An event at the input to a delay-latch at simulation time t_i will cause an update of its state at time t_{i+1}. That is the reason why **Line 12** in the above algorithm advances the simulation time tick by one for the delay-latches. Since the circuit structure is now composed of only combinational nodes and delay-latch elements (assuming that regular state holding elements have their feedback broken and a single delay-latch is introduced in the feedback path), the time increment of 0 in **Line 12** enables the algorithm to evaluate combinational logic in 0-delay time. Using this algorithm, combinational logic between delay-latches is evaluated in 0 time. When all the events have been processed and removed at the current time t_c, the delay-latches are ready to be updated with new values in the next simulation time tick $t_c + 1$.

The modified event-driven simulation algorithm is now independent of the underlying multi-valued logic symbolic domain and is data independent. Moreover, it removes the problem of event cancellation (Line 21 in Algorithm 1) required to implement inertial delays and eliminates the overhead of event list searching to determine the last pending event. However, this algorithm is possible only at the expense of modifying the circuit model (extracted from the circuit implementation) with the delay-synthesized logic. Note that the modification is not to the actual circuit implementation but to the extracted model and therefore can be implemented directly in software.

2.4 Validation using Symbolic Trajectory Evaluation

A Symbolic Trajectory Evaluator (STE) [103][104] is a modified form of symbolic simulation that operates over the quaternary logic domain **0**, **1**, **X** and $\top$. STE does data independent simulations and is fundamentally based on Algorithm 2 described in Section 2.3.1. It differs from symbolic simulation in that assertions of the form $Antecedent(A) \Rightarrow Consequent(C)$ can be expressed and proved to hold for a given simulation model of a circuit. The stimulus to the circuit, consisting of boolean variables driving the inputs and the state-holding nodes, is specified by the antecedent. The expected values on the nodes are specified by the consequent. The model is then simulated, typically for one or two clock cycles, while driving the inputs with symbolic values during the course of the simulation. Then the resulting values that appear on selected internal nodes and primary outputs are compared with the expected values expressed in the consequent. In the more general case, the values could be

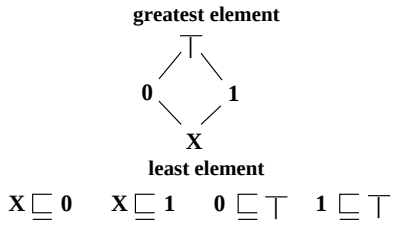


Figure 2.21: Partially ordered state space for a single node

functions over a finite set of variables. The symbolic trajectory evaluator can simulate both RTL and switch-level models [16][18][19].

A symbolic trajectory evaluator can also be thought of as a model-checker where the state-space is partially ordered [54]. Partial orders are effective representations of state spaces of systems. In this technique, the partial order represents the relative information content of states. $r \sqsubseteq q$ implies that r has less information than q. If $r \sqsubseteq q$ and $r \sqsubseteq t$, we can think of r as representing both q and t. By proving a property on a state s lower in the lattice (i.e. closer to the bottom element), the same property automatically holds for all states above s in the lattice. It requires a system expressed as a model of the form $\langle S, \preceq, y \rangle$ where S is a set of states, $\preceq$ is a partial order on the states ($\preceq \subseteq S \times S$) and $y : S \rightarrow S$ is a state transition function. S must form a complete lattice and y must be monotonic under $\preceq$, i.e., if $s \preceq t$ then $y(s) \preceq y(t)$.

Traditionally, for transistor level verification, STE operates over the ternary logic domain $\mathcal{T} = \{0, 1, X\}$, where X denotes an "unknown" value. In order to formalize the concept of an "unknown" value, define a partial order $\preceq$ on $\mathcal{T}$ as illustrated in Figure 2.21, so that $X \preceq 0$, and $X \preceq 1$. States of an STE model are vectors of elements taken from $\mathcal{T}$. The partial order over $\mathcal{T}$ is extended pointwise to yield a partial order on the space

$\mathfrak{T}^n$. Unfortunately, $\langle \mathfrak{T}^n, \preceq \rangle$ is not a complete lattice, since the least upper bound does not exist for every pair of vectors in $\mathfrak{T}^n$. Introduction of a new *top* element, $\top$, solves this problem. Intuitively, $\top$ can be viewed as an "over-constrained" state, in which some node is both 0 and 1 at the same time. This makes the state space to be $S = \mathfrak{T}^n \cup \{\top\}$. $X \sqsubseteq 0$ indicates X has less information than 0 or X is weaker than 0. 0 is neither weaker nor stronger than 1 since their information contents are incomparable. For a complete treatment of partial order and lattice theory, the reader is referred to [10].

In STE, the state of a circuit model includes ternary values of the input, internal and output nodes of the circuit. This is similar to the earlier definition of state in the Kripke structure model except that the state values are ternary and not binary. So, typically a state vector from $\mathfrak{T}^n$ represents a string of n ternary values of circuit nodes, where n is the collective number of all circuit nodes. STE provides a mathematically rigorous method for establishing that temporal logic properties of the form $Antecedent(A) \Rightarrow Consequent(C)$ hold for a given simulation model of a circuit. The Antecedent and Consequent are trajectory formulas [104]. A trajectory formula is a temporal logic formula that combines Boolean expressions and the temporal logic next-time operator. In general, trajectory formulas are defined recursively as follows:

1. **Simple predicate:** Any simple predicate of the form n_i *is value* v is a trajectory formula.

2. **Conjunction:** $(F1 \wedge F2)$ is a trajectory formula if F1 and F2 are trajectory formulas.

3. **Domain restriction:** $(g \rightarrow F)$ is a trajectory formula if F is a trajectory formula and g is a boolean function

4. **Next time:** (**N**F) is a trajectory formula if F is a trajectory formula.

For example, *Node* **n** *has the value* **v** *from time* **t1** *to* **t2** *under the condition that* **g** *is true* is a trajectory formula. If **g** is false and the formula is part of the antecedent, then the node **n** takes on the value of its next-state function. If **g** is false and the formula is part of the consequent, then no check is performed on the node.

A trajectory is a sequence of states such that each state has at least as much information as the next-state function applied to the previous state. Let Y be the next-state function and $q_0, q_1, q_2, \ldots$ be an arbitrary infinite state sequence. The infinite state sequence $q_0, q_1, q_2, \ldots$ is a trajectory if and only if $\forall i \geq 0, Y(q_i) \sqsubseteq q_{i+1}$. Intuitively, a trajectory is a state sequence constrained by the system's next-state function. Consider an antecedent for an inverter stated as *Node A is* 0 *from 0 to 1 and is* 1 *from 1 to 2 and is* 0 *from 3 to 4*. Valid trajectories for the antecedent are the first and second state sequences shown in Figure 2.22. The third state sequence is an invalid trajectory since $< X, 0 > \not\sqsubseteq < X, 1 >$. The value on node B is 1 instead of the correct value 0.

A successful simulation of $A \Rightarrow C$ establishes that any sequence of assignments of values to circuit nodes that is both consistent with the circuit behavior and consistent with the antecedent A is also consistent with the consequent C. For example, consider a specification for an inverter. An assertion that could be checked on an implementation for an inverter

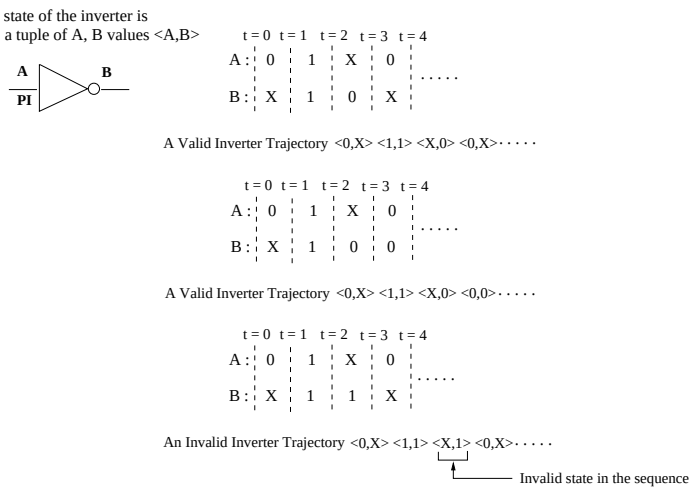


Figure 2.22: Valid and Invalid Inverter Trajectories

is *"If node A is* 0 *from 0 to 1 then B is* 1 *from 1 to 2"*. If the implementation is correct, then the weakest antecedent trajectory that the circuit goes through is shown in Figure 2.23. The trajectory that the circuit goes through is at least as strong as the weakest sequence satisfying the consequent and the assertion holds. If the implementation is incorrect and a buffer is implemented instead of an inverter, the weakest antecedent trajectory that the circuit goes through is shown in Figure 2.24. The trajectory that the circuit goes through is not at least as strong as the consequent's weakest state sequence and the assertion fails. For more details on the structure of these assertions and the derivation of trajectories, the reader is referred to [104].

The three main advantages of symbolic trajectory evaluation are that the underlying state space is partially-ordered, no monolithic transition relation of the design is built, and the symbolic simulation engine is capable of carrying out a data-independent simulation of a switch-level

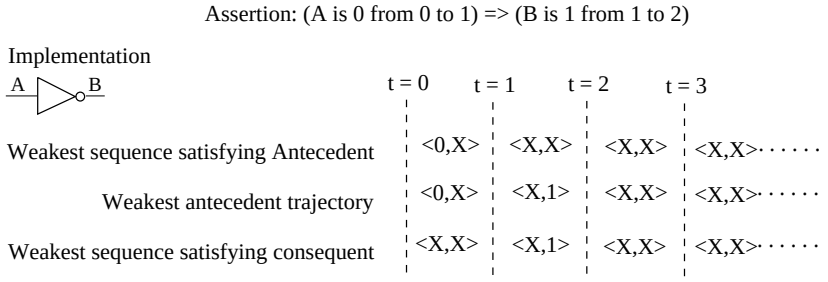


Figure 2.23: Trajectory for Correct Implementation

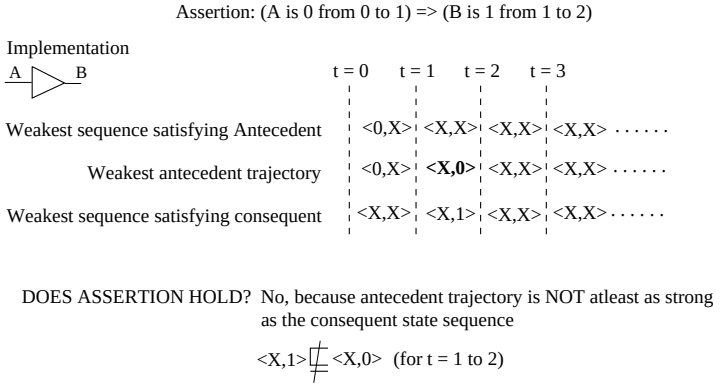


Figure 2.24: Trajectory for Incorrect Implementation

model of the circuit. In most realistic designs, the transition relations required for other formal verification techniques such as model checking or language containment cannot be built. This is due to the large number of state-holding elements. By keeping time explicit and the temporal logic simple, the logic is less expressive but is more intuitive to phrase properties to be checked and debugged on the implementation. At any point during the symbolic simulation, only the present-state and the previous state are kept in memory and the decision procedure for proving the property $Antecedent \Rightarrow Consequent$ is invoked every simulation tick.

2.5 RTL Specifications and Self-Timed Implementations

Our validation involves two different representations, the specification and the implementation, of the circuit. The specification is at the register-transfer level (RTL) and is a description of the required behavior of the circuit implementation. These descriptions are written by logic designers to describe the functionality of their design blocks. The RTL specification is a finite state cycle-accurate description of the required behavior of the circuit. Billions of cycles are simulated on the RTL to ensure that the RTL design functions as intended. In a synchronous design methodology, the model is evaluated either once or twice every clock cycle. Typically, the RTL is written in a hardware description language such as Verilog.

Circuit implementations can be broadly categorized as either synthesized or custom. The key distinction between the two is that synthesized implementations are automatically generated by software programs whereas custom implementations are created manually. The reason custom

implementations are warranted in designs is because of the aggressive performance target metrics of multi-gigahertz, low-power, and minimum area designs. Current synthesis algorithms are capable of dealing with one or at most two of these metrics but are unable to generate circuit structures that satisfy all three criteria. Custom and semi-custom (a mix of synthesized and custom circuit structures) implementations are relying more and more on pulsed clocks, self-timed circuits, and are incorporating asynchronous circuit structures in their custom designs. To achieve all of the above performance metrics, designers resort to creative and non-standard transistor circuit implementation structures.

Consider an informal behavior specification that reads something like "*A tag memory is required that can store 2 tags where each tag is 2 bits wide. The reads and writes do not occur in the same phase of the clock cycle. When the write enable is asserted, the incoming tag must be written into the addressed location and when the read enable is asserted, the tag is read from the address-indexed memory*". A corresponding verilog RTL description of the requirements specification is shown below.

RTL specification for a 4-bit Memory

```
reg [0:1] memory [0:1]
always @(C2 or Write_enable or din or address)
begin
   if (C2 & Write_enable)
      memory[address] <= din
end
assign dout = (C1 & Read_enable) ? memory[address] : 2'bx
```

In the above specification, C1 and C2 depict the two phases of a clock cycle. Since the writes and reads are required by the specification to be in opposite phases, we use $C2$ for the write and $C1$ for the read operation. This RTL description is then simulated and the read and write operations are verified to work as intended.

Consider a custom circuit implementation shown in Figure 2.25 that is intended to satisfy the RTL specification. Only one bitcell, of the 4-bit

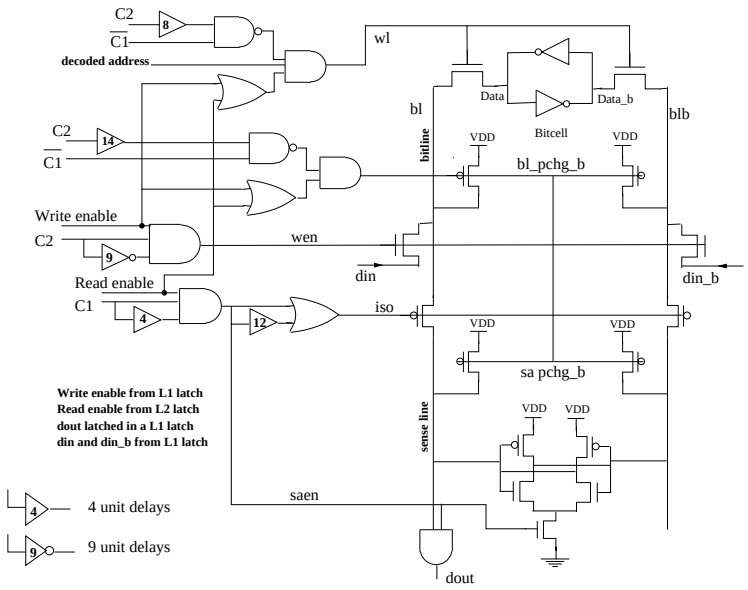


Figure 2.25: Read/Write control logic for a bitcell

custom memory, and its associated read and write control logic is depicted. The control circuitry is identical for all the other bits and the only differences are in the connections of the *din* and decoded *address* bits. The memory has a number of carefully designed timing chains that ensure the correct temporal relationships between precharge, isolate, sense-amp enable, word-line assertion and write enable operations. Each gate in the

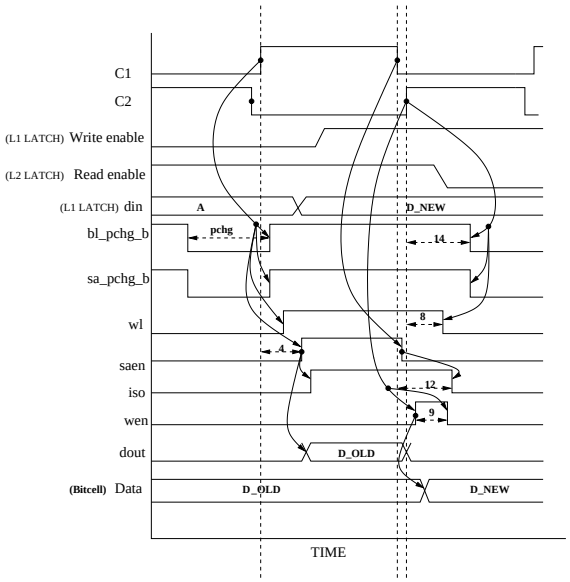


Figure 2.26: Read and Write Control Sequence

control logic is assumed to have a unit delay associated with its output. This assumption implicitly associates a delay-latch with each gate's output as discussed earlier in Section 2.2.4. The numbers on the inverters and buffers in Figure 2.25 indicate the number of unit delays of the inverter/buffer chain in that path. The read and write timing diagram for these signals is shown in Figure 2.26. For the purpose of illustrating the control signal sequence, the timing diagram takes into account only the delays introduced by the inverters and buffers. For more detailed information on the operation, please refer to [69].

A static boolean equivalence checker will not be able to verify such a circuit. In Figure 2.25, the static logic cone feeding the *wen* signal of the write pass transistors is 0 even though the circuit write operation functions correctly. Static analysis of the bitcell node *Data* generates a model that

cannot be written to. This is because the correct model behavior of the write operation is dependent on selecting an additional state variable (in addition to the 4 tag memory bits) in the input cone to *wen*. Without this state variable being identified and made part of the extracted model from the circuit, no simulation or verification technique will be able to prove that the circuit correctly implements the write operation. Current boolean equivalence checking technology, that rely on comparing stable outputs of logic cones and that do not allow additional state variable identification during model extraction, will fail for these self-timed circuits.

It is critical to draw the distinction between the models extracted from a circuit implementation and the technology underlying the verification technique. The custom implementation is a netlist of transistors from which a gate-level or switch-level simulation model [11][12][13][18][19][76][111] is extracted. We use switch-level models in our verification methodology which are similar to the models generated in [19]. These switch-level models represent a MOS circuit as a network of capacitive nodes connected by resistive transistor switches. The node capacitances and transistor strengths are represented by discrete sizes and strengths respectively. Many important detailed effects arising from circuit structures such as ratioed and precharged logic, bidirectional pass transistors, and stored charge are accurately modeled. A transistor circuit is partitioned into channel-connected sub components (CCS) and the ternary-valued steady-state response of each node in a CCS is computed. Since the underlying domain is $\{0, 1, X\}$, the state of each node m is encoded by a pair of bits $(m.1, m.0)$. A set of output nodes can be associated with a CCS that drive the gates of transistors in other CCS. Each node's excitation is a ternary valued function of

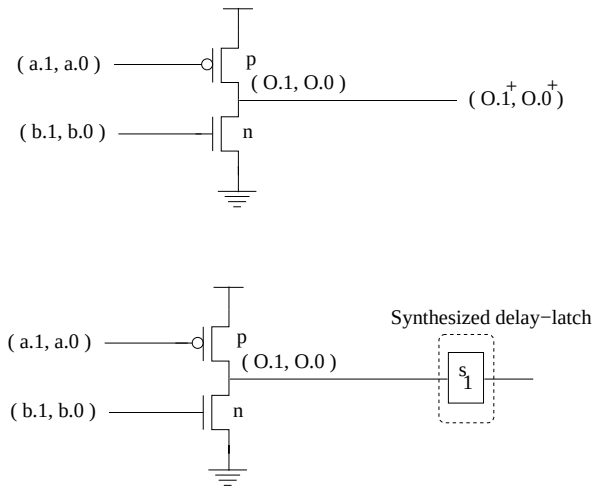


Figure 2.27: Tristate Buffer

its own state and all other node states (input and internal node states) in the CCS. Note that the CCS is the *element* for the event-driven simulation algorithms described in Section 2.3.

The ternary valued functions obtained for a tri-state node O shown in Figure 2.27 are given below. If we assume that both the p and n transistors are of equal strength and O^+ denotes the next-state of O, we obtain

$$\begin{aligned} O.1^+ = &\quad a.0 \vee (a.1 \wedge b.0 \wedge O.1) \\ O.0^+ = &\quad b.1 \vee (a.1 \wedge b.0 \wedge O.0) \end{aligned}$$

If we assume that the p transistor is stronger than the n, then

$$\begin{aligned} O.1^+ = &\quad a.0 \vee (a.1 \wedge b.0 \wedge O.1) \\ O.0^+ = &\quad (a.1 \wedge b.1) \vee (a.1 \wedge b.0 \wedge O.0) \end{aligned}$$

and if the p transistor is weaker than the n, then

$$\begin{aligned} O.1^+ = &\quad (a.0 \wedge b.0) \vee (a.1 \wedge b.0 \wedge O.1) \\ O.0^+ = &\quad b.1 \vee (a.1 \wedge b.0 \wedge O.0) \end{aligned}$$

In all these formulas, there is a present-state and next-state variable associated with the node O. This formulation of the next-state function implicitly associates a state variable with every CCS node. Assuming a buffer to consist of two CCS (two inverters in series) and that each CCS has just one output state node, there are a total of 85 state nodes in the control logic portion of the circuit (38 buffers and 9 inverters) shown in the left of Figure 2.25. We could have derived a switch-level model with the same simulation semantics by associating a transport or inertial delay of $(d_R = 1, d_F = 1)$ for each of the 85 CCS output nodes. The modified circuit switch-level model would have consisted of a synthesized delay-latch for $(d_R = 1, d_F = 1)$ at each of the CCS output nodes. Now each CCS output node's next-state function can be treated as purely combinational that does not depend on its own previous state. For example, the modified delay-latched tri-state buffer is shown in the bottom half of Figure 2.27 and the corresponding functions for node O is

p is same strength as n

$$O.1 = a.0 \vee (a.1 \wedge b.0)$$
$$O.0 = b.1 \vee (a.1 \wedge b.0)$$

p is stronger than n

$$O.1 = a.0 \vee (a.1 \wedge b.0)$$
$$O.0 = (a.1 \wedge b.1) \vee (a.1 \wedge b.0)$$

p is weaker than n

$$O.1 = (a.0 \wedge b.0) \vee (a.1 \wedge b.0)$$
$$O.0 = b.1 \vee (a.1 \wedge b.0)$$

Any switch-level model that considers every node as a potential state vari-

able implicitly associates a delay with that node. By extracting a state-based switch-level model from the circuit in Figure 2.25, symbolic simulation is now sensitive to the dynamic behavior of the circuit and is able to verify the write operation. However, note that this switch-level model has more information than necessary for verifying the write operation in this circuit. A switch-level model that only identified the output of the 9-unit buffer (input to the *wen* AND gate) as a state variable or the modification of the circuit switch-model to incorporate the delay-latch only at the output of the 9-unit buffer would have been sufficient for modeling the correct write behavior of the circuit. Automatic deduction of these state variables as described in Section 2.2 in the implementation would enable the adaptive modification of switch-level models extracted from the circuit implementation.

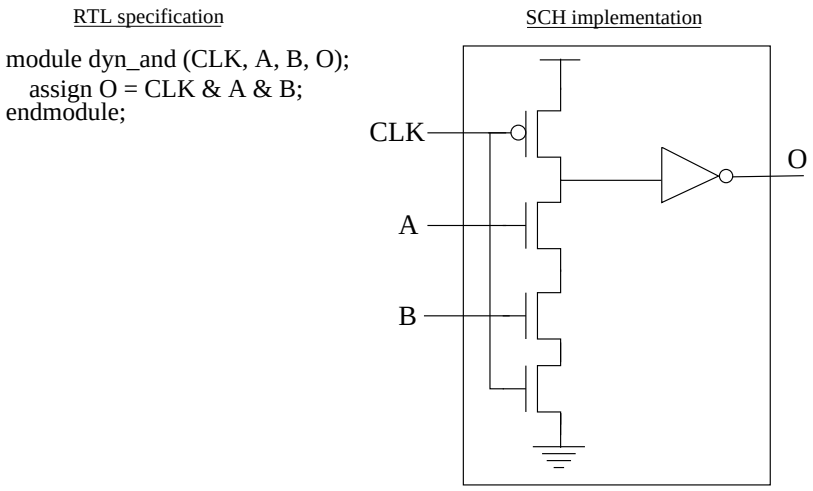


Figure 2.28: Dynamic logic circuit

2.6 The importance of accurate event-ordering

Consider a dynamic logic circuit and its RTL specification shown in Figure 2.28. The circuit computes the function $CLK \wedge A \wedge B$. Assume that the primary inputs are fed by dynamic logic that is precharged using CLK. Primary input A is precharged to 0 while B is precharged to 1. The primary input B is precharged to 1 to enable a faster rise time at the output during evaluation. A static boolean equivalence checking tool would verify that the dynamic circuit implements the RTL specification. Now consider delay values for the primary inputs as shown in Table 2.8.

Table 2.8: Rise and Fall delays for AND dynamic logic

Primary Input	Rise delay	Fall delay
A	5	2
B	4	4

A symbolic simulation of the circuit will verify that the circuit implements

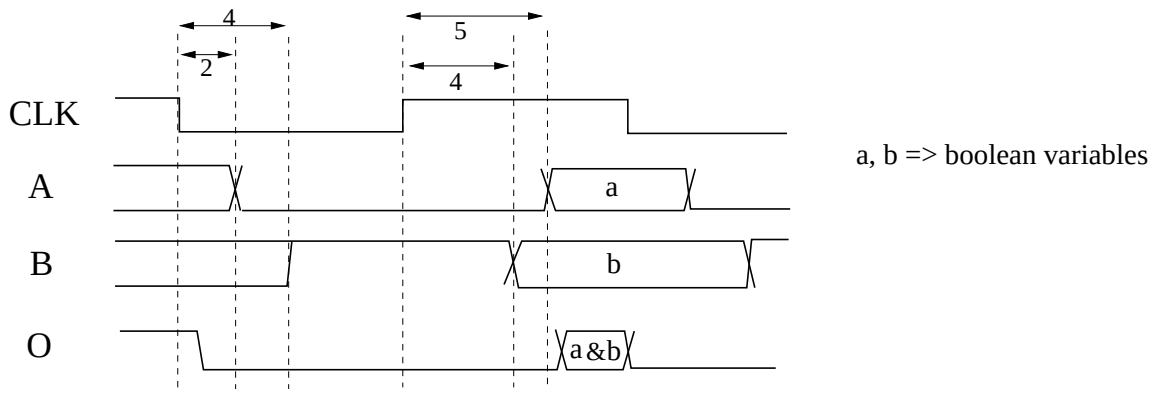


Figure 2.29: Timing diagram for user-specified rise/fall delays

the RTL specification as depicted in Figure 2.29. Consider a different set of delays as shown in Table 2.9. The rise delay for A is now faster at 4 units and the fall delay is slower for B at 5 units. Although, the change in delays is a single unit, the dynamic circuit no longer implements the RTL specification and the verification fails. The symbolic simulation tool will now produce the waveforms shown in Figure 2.30. Consider the circuit in Figure 2.28 instantiated in a larger design as shown in Figure 2.31. The boolean function computed at the outputs driven by O is actually dependent on the delays at the internal nets A and B. Using a static boolean equivalence checker is akin to validating the design using zero-delay simulation. This results in a false positive since there could be real event orders that were not considered during the verification. By symbolically

Table 2.9: Modified Rise and Fall delays for AND dynamic logic

Primary Input	Rise delay	Fall delay
A	4	2
B	4	5

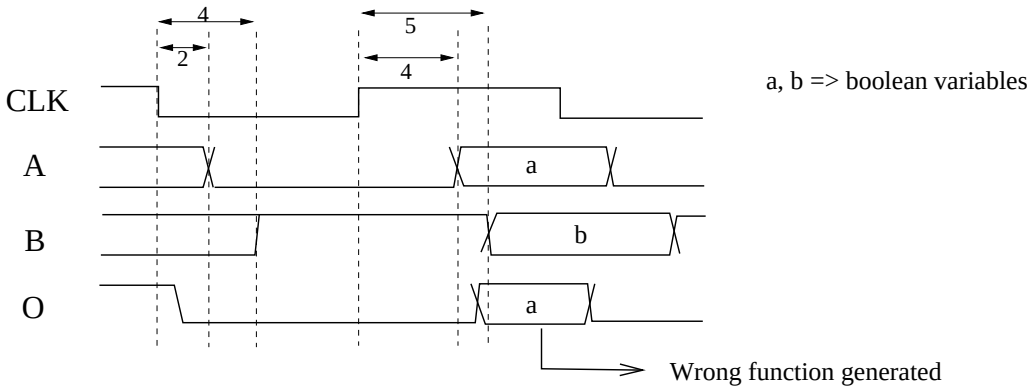
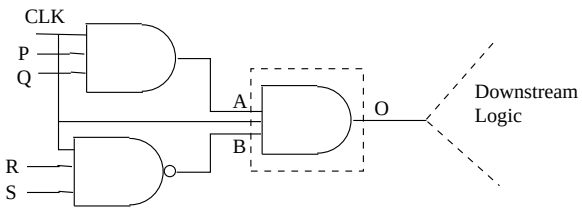


Figure 2.30: Timing diagram for modified rise/fall delays

simulating the circuit with delays, we prove that the circuit implements the specification under a set of specified delays. This not only enables a more accurate verification but also enables the designer to analyze his circuit for different event orders. Most design methodologies will have circuit design rules to ensure that such circuits with races are not designed. However, there will always be circumstances in which the violation of those rules is warranted. Any validation technique must have the capability of addressing these concerns. A symbolic simulation methodology using accurate event orders is essential for verifying high-performance designs.

2.6.1 Illustration of an STE validation

For illustration, we present an example of an RTL vs. implementation validation using symbolic simulation for a simple design. Consider the RTL design shown in Figure 2.32. The RTL represents a two-phase clocked design that is to be custom built. The design models an AND gate driven by two slave latches A.L2 and B.L2 and whose output is latched in a third master latch C.L1. With this RTL as a specification, a custom



RTL specification

```
module (CLK, P, Q, R, S, .....);
  assign O = CLK ? P & Q & ( ~R  |  ~S ) : 1'b0 ;
  .........
  .........
endmodule;
```

Figure 2.31: Dynamic logic instantiated in a combinational circuit

implementation of the RTL design is built as shown in Figure 2.33. For example, the A.L1 master latch in the RTL corresponds to the *D.L1* master latch in the implementation. A typical implementation of a master-slave latch pair is shown in Figure 2.34. The master latch consists of an internal state-holding node *L1SN*, a data input *DIN*, a clock input *C1*, and a data output *L1OUT*. To verify such a latch in STE, we set up an antecedent A and consequent C as illustrated in Figure 2.35 and then use STE to prove $A \Rightarrow C$. The grey shaded portions represent places where no assumption is being made in the antecedent or no check is being performed in the consequent. A similar assertion would be run for the slave latch to verify the complete master-slave pair. This verification has only proved that the latches are correct with respect to the informal property we have specified for a latch. We have not yet proved that the latch pair *D* behaves as predicted by the RTL. To achieve this we derive the properties from the RTL automatically [118]. For example, the antecedent and consequent for

RTL MODEL

```
module (C1, C2, Ain, Bin, Cout);
input Ain, Bin, C1, C2;
output Cout;
wire and_out;
reg A.L1, A.L2, B.L1, B.L2, C.L1, C.L2;
assign and_out = A.L2 & B.L2;
always @(C1 or Ain or Bin or and_out)
   if (C1)
      A.L1 = Ain;  B.L1 = Bin;  C.L1 = and_out;
always @(C2 or A.L1 or B.L1 or C.L1)
   if (C2)
      A.L2 = A.L1;  B.L2 = B.L1;  C.L2 = C.L1;
assign Cout = C.L2
```

Figure 2.32: RTL design representations

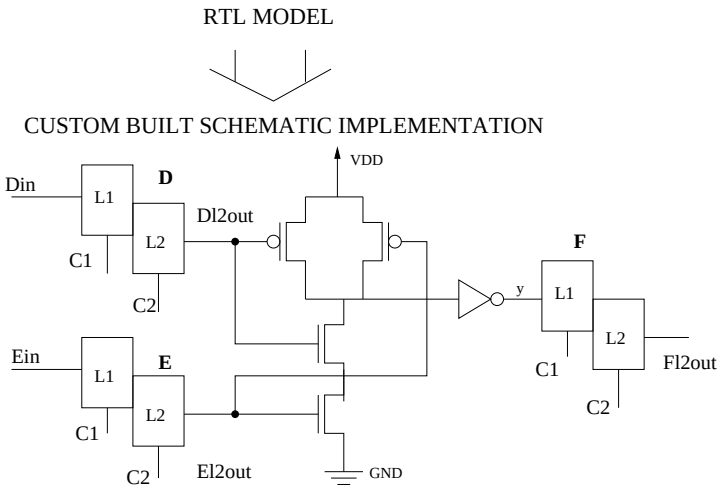


Figure 2.33: Custom implementation of design

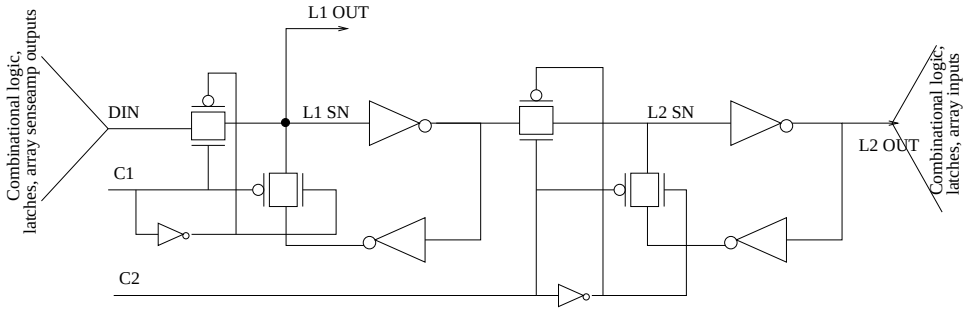


Figure 2.34: A Master-Slave Latch

the latch *D.L2* are derived from the corresponding RTL latch A.L2. This assertion is then mapped onto the implementation and it is the mapped assertion that is given to the symbolic trajectory evaluator.

2.6.2 Does the implementation realize the RTL specification?

The RTL design is first partitioned into a set of *checkpoints* [71]. These checkpoints are elements such as primary outputs, storage elements, and cutpoints about which properties can be stated and where states can be compared. Cutpoints are intermediate nodes that exist in both the schematic and the RTL models. They are used as checkpoints to reduce the complexity of the logic cones being checked. The RTL encodes a set of next-state functions for the checkpoints in the design. These functions are captured as STE assertions which are then checked to hold on the implementation.

Consider the RTL design shown in Figure 2.32. The checkpoints are the storage nodes A.L1, A.L2, B.L1, B.L2, C.L1, C.L2, and the primary output Cout. Each checkpoint has its own assertion. For example, the next-state function, in terms of RTL node names, for latch C.L1 automat-

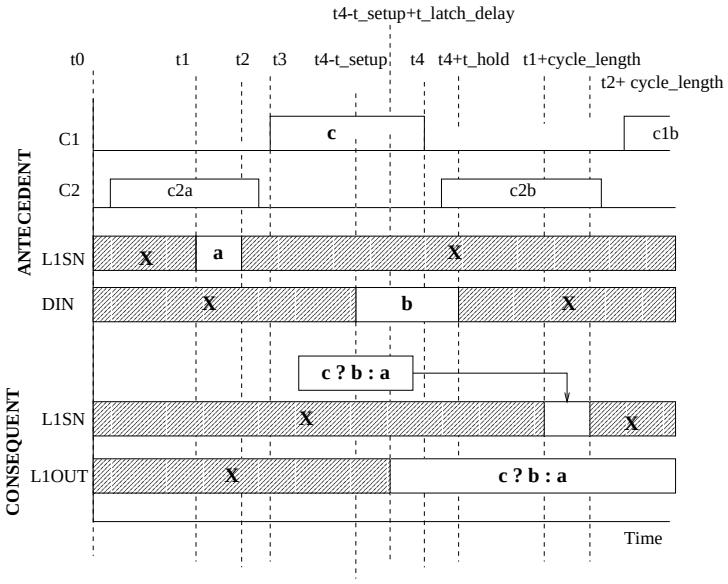


Figure 2.35: Latch verification using symbolic simulation

ically derived from the RTL model is as follows:

$$C.L1 = (C1 \ \wedge (A.L2 \wedge B.L2)) \mid ((\neg C1) \ \wedge C.L1)$$

This function forms the value part of the consequent for the assertion associated with it. The antecedent for the C.L1 assertion would involve driving inputs *C1* and *C2*, and the checkpoints *A.L2* and *B.L2* with symbolic variables *c1*, *c2*, *a*, and *b* respectively. Having obtained the assertion for C.L1, we now have to verify that it holds on the implementation shown in Figure 2.33. The RTL assertion is mapped to an implementation assertion by mapping nodes and values in the RTL domain to nodes and waveforms in the implementation domain. The C.L1 latch RTL assertion is mapped into an implementation assertion as shown in Figure 2.36. The implementation-mapped assertion for the C.L1 latch involves

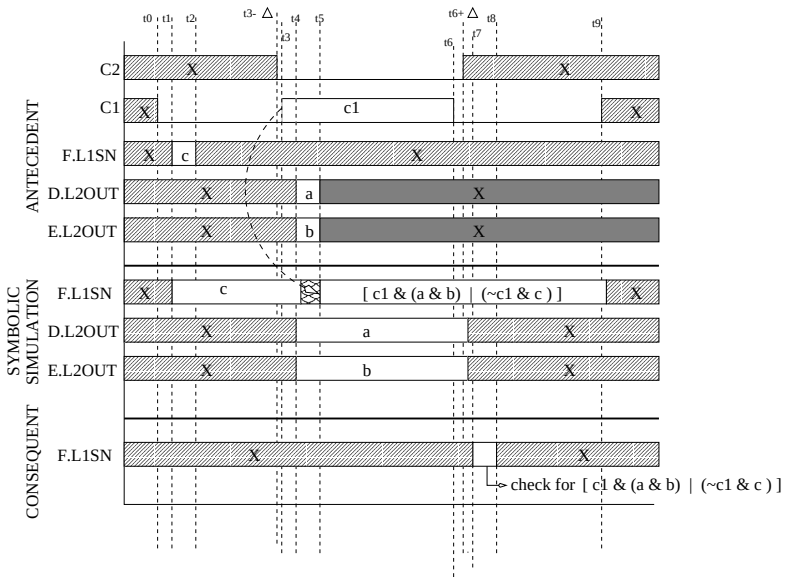


Figure 2.36: Antecedent and Consequent pair

1. Identifying the *F.L1SN* corresponding checkpoint in the implementation where states can be compared. This is called state-mapping.

2. Definition of the timing window (t_7, t_8) for the consequent check of the C.L1 latch next-state function. This time window is also used in the antecedent when logic downstream of the C.L1 latch is checked. The time windows are derived such that enough time is provided for the signal values to reach a stable value.

The *D.L2OUT*, *E.L2OUT* and *F.L1SN* latch nodes are initialized with independent variables a, b and c respectively, precisely at the time there is a stable feedback loop established in the latches after the corresponding clock goes to 0. The L2 latches in the implementation are initialized from t4 to t5 when the C2 clock is 0. The *F.L1* latch is checked precisely one clock cycle after the time that it was set up. The timing windows

are defined by considering the setup and hold times of the latches. The primary output *Fl2out* is verified by setting up the *F.L2OUT* latch node in the implementation and checking to see that the primary output is the same function as that of the *F.L2* latch storage node L2OUT (Figure 2.34).

Assertion mapping is done for all the checkpoints in the design. This is derived from the state maps provided by the user. The verification methodology can deal with different state encodings between the circuit implementation and the RTL specification. For example, consider the RTL specification of a pipelined register shown below.

RTL specification for a pipelined register

```
input C1, C2, xin, yin
output mout, zout
wire c, d
reg x, y, z, m /* Registers */
always @(C1 or xin or yin)
begin
   if (C1)
      begin
         x <= xin
         y <= yin
      end
end
assign c = ¬ (x ⊕ y)
assign d = x ∧ y

always @(C2 or c or d)
begin
   if (C2)
      begin
         z <= c
         m <= d
      end
end
assign zout = z
assign mout = m
```

The corresponding circuit implementation is shown in Figure 2.37. There

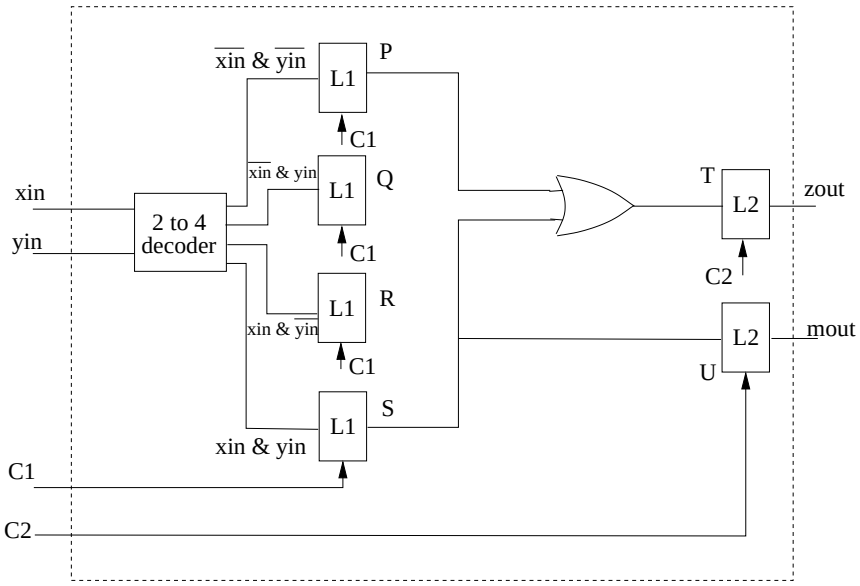


Figure 2.37: Circuit Implementation of Pipelined Register

are four state elements in the RTL specification whereas there are six state elements in the circuit implementation. To deal with the different state encodings between the specification and the implementation, the verification methodology requires the following state maps to be specified.

$$
\begin{aligned}
P &= \neg x \wedge \neg y \\
Q &= \neg x \wedge y \\
R &= x \wedge \neg y \\
S &= x \wedge y \\
T &= z \\
U &= m
\end{aligned}
$$

The left-hand side of the above equations depict nodes in the circuit im-

plementation while the right-hand side depicts nodes in the specification. The specification of the relationship between the encodings allows the verification of such circuits.

The implementation in Figure 2.33 is a correct implementation of the RTL in Figure 2.32 with respect to the assertion mapping only if the conjunction of all the checkpoint assertions are satisfied by the implementation. The design decomposition into checkpoints and the definition of the mapping functions are carefully done so that the final verification result can be established by composing the results of the individual checkpoint assertions.

2.7 Validation Methodology for Custom Memories

All custom memories on a PowerPC microprocessor[1] were validated using symbolic trajectory evaluation. Our objective was to expose any implementation errors. A typical custom memory consists of an array of bitcell storage nodes as shown in Figure 2.38. The verification methodology is shown in Figure 2.39. The first step in the methodology is to partition the design into checkpoints. Checkpoints for a memory design are all the state-holding elements, cutpoints and the primary outputs. The state holding elements constitute the latches and full keepers in the design. Cutpoints are intermediate nodes that exist in both the implementation and the RTL models. They are used as checkpoints to reduce the complexity of the logic cones being checked.

Then, the state mapping between the RTL and the implementa-

[1]compliant with IBM's PowerPC instruction set architecture

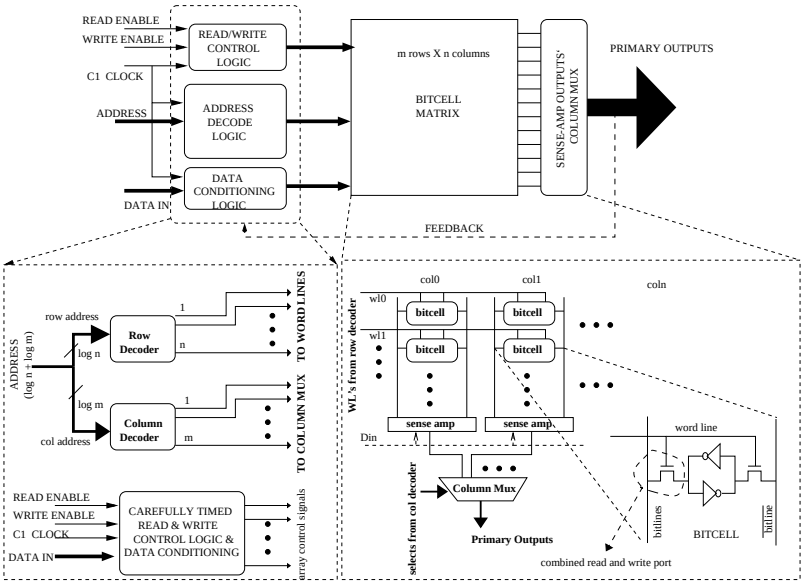


Figure 2.38: Custom Memory

tion is carried out. In cases where the state machine encoding differed between the RTL and the schematic, the exact relationship between the nodes in the RTL and schematic are identified and specified. Often, this mapping is non-trivial because the schematic implementation hierarchy is not reflected in the RTL model and may require detailed knowledge of the circuit implementation.

After the state maps have been identified, the switch-level model is augmented with inertial delay information so that the resulting ordering of events between critical signals is consistent with SPICE simulations. Each node in the circuit implementation has an associated rise and fall inertial delay. The first implementation of the simulation engine incorporated a delay circuit **??** for each node in the circuit model. A second implementation of the simulation engine incorporated a delay circuit for each node

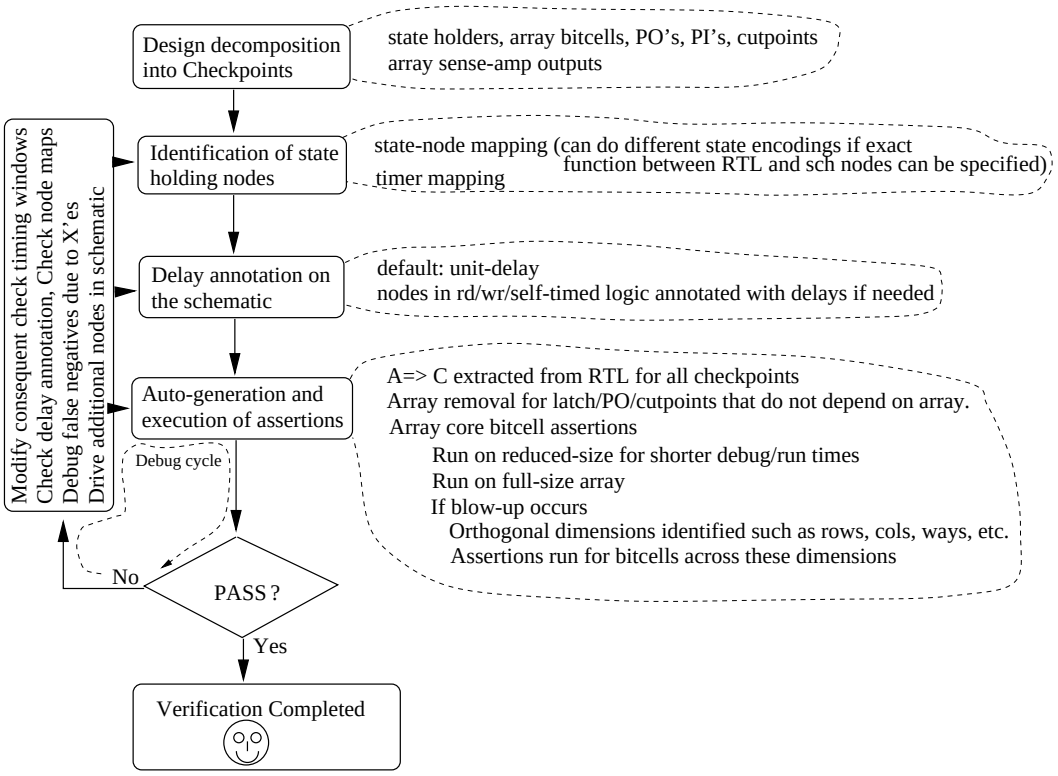


Figure 2.39: Verification Methodology

that was logarithmic in the size of the inertial delay. Each inertial delay was modeled as a Moore machine (elaborated in Section 2.2.3) and synthesized to a counter-like circuit structure. A number of experiments were conducted to compare the efficiency of the simulation engine using both the above approaches and the results are discussed in Section 6.3 of Chapter 6.

After the correct set of inertial delays have been annotated for each node in the circuit implementation, the RTL model is then analyzed to obtain the assertions. While verifying checkpoints other than the memory bitcells, a switch-level model without the memory core is created. This reduces model load time and debug time for assertion failures. Once all

the primary outputs and latches are verified, the memory core is then verified. The initial verification runs are done on a model that has only a few number of bitcells. Typically, a row or column of bitcells is selected. Once they are verified, they are run on the full-size memory. In cases, where assertions could not be run on the full-size memory due to BDD blow-up, different sized models are created and then verified using an orthogonal set of assertions. For more details on each of these steps, the reader is referred to [70].

2.7.1 Extra Circuit Implementation Inputs

So far we have assumed that the number of primary inputs to a specification is exactly the same as the number of primary inputs to the corresponding circuit implementation. This assumption may be violated due to design and manufacturability considerations where the circuit implementation has more primary inputs than the specification. An example of extra primary inputs that are not modeled in the specification are the timing and edge control inputs. These inputs are used in custom memories to control the timing of control signals required for reading and writing.

Consider the RTL specification of a design as shown below.

RTL specification

input a, b, c, d

output O

assign O = F (c, d)

From the RTL specification, it can be inferred that the primary output O

is a function of the primary inputs c and d (and not a and b). Consider a circuit implementation of the RTL specification as shown in Figure 2.40. The implementation has a decoder that decodes the primary inputs a and

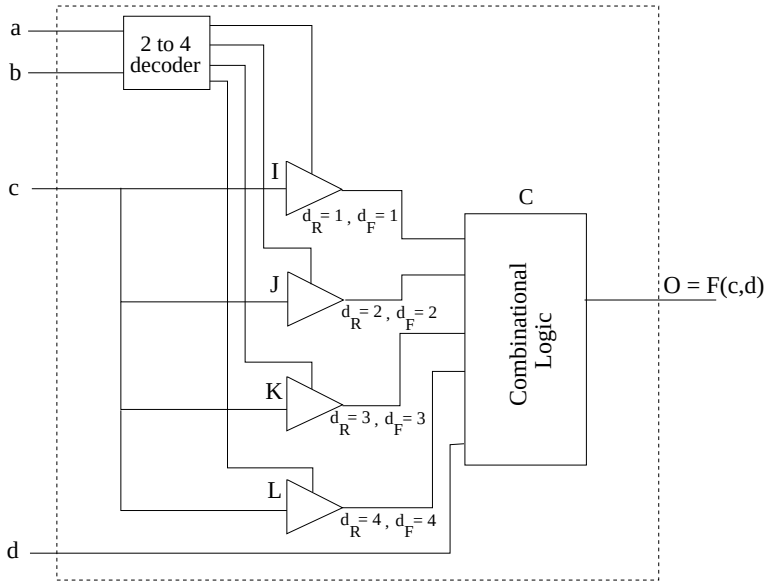


Figure 2.40: Extra Inputs in Circuit Implementation

b and enables any one of the four tristate buffers from c to the logic block C. Each buffer corresponds to a different delay path that delays input c by different amounts prior to reaching C. However, regardless of which path is used to provide c to C, the logic functionality of the circuit remains unchanged. This is the reason primary inputs a and b are not driving any logic in the RTL specification.

Since the RTL specification does not include the dependency on inputs a and b, these inputs would normally be set to an X. The X on inputs a and b results in the output O becoming an X in the circuit implementation regardless of what the inputs c and d are. This X is a false

negative and is automatically avoided in our verification methodology by identifying the extra inputs a and b in the circuit implementation as special purpose additional required inputs. The assertion generator automatically adds the user-specified additional inputs to the antecedent of all generated assertions thereby eliminating the false negatives.

Chapter 3

State Mapping vs. Product Machine Approaches

In the earlier chapter, our verification methodology was dependent on the apriori identification of the state mapping functions between the RTL and the implementation. In this chapter, we show why this is necessary for a sound verification strategy. Symbolic simulation is used to establish equivalence between two descriptions by either the Product Machine Approach [60] (PMA) or the State Mapping Approach [72] (SMA). In the following section, we compare SMA to PMA from a bug coverage and verification complexity perspective.

A state of a circuit is defined as a tuple of values on all its storage registers. A state map specifies the correspondence between states in the RTL and states in the schematic circuit. It has been informally argued that state maps may not be necessary and that even if state mapping is possible, the same level of verification coverage can be obtained without them using symbolic simulation. We show that if state mapping is *possible* between an RTL and the corresponding custom implementation, then it must be done to expose Crossover Bugs (CB's). Equivalence checking using SMA does not distinguish between test, debug and functional modes and implicitly covers *all modes* of operation and thereby results in achieving a higher coverage. This is particularly noticeable in scan-based designs

and pipelined memories because of the added complexity due to potential presence of CB's [73].

3.1 PMA versus SMA

In this section, our verification process is presented and the underlying techniques of PMA and SMA are described and compared. The

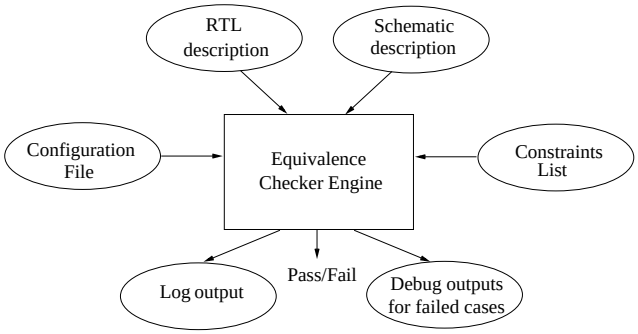


Figure 3.1: Verification flow

equivalence checker in Figure 3.1 takes an RTL and a transistor circuit description. Both the RTL and the transistor descriptions model scan functionality in the design. As far as equivalence checking is concerned, the verification process does not differentiate between the functional and non-functional modes of operation.

3.1.1 PMA

In PMA [60], the RTL and switch-level models are combined to form a single model, say M as shown in Figure 3.2. The primary inputs of the RTL model and switch-level models are connected together in M and corresponding primary outputs are connected together using equivalence

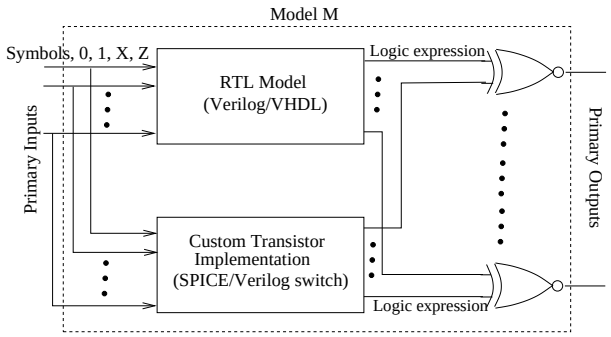


Figure 3.2: Product Machine Verification

gates. The outputs of the equivalence gates are the primary outputs of M.

Symbols are applied at the inputs and M is simulated over a few cycles. Symbolic expressions are propagated to M's primary outputs and are checked for the value 1. If a value of 1 does not result, a difference between the RTL and circuit implementation is reported. Due to the complexity of the control logic and the size of custom memories, the symbolic expressions can very quickly exceed available memory. In such cases, the verification engineer assigns symbols judiciously to a subset of the inputs and the rest of the inputs are assigned scalar values from {0, 1, X, Z }. M is again simulated over a few cycles while checking the outputs. Based on the design, a set of symbolic test cases is manually derived and if the model M passes this test suite, the RTL is often declared to be equivalent to the implementation [60]. In PMA, both the RTL and the switch-level models are in memory and are simulated simultaneously.

3.1.2 SMA

In this approach, the RTL design is partitioned into *checkpoints* as described in Section 2.6.2 in Chapter 2. SMA involves establishing a map

from states in the RTL to states in the implementation. In Figure 3.3, the RTL node *latch_op* is mapped to the hierarchical circuit node *I0.I1.I2.lout* in the implementation. The RTL to implementation maps could be one-to-

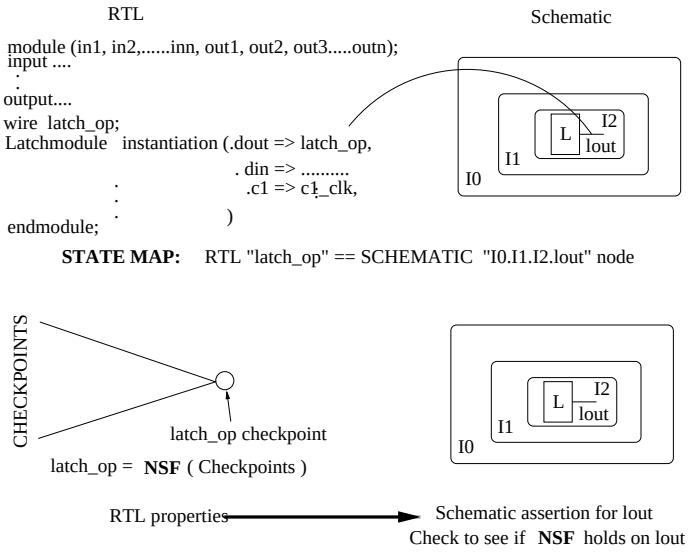


Figure 3.3: State Map based Verification

one or one-to-many. After state mapping is done, the properties encoded by the RTL are automatically extracted and the switch model is symbolically simulated [71]. This enables us to check whether the Next-State Functions (NSF) of the RTL checkpoints as implemented by the circuit are correct or not. In Figure 3.3, the next-state function for the RTL checkpoint *latch_op* is **NSF** and is a function of a set of RTL checkpoints. For a correct custom implementation of *latch_op*, *I0.I1.I2.lout* must implement **NSF**.

In SMA, the state maps are established before any verification begins. For a typical custom memory, this would include identifying all the state holding nodes in the circuit and listing the state maps. During simulation, the only model present in memory is the switch model and symbolic

simulation proves that the next-state function and thereby the state transitions as predicted by the RTL model holds on the implementation.

The main drawback with SMA is the effort associated with the identification of the state maps. Also, SMA will not work where no mapping is possible between the RTL and the circuit implementation. This typically occurs in synthesized designs where the mapping is lost during phases of the synthesis algorithm. However, in scan-based embedded memories, test operations are modeled in the RTL for test simulation purposes and therefore state maps do exist. Also, since memories have to be modeled in the RTL to capture state (such as two-dimensional registers in Verilog), there always exists a state map for the memory as well. In any high-level behavioral model of a design that models reads from and writes to memories, the memories have to be modeled using a data structure that encodes the address and data pairs. In practice, it is always the case that there exists a bijective mapping for the memory storage elements from the high-level behavioral model to the memory implementations.

3.2 Corner-case Crossover bugs

For a microprocessor chip, all on-chip operations can typically be divided into three categories.

- Normal functional operation
- Debug or Test operation
- Low power (sleep, nap, etc.) operation

To implement these operations, an appropriate clocking scheme is designed.

3.2.1 Clocking technique for full-scan designs

Most scan-based designs have a flip-flop primitive that is often designed as shown in Figure 3.4. The input-output behavior of the flip-flop

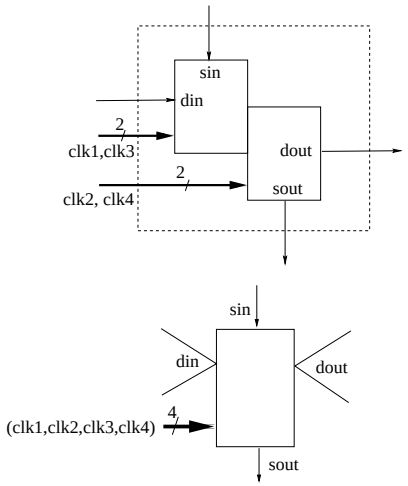


Figure 3.4: Scan Flip Flop

is determined by the activation of any of the four paths from input to output as shown in Figure 3.5. The multiplexing of these paths is controlled by a set of signals and are typically the clocks. For example, as shown in Figures 3.4 and 3.5, clocks *clk1*, *clk2*, *clk3* and *clk4* control the four different paths from *din* and *sin* to *dout* and *sout*. A clocking methodology with precise relative timings that enable realization of both functional and debug/test operations is shown in Figure 3.6. In the temporal domain, the clocking scheme can be divided into *scan-only* (SO), *functional-only* (FO), *scan-to-functional* (S2F) and *functional-to-scan* (F2S) clocking modes. Normal functional operation would involve only the FO clocking mode whereas the debug and test operations would involve all the four clocking modes. In Figure 3.5, a sequence of *clk1* and *clk2* controls the

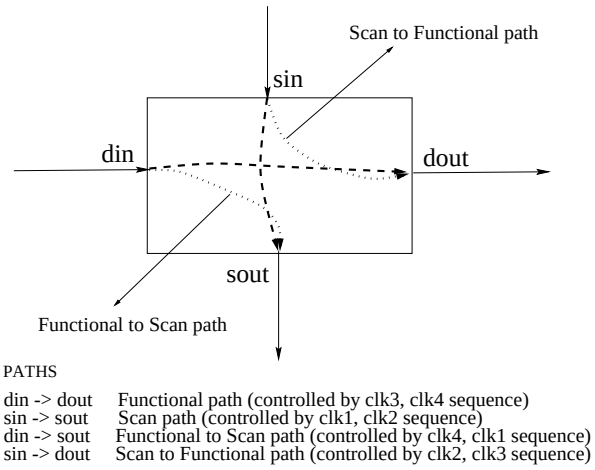


Figure 3.5: Functional and Scan Paths

sin => *sout* path and a sequence of *clk3* and *clk4* controls the *din* => *dout* paths. The *sin* => *dout* and *din* => *sout* paths are enabled during the S2F and F2S clocking modes shown in Figure 3.6.

3.2.2 Crossover Bugs

Definition 3.2.1. *Crossover Bugs* (CB's) are logic bugs that can occur when a scan-based design transitions from normal functional operations to debug/test operations or vice-versa.

Lemma 3.2.1. *S2F and F2S clocking modes are necessary to detect CB's in scan-based designs and cannot be detected by FO or SO clocking modes.*

For PMA to detect CB's, it must include the S2F and F2S clocking modes explicitly during symbolic simulation. Specific symbolic simulation test cases need to be simulated so that the logic paths to propagate CB's to the primary outputs are sensitized. Therefore, the verification coverage is wholly dependent on the effectiveness of the test suite. A brute force

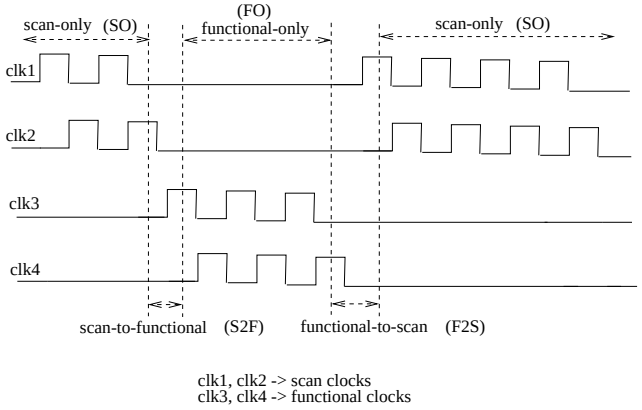


Figure 3.6: Two phase clocking for functional and test operations

approach of PMA would be to apply symbols to *all* inputs and simulate M in Figure 3.2 over a few cycles. However, PMA is very expensive in time and memory and for most reasonably sized industrial scan-based designs, it will not complete.

State mapping techniques automatically enable the coverage of all CB's. This is because the SO, FO, F2S and S2F clocking modes are implicitly covered by the next-state function checks on the switch-model using symbolic simulation. All the logic paths that affect a checkpoint are captured by the next-state function.

CB's that are not detected before tapeout, may manifest themselves when manufacturing test patterns are first applied to the chip. This will result in additional masks and re-engineering efforts. Also, there is a high probability that functionally good chips are labeled defective due to CB's and can lead to lower yields. However, CB's can be effectively caught earlier in the design cycle (thus saving time and money) if SMA based symbolic simulation is done.

Lemma 3.2.2. *Any equivalence checking methodology between RTL and transistor implementations for scan-based designs must either implicitly or explicitly incorporate the S2F and F2S clocking modes for CB detection.*

Note that Lemma 3.2.2 deals with more generic scenarios of how one can detect CB's whereas Lemma 3.2.1 specifies RTL vs. implementation equivalence checking without being specific about the verification technology used. In the following section, we will compare the effectiveness of SMA with PMA in exposing specific CB's.

3.2.3 CB Detection Capability using PMA and SMA

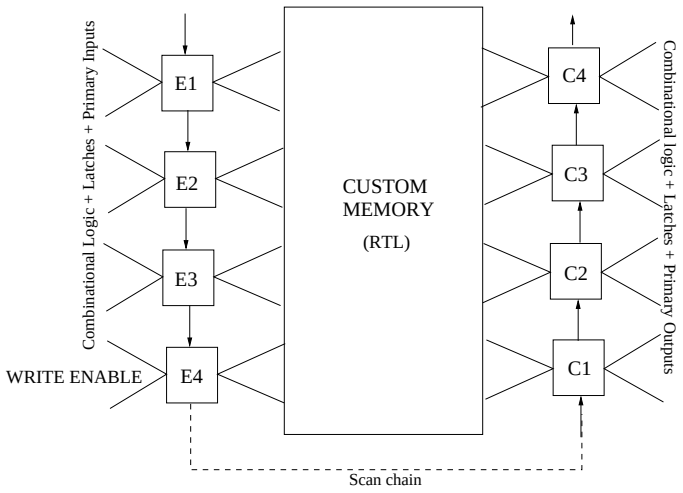


Figure 3.7: Pipelined RTL implementation

Consider an RTL specification of a custom memory as shown in Figure 3.7. Assume that due to performance considerations, the memory is pipelined and has banks of pipelined registers connected in a scan-chain on the input and output. In Figure 3.7, let the latch E4 be a write enable

that determines whether a write occurs to the memory or not. Consider a custom implementation as shown in Figure 3.8.

Assume that the inverters before and after E4 were added by the designer for buffering and/or other electrical reasons. This implementation is equivalent to the RTL for only the SO and FO clocking modes because the CB's do not get sensitized in the other two modes of operation. However, they are not equivalent for the S2F mode that can occur during a manufacturing test operation. In SMA, this bug would be detected since the next-state function of the E4 register in the implementation would be different from that of the RTL predicted next-state function for E4. This verification would require a variable for E3, a variable for E4 and a number of symbolic variables equal to the size of the support set of the write-enable function. The number of symbolic simulation cycles would be one. In PMA [60], four scan-only symbolic simulation cycles and one

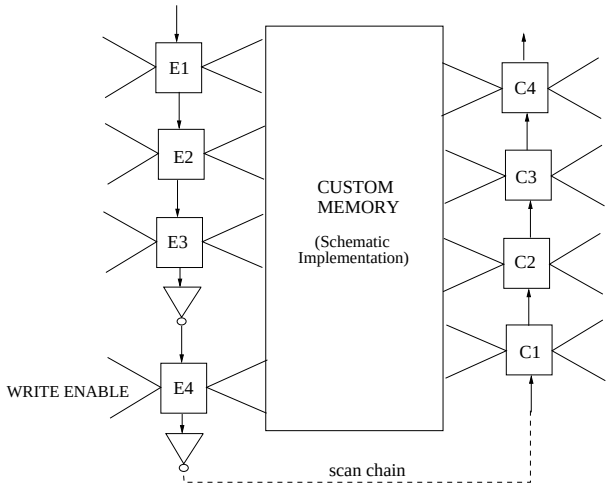


Figure 3.8: Scan to Functional Implementation Bug

functional symbolic simulation cycle would be required. In addition, the

number of symbolic variables required would be four variables and a number of variables equal to the size of the support set of the write-enable function.

Now consider another custom implementation of the same RTL specification as shown in Figure 3.9. This implementation is also equiva-

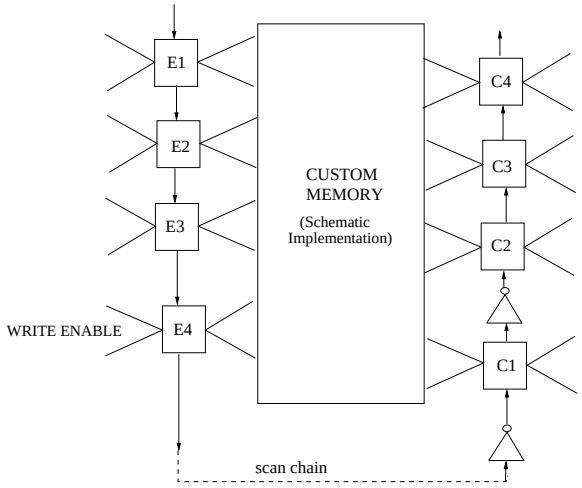


Figure 3.9: Functional to Scan Implementation Bug

lent to the RTL for only the SO and FO clocking modes. However, they are not equivalent for the F2S clocking mode that can occur during a manufacturing test operation. A typical test operation would be a write to the memory followed by a read and then a scan out procedure. This bug will manifest itself during this operation.

There are two ways that this type of bug can be detected using PMA. The first approach would be to scan in eight symbolic variables using eight scan symbolic simulation cycles, applying a functional clock cycle and then checking all the primary outputs. The second approach would be to scan in four symbolic variables using four scan symbolic simulation cycles,

applying a functional clock cycle and then executing a scan out operation using four additional scan simulation cycles. Either of these approaches would detect the bug in register *C1*. However, the symbolic simulation vector has to be targeted towards a specific bug and in the worst case the time and memory complexity grow linearly with the length of the scan chains. The intermediate logic expressions at the memory outputs grow exponentially with the length of the address and control bits and these need to be propagated to the primary outputs. With embedded custom memories having scan-chain lengths ranging in the thousands, PMA is ineffective at catching crossover bugs.

Lemma 3.2.3. *Given a custom memory design with r rows, c columns, i primary inputs, an embedded scan chain of length n and a latency of l cycles, the simulation time complexity of detecting CB's is* $O(n+l)$ *simulation cycles in PMA.*

In SMA, the verification time complexity to detect similar bugs is independent of the length of the scan-chains. Moreover, CB detection in SMA can be split into two verification phases. The first phase verifies all the registers in the scan-chains that are not driven by the memory outputs. The state of the memory in the design is not required for the first verification phase and therefore, the memory may be completely removed from the model, thereby resulting in huge savings in memory requirements during verification. This is not possible in PMA. The second phase then verifies all the other scan registers. In the second phase, the memory state is setup using symbolic indexing [5]. SMA allows the partitioning of the CB detection process into multiple verification phases, thereby avoiding the exponential blowup at the memory outputs that occurs in PMA. See

Table 3.1: PMA vs. SMA Verification Complexity

	PMA	SMA
Bool Vars	$O(n + (i \times l))$	$O(n + i + \log(r \times c))$
Sim Cycles	$O(n + l)$	$O(1)$

Table 3.1 for a comparison of the SMA and PMA crossover bug detection complexities. Note that even though both PMA and SMA seem to require $O(n)$ boolean variables for CB detection, it is the exponential sizes of the intermediate boolean functions in PMA that render PMA ineffective and hence mostly unusable in practice.

Given a scan-based custom memory where both brute-force SMA and PMA fail to complete, SMA can be made to complete and achieve 100% CB coverage using multiple-phase SMA verification. This is because each of the verification phases does a complete exhaustive check. To enable PMA to complete, one needs to have a judicious combination of symbols and scalar values on the inputs, thereby losing CB coverage.

3.2.4 Experimental Results

An F2S CB and an S2F CB were injected in each of four scan-based custom memories from a MPC7455[1] design. All injected CB's were detected using our SMA based approach. The verification runs were carried out on a 360MHz UltraSPARC-II workstation running SunOS 5.6 with 1024 MB of RAM. The time and memory usage for each of the verification runs is shown in Table 3.2. The time and memory utilization represent the summation of the verification times and memory required for the detection

[1]compliant with IBM's PowerPC instruction set architecture

Table 3.2: SMA-based PowerPC Custom Memory Verification

Custom Memory	Bitcells	Latches	Time (mins)	Mem (M)
A	73728	1346	29	399
B	24576	1612	2	125
C	24576	935	3	126
D	21824	1192	18	466

of both F2S and S2F CB's. Memories B and C require much less memory compared to A and D due to the simpler **NSF** checks that are needed to detect the injected CB's.

A second set of experiments were conducted where the SMA approach was compared to a commercial tool utilizing the PMA technique. This was done primarily to evaluate the commercial tool's capabilities/feature set and its ability to detect logic bugs. A total of seven custom memories were used and specific logic and crossover bugs were injected into both the RTL and the custom circuit implementation. Some of the representative bugs that were injected were

- Modified the RTL specification where the scan clocks for two different registers where interchanged and therefore coming from different clock regenerators.
- Added an extra inverter in the circuit implementation for one specific write timing path into the memory.
- Created an open word-line in the implementation that fed a specific memory row such that a write could not take place into the memory.

- Added two inverters, one in the scan-in path to a latch that latches the write enable and and another inverter before the scan-out pin of the memory (cross-over bug).
- Modified the RTL specification such that the control logic was incorrectly coded up as compared to the control circuitry in the implementation.

The commercial tool was able to detect one logic bug in one custom memory over an evaluation period of seven man-months, whereas our verification methodology using the SMA approach detected all the injected bugs.

Chapter 4

Dynamic Circuit Equivalence Checking and Design Constraint Derivation

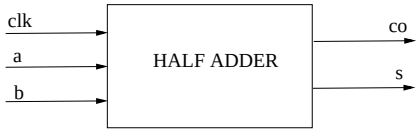
This chapter deals with the equivalence checking peculiarities that one might face when verifying dynamic circuit implementations. Ideally, the correctness of the circuit implementation with respect to the RTL specification must be established under all possible input sequences. In the verification methodology discussed earlier, we did not make any assumptions about the behavior of the primary inputs and assumed them to be completely non-deterministic. In most cases, this is not possible as the logic and/or underlying circuit implementation is designed to work correctly only under a restricted environment. Therefore, assumptions about the environment that constrain the behavior of the primary inputs are necessary in order to establish equivalence. The abstract codification of these assumptions to model an environment's behavior is a constraint.

Constraints restrict the set of possible trajectories that the circuit can go through. This implies that the circuit implementation is equivalent to the RTL only for a subset of the reachable state space. In other words, the circuit's environment must satisfy the assumptions made during equivalence checking in order for the RTL specification to be predictive of silicon behavior. Correctly constraining the environment is necessary in order to obtain correct results from any verification tool. Incorrectly over-

constraining or under-constraining the environment could result in false negatives or positives.

4.1 Dynamic Circuit Equivalence Checking

Consider an RTL specification of a dynamic half adder as shown in Figure 4.1. The adder adds the two inputs a and b when the clock signal, clk, is asserted. Both the sum s and the carry-out co outputs are modeled



RTL Specification

```
module halfadder (clk, a, b, co, s) ;
input clk, a, b;
output co, s ;

assign s = clk ? ( a ^ b ) : 1'b1 ;
assign co = clk ? ( a&b ) : 1'b1 ;

endmodule
```

Figure 4.1: Half adder specification

as continuous assignment statements using Verilog conditional operators. This is done to model the precharge values on the output s and co when the input clk is zero. Now, consider a custom transistor implementation of the sum output s of the RTL specification as shown in Figure 4.2. Due to circuit performance and electrical reasons dual-rail complementary signals are used for both a and b. The circuits driving a and b are shown in Figure 4.3.

The half adder implementation has two sets of complementary inputs a, a_b and b, b_b. Since they are complementary inputs that are driven

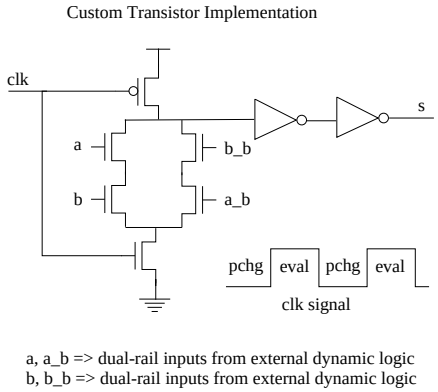


Figure 4.2: Custom circuit implementation of half adder output s

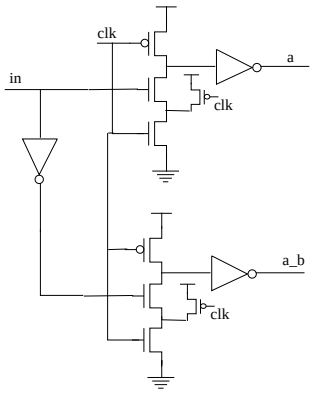


Figure 4.3: Dual rail complementary inputs

by the half-adders environment, constraints modeling these assumptions would be the following:

Constraint 1 (c_0): The a and a_b are one-hot during the evaluate phase only and are both low during the precharge phase.

Constraint 2 (c_1): The b and b_b are one-hot during the evaluate phase only and are both low during the precharge phase.

Let the user-specified constraint c_u that captures both the constraints c_0

and c_1 be specified as

$$c_u \;=\; (clkM \;\vee\; \bar{a}\,\overline{a\_b}\,\bar{b}\,\overline{b\_b}) \;\wedge\; (\overline{clkM} \;\vee\; (a \oplus a\_b).(b \oplus b\_b)) \quad (4.1)$$

Using the constraint c_u and the verification methodology in Chapter 2, the circuit in Figure 4.2 is proved to satisfy the half adder RTL specification in Figure 4.1. Functionality during both the precharge and evaluate phases is verified to be equivalent.

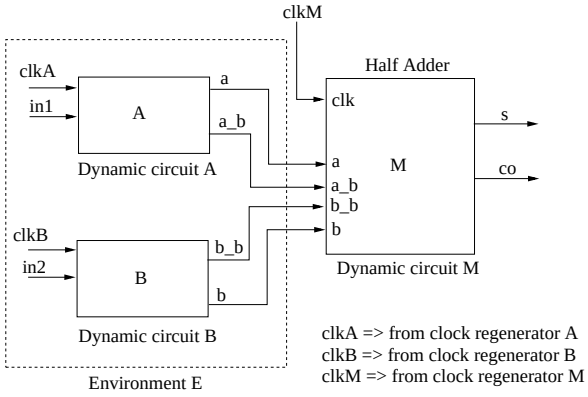


Figure 4.4: Half adder instantiation

Now consider an instantiation M of the half adder circuit as shown in Figure 4.4. We have three dynamic circuits (A, B and M) connected together in a domino logic structure. Inputs a, a_b, b and b_b of circuit M are driven by dynamic logic circuits A and B whose implementations are shown in Figure 4.3. Assume that the clocks *clkA*, *clkB* and *clkM* feeding the dynamic logic structures are fed by three potentially independent clock regenerators. Each clock regenerator is capable of being controlled independently of the others and this typically could occur during debug, test, or other non-functional operations.

Now, consider the following two cases.

Case A: clkA, clkB and clkM all fed from the same clock regenerator If $clkA$, $clkB$ and $clkM$ are all fed by the same clock regenerator, then all of them will have the same value at all points in time. Using constraints c_0 and c_1, the switch-level model and the RTL model are shown to be equivalent. Now let us consider an underlying circuit implementation change.

Case B: clkM fed by a different clock regenerator than that of clkA and clkB This is an entirely practical scenario in high performance designs because of electrical considerations arising out of various factors like clock skew, performance and so on. In this case as well, using constraints c_0 and c_1, the switch-level model and the RTL model are shown to be equivalent.

However, due to the clock configuration change of the underlying implementation, **Case B**'s verification results in a false positive due to the following scenario. When $clkA$ and $clkB$ are off and $clkM$ is on, the half adder circuit M evaluates while the inputs a, a_b, b, b_b are all precharged low. This produces the value 1 on the sum output s of the circuit implementation shown in Figure 4.4. However, the RTL in Figure 4.1, under the very same input signal values will produce the value 0 on the RTL sum output s.

Therefore, a simple decision that made changes to the clock balancing and connection of the regenerators in the underlying implementation resulted in the invalidation of an otherwise correct higher level verification result. The problem can be partially attributed to the difference in the levels of abstraction between the RTL and transistor level descriptions. One potential solution is to modify the RTL description to model the transistor

level description at a greater level of detail. This would "move" the RTL description towards the transistor level description. However, this leads to a couple of problems. Firstly, any change in the transistor level implementation would require a change in the RTL. Secondly, with the RTL being less abstract and more like the implementation itself, functional simulation time for the RTL would increase. Situations like the above provide enough motivation to devise a methodology that can (a) address correct generation of constraints and avoid false positive verification failures and (b) tightly couple underlying implementation changes to design constraints used in RTL and transistor level verification.

4.2 Constraints for Dynamic Circuit Implementations

In this section, we formalize the relationship between dynamic circuit implementations and related design constraints that are required to verify them.

Definition 4.2.1. A domino logic circuit implementation D comprises a non-empty set of dynamic logic stages $s_1, s_2, \ldots s_n$, where n is the number of stages.

$$D = \{s_1, s_2, \ldots, s_n\}$$

For example, in Figure 4.4, the logic stages are A, B and M.

Definition 4.2.2. For a dynamic logic stage s_i in a domino logic implementation, D, the fan-in is a function $\gamma : D \rightarrow 2^D$, returning a set of stages that have outputs feeding at least one input of s_i.

$$\gamma(s_i) = \{s_j | \textit{ at least one output of } s_j \textit{ feeds an input of } s_i\}$$

Definition 4.2.3. Let C_{rg} be a clock regeneration configuration that is a non-empty set of clock regenerators such that

$$C_{rg} = \{c_1, c_2, \ldots, c_m\}$$

where m is the total number of clock regenerators in the circuit. Every design has an associated C_{rg}.

Definition 4.2.4. Let $F_{dtc} : D \rightarrow C_{rg}$ be the domino-to-clock-regenerator function for a domino circuit D and a clock regenerator configuration C_{rg}

$$\forall s \in D, \exists c \in C_{rg} : F_{dtc}(s) = c$$

This implies that the stage s is driven by the clock regenerator c.

Definition 4.2.5. A constraint-wise best case domino logic circuit implementation D_b with respect to a clock regenerator configuration C_{rg} is one where each dynamic logic stage is driven by the same clock regenerator, say c, i.e.,

$$\forall s \in D_b, F_{dtc}(s) = c$$

The constraint-wise best case domino logic circuit configuration ensures that all the dynamic logic stages are driven by the same clock and hence are perfectly synchronized in the precharge and evaluate phases.

Corollary 4.2.1. *In a D_b circuit, all the stages, namely, $s_1, s_2, \ldots, s_n$, simultaneously precharge in one timing window (or clock phase) and simultaneously evaluate in the another timing window (or clock phase).*

Definition 4.2.6. A constraint-wise worst case domino logic circuit implementation D_w with respect to a clock regenerator configuration C_{rg} is

one where no two stages s_i and s_j, where $1 \leq i, j \leq n$ and $i \neq j$, are driven by the same clock regenerator, i.e., F_{dtc} is an injective function:

$$\forall s_i, \forall s_j \in D_w : F_{dtc}(s_i) = F_{dtc}(s_j) \Rightarrow s_i = s_j$$

The constraint-wise worst case domino logic circuit configuration ensures that no two dynamic logic stages are driven by the same clock and hence cannot guarantee any synchronization between the precharge and evaluate phases of the different dynamic logic stages.

Corollary 4.2.2. *In a D_w circuit, it is not necessary that all the stages, namely, $s_1, s_2, \ldots, s_n$, will simultaneously precharge in a timing window (or clock phase) or simultaneously evaluate in another timing window (or clock phase).*

The constraint-wise best case and worst case logic circuit configurations do not have any implications on any efficiency issues in the design. However, from the point of view of generating constraints for verification purposes, these circuit configurations provide the limits of possible environments for the dynamic logic implementation of a given RTL specification. In between these two extremes one can have many logic configurations where the dynamic logic stages and the clock regenerator circuits are mapped differently. The next corollary outlines a range of circuit implementations that can arise because of various practical, electrical and performance related issues involved in high performance custom circuit designs that may directly impact the domino-to-clock-regenerator function F_{dtc}.

Corollary 4.2.3. *There is a range of possible circuit implementations based on different domino-to-clock-regenerator functions F_{dtc} that are neither D_b nor D_w. Let such a set of implementations be $\hat{\mathcal{D}}$. Therefore, the set of possible implementations is $\mathcal{D} = \{D_b\} \cup \hat{\mathcal{D}} \cup \{D_w\}$.*

A change in the clock regenerator design in an underlying implementation of a circuit can change the circuit in terms of its position in the set $\mathcal{D}$ – thereby changing the design constraints that would be required in order to verify equivalence between the block's RTL and transistor level descriptions. The following two lemmas express the necessary and sufficient set of constraints required to verify such implementations.

Lemma 4.2.4. *For a D_b circuit implementation, it is necessary and sufficient that the set of environment constraints, C, $\forall s_i \in D_b : 1 \leq i \leq n$, capture only the evaluate conditions of the set of stages $\gamma(s_i)$ for verifying that the circuit implementation of the stage s_i satisfies its corresponding RTL specification.*

Lemma 4.2.5. *For a D_w circuit implementation, it is necessary and sufficient that the set of environment constraints, C, $\forall s_i \in D_w : 1 \leq i \leq n$, capture the precharge and evaluate conditions of the set of stages $\gamma(s_i)$ for verifying that the circuit implementation of the stage s_i satisfies its corresponding RTL specification.*

Necessity of the constraints in both the lemmas is obvious. In the best case configuration, the evaluate constraint is also sufficient because whenever any stage is precharging or evaluating, all others are also in the same phase, thus making sure that all the stages are in complete synchronization. However, in the worst case configuration having only the evaluate

constraint is not sufficient because during the time the circuit under verification is in its evaluate phase, an environmental module might be in the precharge phase. Thus using only the evaluate constraint might result in a false positive verification failure.

Theorem 4.2.6. *For any $D_e \in \mathcal{D}$ circuit implementation, it is necessary and sufficient that the set of environment constraints, C, $\forall s_i \in D_e : 1 \leq i \leq n$, capture both the precharge and evaluate conditions of the set of stages $\gamma(s_i)$ for verifying that the circuit implementation of the stage s_i satisfies its corresponding RTL specification.*

The intuition behind the theorem is that as one explores the design implementation space represented by the set of designs $\mathcal{D}$, keeping the RTL specification same, one needs different sets of constraints for proving equivalence between the RTL and transistor level descriptions of the same design. Figure 4.5 shows the relation between the different sets of constraints that become necessary during such an exercise. The constraints that are needed for the verification when the implementation is D_b is the strongest as opposed to the weakest constraint that is needed for verification when the implementation is D_w. However, when the transistor level implementation is anywhere in between the two extremes, say some $D_e \in \hat{\mathcal{D}}$, the constraints are weaker than those used for a D_b implementation and stronger than those used for a D_w implementation. The goal during any verification of equivalence between an RTL and a transistor level description is to assume a D_w implementation that ensures that the weakest possible constraints were used during the verification. This avoids false positives that could arise during other modes of operations such as test, debug, and other non-functional modes. However, it may not always

be possible to design a circuit that works under any environment it is placed in.

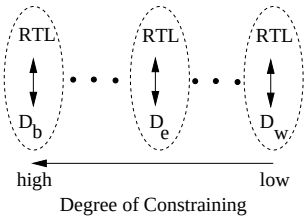


Figure 4.5: Strengthening and weakening of constraints

In an implementation verification methodology, it is important to have the flexibility of being able to incorporate design changes late in the design cycle. Having the capability of automatically generating design constraints as the environment changes and re-running equivalence checking enables a push-button methodology for implementation verification. For example, due to electrical and routing considerations, there may be a re-arrangement in the clock regeneration and balancing schemes for a high performance design. This can potentially invalidate the assumptions for dynamic circuit implementations as shown earlier. A methodology that can automatically detect and generate a new set of design constraints for equivalence checking can eliminate false positives. In the next section, we show how constraints can be automatically derived from a circuit's environment.

4.3 Automatic generation of Constraints

In a typical design verification flow, there are a number of *user-constraints* that are used in a sub-component's equivalence checking flow. These constraints are then treated as assertions that must be satisfied in the

context of the environment in which they are instantiated. These asserts are expressed as boolean conditions or invariants and must always be valid during all modes of operation of the microprocessor. A methodology to check the correctness of these invariants and to ensure the absence of over-constraining [9] is of prime importance. This is critical for finding silicon logic bugs due to incorrect/over-constrained equivalence checking at the sub-component level. One often-used, yet inefficient method of invariant checking is to write simulation monitors to monitor assertion violations every simulation cycle using directed test suites and functional test vectors. The problem with such a method lies in the difficulty in generating directed test vectors, and their inherent incompleteness from the point of view of coverage – the simulation sequence used may not "hit" the bug, which may then go undetected. Also, simulation monitors usually have an adverse effect on simulation performance.

We present an algorithm that automatically extracts a set of constraints *auto-constraints* from a design's environment. Ideally, the *user-constraints* set should be identical to or weaker than the *auto-constraints* used during the equivalence checking flow. Auto-constraints eliminate the need for the manual codification of *user-constraints*. However, in an industrial setting, the environment of the design sub-component may not be known until later in the design cycle. Also, it may be the case that there are many possible environments for the sub-component. The non-deterministic nature of the environments for a design sub-component mandates the need for a *correct-by-construction* methodology for constraints. We finally present a methodology that guarantees the absence of over-constraining in our equivalence checking flows by considering both the en-

vironment in which the design component is embedded and the design component itself. This guarantees the elimination of *false positives* due to incorrect constraints assumed during the equivalence checking flow.

Definition 4.3.1. Let m^d denote a set of storage elements of dimension d. A d-dimensional organization of a set of storage elements m, where $d \in \mathcal{N}$, is a set of factors $F_{m^d} = \{f_1 \dots f_d\}$ such that $|m^d| = f_1 \times f_2 \times \dots f_d$. Let S^{m^d} be the state space associated with the d-dimensional set of storage elements m. Then $S^{m^d} = 2^{|m^d|}$.

Definition 4.3.2. A design $\mathcal{D}$ is a 5-tuple (I, O, L, A, C), where $I(O)$ is the input (output) space, L is the set of 1-dimensional storage elements (latches), A is the set of q-dimensional storage elements (memories), where $q > 1$, and C is the set of elements in the design other than L and A. If t is the number of latches, then $L = (l_1 \dots l_t)$. If u is the number of memories, then $A = (m_1^{d_1} \dots m_u^{d_u})$, where $d_i \in N$ is the d_i-dimensional organization of memory m_i. If v is the number of nodes other than inputs, outputs, latches and memories, then $C = (c_1 \dots c_v)$.

Corollary 4.3.1. *Let S^A be the memory state-space associated with the set of memories A.*

$$S^A = \prod_{i=1}^{u} S^{m_i^{d_i}}$$

Let S^D be the state-space associated with design D. Then S in Definition 2.1.2 is

$$S = S^D = S^L \times S^A$$

Note that L (in Definition 4.3.2) is a 1-dimensional organization of a set of storage elements and can also be expressed as a 1-dimensional memory m^1

Definition 4.3.3. Let K_i denote the representative dimension for memory $m_i^{d_i}$. K_i is obtained by selecting a factor $f_j \in F_{m_i{}^{d_i}}$, where $1 \leq j \leq f_{d_i}$. Let G_{m_i} denote the memory state-abstraction function $G_{m_i} : S^{m_i^{d_i}} \rightarrow S^{m_{K_i}^1}$ associated with memory $m_i^{d_i}$, where

$$m_{K_i}^1 = \big(l_{(t+(i-1)(K_{i-1})+1)},\ l_{(t+(i-1)(K_{i-1})+2)},\ \ldots,\ l_{(t+(i-1)(K_{i-1})+K_i)} \big)$$

Let Q be the complete set of the 1-dimensional organization of the abstracted memory state nodes.

$$Q = \bigcup_{i=1}^{u} m_{K_i}^1$$

Q can also be viewed as the set of data output nodes driven by the memory and are treated just as latches. Let H_D denote the design's state-abstraction function where H_D is a vector of the memory state-abstraction functions G_{m_i}.

$$H_D = \{G_{m_1} G_{m_2} \ldots G_{m_u}\}$$

Definition 4.3.4. Let S_c^A be the abstracted memory state-space associated with the set of memories A.

$$\begin{aligned} S_c^A &= H_D(S^A) \\ &= \prod_{i=1}^{u} G_{m_i}(S^{m_i^{d_i}}) \\ &= \prod_{i=1}^{u} S^{m_{K_i}^1} \end{aligned}$$

Let S_c^D be the complete abstracted state-space associated with design D.

$$S_c = S_c^D = S^L \times S_c^A$$

Let n_c be the total number of state variables in this new state space.

$$n_c = \sum_{i=1}^{u} K_i \; + t \qquad \text{Definitions 4.3.2, 4.3.3}$$

$$S_c = S_c^D = S^L \times S_c^A = 2^{n_c}$$

Definition 4.3.5. A design's invariant-evaluation state machine M^C is a 6-tuple $(I, O, S_c, \delta^c, Z, S_c^{init})$, where $I(O)$ is the input (output) space, S_c is the abstracted state space in Definition 4.3.4, δ^c is the invariant-evaluation function, $Z = L \cup Q \cup C$ is the complete set of latch nodes, abstracted memory nodes, and all other nodes not classified as latch, memory state nodes, or primary outputs (Definitions 4.3.2 and 4.3.3) and S_c^{init} is the set of abstracted initial states derived from S_0 using the memory state abstraction function G_{m_i}. (Definition 2.1.2).

If $z = k + n_c + v = |O \cup Z|$, then

$$\delta^c = \mathcal{B}^m \times \mathcal{B}^z \rightarrow \mathcal{B}^z$$

Let u denote input state variables for nodes in I, s denote current state variables for nodes in $L \cup Q$, and y denote constraint variables for nodes in $O \cup Z$. Whenever we do not distinguish between input and current state variables, we use x to denote the union of u and s. The constraint relation between nodes in $O \cup Z$ is implicit in the δ^c function and is captured by the characteristic function

$$\chi(u, s, y) : \; \mathcal{B}^m \times \mathcal{B}^{n_c} \times \mathcal{B}^{k+v} \rightarrow \mathcal{B}$$

The characteristic function $\chi(u, s, y)$ returns 1 iff the constraint relation y is an the image of the current state $s \in S_c$ and of the input $u \in I$ according to δ^c.

In the case of deriving invariant-relations between nodes in the set $O \cup C$, the δ^c simplifies to

$$\delta^c = \mathcal{B}^m \times \mathcal{B}^{n_c} \rightarrow \mathcal{B}^{k+v}$$

Definition 4.3.6. Let M^D be an invariant-evaluation state machine associated with a design D. The invariant-evaluation relation R associated with M^D is

$$R_D(u, s, y) = \prod_{i=1}^{v} (y_i \equiv \delta_i^c(u, s)) = \prod_{i=1}^{v} r_i(u, s, y)$$

When we don't distinguish between input and current state variables, the invariant-evaluation relation is expressed as

$$R_D(x, y) = \prod_{i=1}^{v} (y_i \equiv \delta_i^c(x)) = \prod_{i=1}^{v} r_i(x, y)$$

where $r_i(u, s, y)$ is a subset of the Cartesian product $I \times (L \cup Q) \times (O \cup C)$

Theorem 4.3.2. *Let a design-cut D_{cut} be any arbitrary subset of C in design D.*

$$D_{cut} = \{w_1, w_2, \cdots, w_n\} \subseteq C$$

If C_f denotes the characteristic function that expresses the relation between nodes w_i in D_{cut} using boolean variables y_i for w_i, then the constraints associated with the design-cut D_{cut} is expressed as $C_{cut} = C_f(y_1, y_2, \cdots, y_n)$.

$$C_{cut} = \exists_x \left(R_D(x, y) \wedge S_c^{init}(x) \right)$$

$$C_{cut} = \exists_x \left(\prod_{i=1}^{n} r_i(x, y) \ \wedge S_c^{init}(x) \right)$$

Some of the above definitions are similar to the transition relation and image computation techniques used in model checking [29][41][110]. The building of the *invariant-relation* and the constraint computation algorithm utilizes the *partitioned* and/or the *clustered* techniques thereby avoiding the building of the "monolithic" relation. Both the techniques also employ *early quantification* for the image computation [31]. Although these optimizations are required for efficient implementation, it is not relevant here as what is being established is a more general theoretical result for computing invariants.

In fact, any n-ary relation between an arbitrary set of n nodes in a *design-cut* can be automatically derived or extracted employing image or range computations using independent variables for the *relevant* nodes themselves while existentially quantifying out all the remaining *irrelevant* variables from the relation. A key observation is that we compute the invariant relations between arbitrary elements in a design and therefore do not use additional boolean variables for expressing the next-states of the latches or memories.

Consider the design D shown in Figure 4.6. The circles represent design sub-components that comprise design D. We have expanded two circles depicting two design sub-components. We consider two design-cuts D^1_{cut} and D^2_{cut} in the two sub-components respectively. The design cuts $D^1_{cut} = (w_1, w_2)$ and $D^2_{cut} = (w_3, w_4, w_5, w_6)$ are then used to derive the constraint relationships. In the following sections we illustrate the steps of deriving the two invariants C^1_{cut} and C^2_{cut} associated with the design-cuts in design D using the algorithm described in Section 2.1.

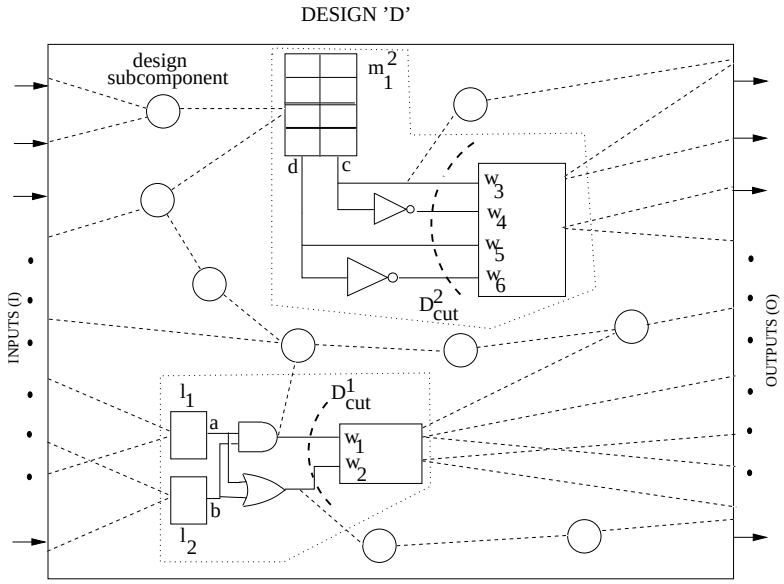


Figure 4.6: Constraints for Design-Cuts in Design D

4.3.1 An Implication Constraint

The design-cut D^1_{cut} consists of two nodes w_1 and w_2. Let y_1 and y_2 be two boolean variables that are associated with the nodes w_1 and w_2 respectively. The environment for nodes w_1 and w_2 includes all the logic in the fanin-cone of w_1 and w_2 up to the set of latches l_1 and l_2. Assume S^{init}_c to be the set of all possible starting states by assuming no initial state criteria for latches l_1 and l_2. The constraint relation C^1_{cut} between nodes

w_1 and w_2 is computed as follows.

$$\begin{aligned}
\delta_1^c &= y_1 \equiv (a \wedge b) \\
\delta_2^c &= y_2 \equiv (a \vee b) \\
C_{cut}^1 = C_f(y_1, y_2) &= \exists_{a,b}(\delta_1^c \wedge \delta_2^c) \\
&= \exists_a \exists_b (\delta_1^c \wedge \delta_2^c) \\
&= \exists_a \exists_b \left((y_1 \overline{\oplus}(a \wedge b)).(y_2 \overline{\oplus}(a \vee b))\right) \\
&= \vdots \ /^* \text{ Boolean manipulation } */ \\
&= \overline{y_1} \vee y_2 \\
&= y_1 \Rightarrow y_2
\end{aligned}$$

4.3.2 A Restricted One-hot Constraint

The design-cut D_{cut}^2 consists of four nodes w_3, w_4, w_5, w_6. These nodes are driven by a 2-dimensional memory m_1^2 having four rows and two columns. The set of factors $F_{m_1^2}$ for the memory are $\{2, 4\}$ and the representative dimension is $K_1 = 2$. The memory state-abstraction function G_{m_1} associated with m_1 reduces the state space from an original total of 256 states (m_1 has 8 storage bits) to an abstracted state space total of 4 states. Here $Q = m_{K_1}^1 = (l_3, l_4)$ (See Section 2.1). Assume that the set of initial states for memory m_1 was specified as follows.

```
Every row of the memory has at least one ‘‘1’’
```

By using the memory state-abstraction function G_{m_1}, the set of original initial states S_0 containing 81 states (each memory row has 3 possible states and there are 4 rows) now abstracts to a reduced set S_c^{init} of 3 states. The

abstracted memory nodes $l_3 l_4$ can take on any of the values 01, 10, and 11. S_c^{init} can now be represented by the following characteristic function

$$\chi_{S_c^{init}} = c \vee d$$

Let y_3, y_4, y_5, and y_6 be four boolean variables that are associated with the nodes w_3, w_4, w_5, and w_6 respectively. The 4-ary constraint relation C_{cut}^2 between nodes $w_3 \ldots w_6$ of design-cut D_{cut}^2 is computed as follows.

$$\begin{aligned}
\delta_3^c &= y_3 \equiv c \\
\delta_4^c &= y_4 \equiv \overline{c} \\
\delta_5^c &= y_5 \equiv d \\
\delta_6^c &= y_6 \equiv \overline{d} \\
C_{cut}^2 &= \exists_{c,d}(\bigwedge_{i=3}^{6} \delta_i^c \ \wedge S_c^{init}) \\
&= \exists_c \exists_d (\bigwedge_{i=3}^{6} \delta_i^c \ \wedge (c \vee d)) \\
&= \exists_c \exists_d ((y_3 \overline{\oplus} c).(y_4 \overline{\oplus} \overline{c}).(y_5 \overline{\oplus} d).(y_6 \overline{\oplus} \overline{d}).(c \vee d)) \\
&= \ \vdots \ \text{/* Boolean manipulation */} \\
&= \Big(y_3.\overline{y_4}.(y_5 \oplus y_6)\Big) \ \vee \ \Big(\overline{y_3}.y_4.y_5.\overline{y_6}\Big) \\
&= \Big((y_3 = 1, y_4 = 0) \wedge (\text{one-hot}(y_5, y_6))\Big) \vee \\
&\quad \Big(\overline{y_3}.y_4.y_5.\overline{y_6}\Big)
\end{aligned}$$

Due to the initial state restriction S_c^{init}, the computation above yields a constraint that allows fewer behaviors (lesser number of 1-minterms) than a one-hot constraint on (w_3,w_4,w_5,w_6). If all states are possible initial states, the algorithm would yield a one-hot constraint on (w_3,w_4,w_5,w_6).

4.4 Consistency Checking of User-Constraints

A *strong* constraint on a set of nodes restricts the possible relationships between the nodes to a smaller set of behaviors as compared to a *weak* constraint on the same set of nodes. Intuitively, a *weak* constraint allows more behaviors and therefore it is imperative that equivalence checking is done using the *weakest* possible constraint. The weakest possible constraint would be the tautology 1 allowing all possible behaviors, whereas the strongest possible constraint would be the contradiction 0 allowing no behaviors.

Once an *auto-constraint* has been generated, we use this *auto-constraint* to check the consistency of the user-specified *user-constraint* in equivalence checking. If the *user-constraint* is stronger (overconstraining case) than the *auto-constraint*, we generate a *modified user-constraint* and re-run equivalence checking with the new *modified user-constraint*. We use the following algorithm to generate the modified user-constraint.

Definition 4.4.1. Let c_{weaken} be the constraint function or the weakening-factor that is used to modify the user-constraint. Given a user-constraint c_u for a design-cut and an auto-constraint c_a automatically extracted for the design-cut from its environment, we generate c_{weaken} as

$$c_{weaken} \quad = \quad \sim (c_a \Rightarrow c_u) \tag{4.2}$$

The modified user-constraint c_u^{weak} is

$$c_u^{weak} = c_u \vee c_{weaken}$$

The equivalence checking between the RTL and the transistor circuit is now re-run using c_u^{weak}. However, it is also sufficient to re-run the

equivalence checking by just using c_{weaken} by itself. This may be much more efficient compared to using c_u^{weak} since c_{weaken} would typically be a much smaller and simpler function (assuming the verification engineer did a good job of identifying c_u initially). The equivalence checking process would be quicker since the equivalence checker now has to prove equivalence for only a very small set of behaviors allowed by c_{weaken}.

Theorem 4.4.1. *The weakening-factor c_{weaken} for the first verification cycle and c_a for subsequent verification cycles are necessary and sufficient constraints that must be used to re-run equivalence checking between the RTL and transistor circuit description.*

By defining appropriate design-cuts at the interfaces of design sub-components in design D, a set C_a of auto-constraints is generated. A simple heuristic to obtain design-cuts is to use the natural design partitioning of the chip into modules/blocks that are done as part of the design flow activity. Let C_u be the corresponding set of user-constraints used during the equivalence checking flow for the same design sub-components. The above algorithm can be applied to each constraint in the set C_u and a new set of *weakening-factors* C_{weaken} and modified weakened user-constraints C_u^{weak} can be obtained. Note that if the user has correctly identified the user-constraint or has even specified one that is weaker than the auto-constraint, then the algorithm ensures that no modification of c_u is done for equivalence checking.

Let us now apply the algorithm to the dynamic circuit equivalence checking problem in Section 4.1. We define the design-cut D_{cut}^{dyn} to consist

of the nodes a, a_b, b, b_b. We obtain

$$\begin{aligned}
\delta^c_a &= y_a \equiv (clkA \wedge in1) \\
\delta^c_{a_b} &= y_{a_b} \equiv (clkA \wedge \overline{in1}) \\
\delta^c_b &= y_b \equiv (clkB \wedge in2) \\
\delta^c_{b_b} &= y_{b_b} \equiv (clkB \wedge \overline{in2}) \\
C^{dyn1}_{cut} = c_a &= \exists_{a,a_b,b,b_b}(\delta^c_a \wedge \delta^c_{a_b} \wedge \delta^c_b \wedge \delta^c_{b_b}) \\
&= \vdots \text{ /* Boolean manipulation */} \\
&= (a \oplus a_b).(b \oplus b_b) \vee (\bar{a}\,\overline{a_b}\,\bar{b}\,\overline{b_b})
\end{aligned}$$

With auto-constraint c_a and the user constraint c_u in Equation 4.1, we derive from Equation 4.2

$$c_{weaken} = (clkM \ \vee \ (a \oplus a\_b).(b \oplus b\_b)) \ \wedge \ (\overline{clkM} \ \vee \ (\bar{a}\,\overline{a\_b}\,\bar{b}\,\overline{b\_b}))$$

It is now sufficient to re-run equivalence checking for the dynamic circuit in Figure 4.4 with the constraint c_{weaken}. This technique was applied to a number of custom designed dynamic circuits from the Motorola MPC7455 microprocessor. where both c_{weaken}, c_a and c_u was used for verifying equivalence between the RTL and transistor-level descriptions. The results are shown in Table 6.6 and discussed in Section 6.3 of Chapter 6. The results clearly illustrate that using c_{weaken} over c_a is a much improved equivalence checking methodology.

Having automatically generated the correct set of constraints and proved equivalence between the RTL and the circuit implementation using STE, the question that is addressed in the next chapter is whether it is still required to functionally simulate a gate/switch-level model of the circuit implementation.

Chapter 5

Towards The Complete Elimination of Gate/Switch Level Simulations

A great deal of effort goes into simulating Verilog gate/switch-level models of microprocessors to root out bugs before tapeout. This is done even if all the RTL modules that comprise the microprocessor have been verified to be equivalent against the gate-level or switch-level models of their corresponding transistor circuit using the methodology described earlier. There are two main reasons for doing this. Firstly, most formal equivalence checking tools [107][117] establish logical consistency and not semantic consistency [48] between two different design representations. This leads to the possibility of two logically consistent models (the RTL and the synthesized or extracted gate/switch-level model) exhibiting semantically inconsistent behavior during simulation due to the different RTL coding styles possible. Secondly, the semantics of the ternary value X (unknown/don't-care/indeterminate) and the synthesis pragmas such as full-case/parallel-case lead to inconsistent interpretation by the simulation, synthesis, and formal tools. The simulation mismatches that occur due to the RTL coding issues and the inconsistent X interpretation are known as the pre- vs post synthesis simulation mismatches [8][42][56]. This dissertation deals with the second scenario of inconsistent X interpretation.

Eliminating mismatches due to the RTL coding styles is possible

by a combination of restrictive RTL coding style, RTL linting, and the use of the symbolic simulation methodology in this dissertation. Since simulation is necessary to expose the semantic inconsistency [48] between the RTL and gate/switch-level models, equivalence checking using the verification methodology in this dissertation can establish both logic and semantic consistency simultaneously. This is because symbolic simulation combines traditional simulation with formal symbolic manipulation to prove equivalence.

It can be argued that if ternary equivalence (X-equivalent in addition to binary equivalent) were established between the RTL and the gate/switch-level models, only then could RTL models be substituted for the Verilog gate/switch-level models used in chip-level simulation. In the next few sections, we show that if the verification methodology in Chapter 2 is applied to every block that comprises the chip, then RTL models are sufficient for chip-level simulations. This obviates the need for gate/switch-level models in chip-level simulations or for power-on-reset simulations [74]. This dissertation contends that metastable state prevention must be part of the circuit design methodology. We also show that constraint violations are not necessarily detected using gate/switch-level model simulations and that the detection capability can vary across different simulators. Constraint validation must be an integral part of the equivalence checking flow.

5.1 Metastable and Unknown states

We define a metastable state to be a state of the circuit where at least one or more nodes in the state vector are at an intermediate volt-

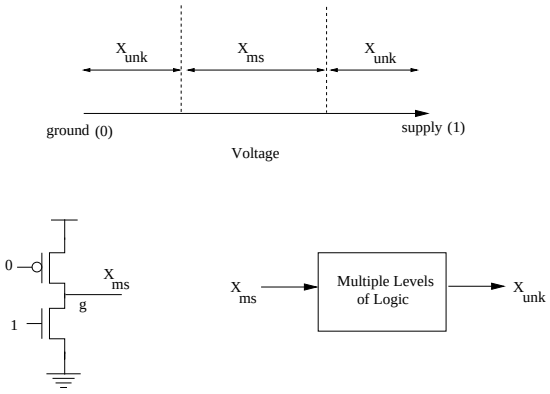


Figure 5.1: Boolean and Ternary Simulation

age. The value of the node at this intermediate voltage is denoted by the unknown value type X_{ms}. These states may arise in hardware due to incorrect timing assumptions, incorrect transistor sizing, transistor leakage issues due to deep sub-micron process technologies, or due to contention within circuit structures. For example, under the right conditions of setup and hold timing violations, a latch may enter a metastable state [119]. The latch is perfectly balanced between making a decision to resolve to a 0 or a 1.

If the voltage is unknown but is guaranteed to be either close to the supply or ground voltages, we denote it by the value type X_{unk}. Note that X_{unk} obeys the *law of the excluded middle* $X_{unk} + \overline{X_{unk}} = 1$ whereas X_{ms} does not. The ternary simulation algorithm does not distinguish between X_{ms} and X_{unk}. To the best of our knowledge, ternary simulators do not distinguish between the two. Figure 5.1 shows the relation between X_{ms} and X_{unk}. In this chapter, if an X is mentioned then it could refer to an either X_{ms} or X_{unk}. In cases where they need to be differentiated, it is explicitly specified.

Consider the circuit shown in Figure 5.1. Let a metastable state X_{ms} arise at the node g. If g fans out to one or more levels of logic, the X_{ms} will resolve to a X_{unk} with a very high probability. This is because the logic driven by g has to be so perfectly balanced and tuned to propagate a voltage that is intermediate to the ground and supply voltages. In practice, noise (switching and/or thermal) or a slight imbalance eventually pushes the X_{ms} to resolve to a X_{unk}. Therefore, the same ternary simulation algorithm for X_{unk} can be used to evaluate the effect of a X_{ms}. In full chip switch-level simulations, X_{ms} and X_{unk} can be treated to be equivalent for the final simulation results.

5.2 Switch-level Model Accuracy

Switch-level models are an abstraction of metal-oxide semiconductor (MOS) transistor circuits where transistor conductances and node capacitances are modeled by discrete strength and size values. Node voltages are represented by discrete states 0, 1, and X. The assignment of integer values for transistor strengths based on transistor conductances can be described by a function $\rho : \Re \rightarrow \mathcal{N}$ where $\Re$ is the set of real numbers and $\mathcal{N}$ is the set of natural numbers.

The equivalence checking methodology described earlier in this dissertation allows for the extraction of two categories of switch-level models from the circuit implementation. One of the switch-level models is derived by interpreting the circuit implementation in terms of its verilog-strength semantics [116]. The other switch-level model is derived by assigning discrete strengths to transistors based on resistances computed from their lengths and widths. This assignment is based on a user-defined resis-

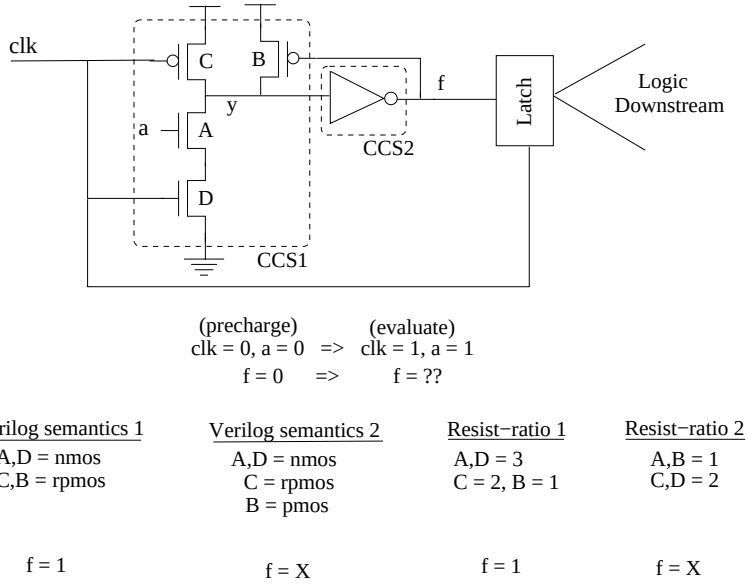


Figure 5.2: Dynamic Buffer

tance ratio parameter [21]. By varying the user-defined resistance ratio and thereby the ρ, different switch-level models can be obtained from the same circuit implementation, thereby leading to different simulation results.

Consider a dynamic circuit shown in Figure 5.2. It has two inputs *clk* and *a* and an output f that is latched. The circuit is comprised of two channel-connected subcomponents CCS1 and CCS2. The circuit precharges when $clk = 0$ and evaluates when $clk = 1$. As seen in the bottom half of the figure, four different switch-level models lead to different simulation results during the evaluation phase. Consider the simulation result obtained when using the *Resist-ratio 1* switch model. In this case, the output f goes to a 1 because the half-keeper transistor B has a strength of 1 and is weaker than the minimum strength 3 of the path from y to *gnd*

through transistors A and D. If our strength assignment algorithm had assigned a value of 2 to transistor B, the output f would still be a 1. In general, if we assume a mapping ρ for resistances of transistors A, B, C, D to strengths i_A, i_B, i_C, i_D respectively, and $i_A \leq i_D$ (without any loss of generality), then output $f = 1$ if $i_B < i_A$ and $f = X$ if $i_B = i_A$ and $f = 0$ if $i_B > i_A$.

For the purpose of illustration, assume a switch-level model Sw_1 that is based on a mapping ρ such that $i_A - i_B = 1$. The simulation of this switch-model will predict that the output f evaluates to a 1. Assume another switch-level model Sw_2 such that $i_A = i_B$. In Sw_2, the output f evaluates to an X. If Sw_1 was used in equivalence checking the circuit would have passed the equivalence check, but if Sw_2 was used, the equivalence check would fail. In SPICE simulations, it is possible that node y is at an intermediate voltage between logic 0 and 1 during evaluation because the half-keeper was sized slightly bigger than what was required, even though the strength assignment algorithm assigned the half-keeper transistor B a lower strength value. This could result in the latch downstream getting into a metastable state. If another half-keeper p transistor whose gate is connected to f is added in parallel to B with the same resistance, the switch-level model generated would be exactly the same as with the one with exactly one half-keeper. This is because no series-parallel linear network analysis is done to compute the voltages but rather the computation is based on path analysis over a totally ordered strength set.

The above example illustrates that switch-level models are a good abstraction for logic verification tasks but cannot be relied upon to detect and propagate metastable conditions or other non-digital circuit behavior.

The problem is further exacerbated because of deep sub-micron process technologies where transistor leakage and charge sharing are now dominating the transistor behavior characteristics. A theoretical analysis of the mapping ρ was done by Cerny et al. [33] where it was established that equivalence/magnitude-class calculations in switch-level modeling are too inaccurate for faithfully capturing detailed and/or marginal/non-digital circuit behavior. The equivalence class separation and the accuracy of the switch-model computations are dependent on the number of transistors in a channel-connected subcomponent and the closeness of their resistances and/or related parametric values. The accuracy problem associated with these switch-level models often show up in ratioed circuits such as footed and non-footed dynamic logic. Memory bitcells and their associated circuitry as shown in Figure 2.25 in Chapter 2 are also types of ratioed circuits where the write operation is dependent on a carefully designed ratioed logic between the write drivers, the bitcell transistors, and the wordline and isolate pass transistors.

5.3 Boolean vs. Ternary equivalence

Ternary discrepancy vectors are defined for circuits that are boolean equivalent. A ternary discrepancy vector between two multi-output boolean equivalent models (for example, an RTL and a switch level circuit) is defined to be a ternary vector that produces at least one output discrepancy. In this section, we present the fundamental differences between boolean and ternary equivalence.

Consider an RTL specification and its corresponding switch level implementation (shown as gates here) in Figure 5.3. A formal boolean

RTL specification

```
module AndOr (a, b, c, y);
   assign y = a | ( b & c );
endmodule
```

Schematic implementation

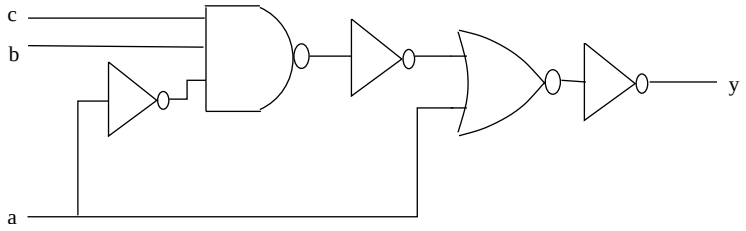


Figure 5.3: RTL and schematic

equivalence checker[117] proves that the gate-level model extracted from the circuit implementation is boolean equivalent to the RTL specification. However, the RTL is not ternary equivalent to the implementation. The two representations differ in their outputs for the ternary discrepancy vector $\{a = X, b = 1, c = 1\}$.

Let the circuit implementation be instantiated in a chip-level environment as shown in Figure 5.4. The inputs to the circuit are driven by latches gated by a c1 clock. The latches are connected together in a scan chain. The output y is captured in another latch driven by a c2 clock that is part of another scan chain. During full chip simulation, if the ternary discrepancy vector $\{A = X, B = 1, C = 1\}$ occurs in the latches A, B and C either during functional or test pattern simulation, the output of the circuit captured in latch Y is an X. The latch Y captures the value

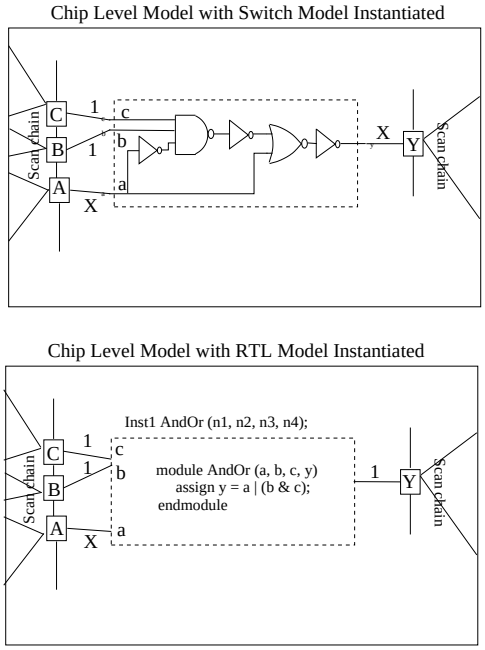


Figure 5.4: Circuit instantiation in a chip-level model

of X. However, if the boolean equivalent RTL model is substituted for the switch-level model, a 1 is captured in the output latch Y. Since the RTL model is not ternary equivalent to the switch-level model with respect to this ternary vector, does it mean that the RTL model substitution would cause a bug to be missed in simulation. Would it then mean that if the hardware were actually implemented in silicon, a logic bug would result which would otherwise have been caught if the substitution were not made?

If X in the latch A is actually a X_{unk}, having the RTL model instantiated in the chip-level environment would not invalidate the simulation pattern. Since X_{unk} encodes either a 0 or 1 but it is not known which one, having proved boolean equivalence between the RTL and the switch-level

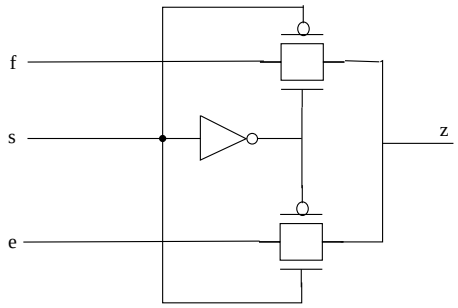
model allows us to ignore the propagation of the X_{unk}. This is because X_{unk} has the same semantics as a boolean variable on latch A used during the boolean equivalence check. No bug in silicon would occur as a result of substituting the RTL model. If the simulated pattern $a = X, b = 1, c = 1$ was an ATPG test, this test could actually be applied to the chip on the tester.

On the other hand, if X in latch A is a X_{ms}, then substituting the RTL model would mask the propagation of the metastable state present in the latch feeding input a. If it is guaranteed that no metastable states are possible in the latches during power-on-reset or normal functional operation, then substituting the RTL model for the switch-level model for chip-level simulations would not mask any real bug in silicon.

Consider another RTL model and its corresponding switch level implementation as shown in Figure 5.5. The ternary discrepancy vector is $\{e = 0, f = 0, s = X\}$. Note that here again, if $s = X$ is a X_{unk}, it does not matter whether the RTL or switch-level model is instantiated. However, if $s = X$ is the metastable value X_{ms}, then the switch-level model would propagate this value enabling the detection of the metastable state. Therefore, it appears that switch-level models are required for the detection of metastable states that can arise during chip-level simulation. However, metastable state detection is totally model and simulator dependent.

Consider an RTL and schematic implementation as shown in Figure 5.6. Now, the ternary discrepancy vector is $\{a = X, b = 1, c = 1\}$. Assume that $a = X$ is being driven by the dynamic buffer output f in Figure 5.2 and the value is of type X_{ms} . The switch level implementation masks the propagation of X_{ms}. However, substituting the RTL model will

```
module Mux (e, f, s);
   assign z = (e & s) | (f & ~s);
endmodule
```



Ternary discrepancy vector : s = X, e = 0, f = 0

RTL produces a 0

Schematic produces an X

Figure 5.5: RTL and schematic for a MUX

enable the propagation and detection of the X_{ms}, assuming a specific implementation of the RTL compiler. In this case, the reverse scenario occurs where substituting the RTL model enables the propagation and detection of the metastable state. It is not a sufficient condition that switch-level models will enable detection of metastable states. The propagation, detection and resolving of X's is wholly dependent on the RTL coding style and the simulation semantics of the particular simulator in use. Therefore, different simulators on the same models could propagate X_{ms} in different ways. It is not sound methodology to rely on switch-level full chip simulation to detect these metastable states.

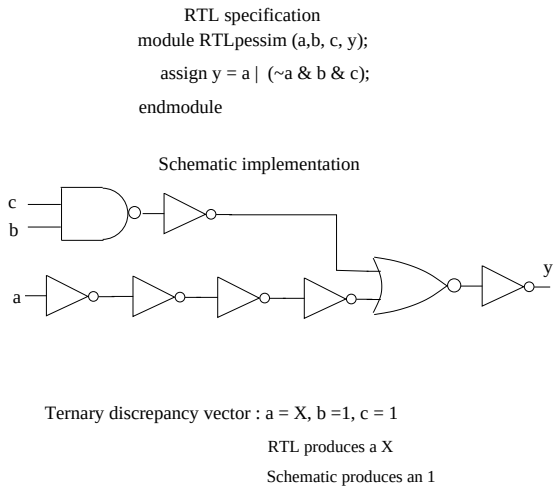


Figure 5.6: RTL and schematic depicting RTL pessimism

5.4 Chip simulations

Consider an enhanced equivalence checking flow as shown in Figure 5.7. Assume that the methodology shown in Figure 5.7 is applied to each of the blocks in Figure 5.8. Consider the chip-level switch-level model shown in Figure 5.9. It is assumed that all the circuits A, B, C, D, E, F and G have been formally verified to be boolean equivalent to their respective RTL specification models. We also assume that all constraints used during the boolean equivalence verification between RTL and switch-level models have been verified using some sort of assume-guarantee reasoning.

5.4.1 RTL model substitution

Consider circuits C and D. Their corresponding RTL models are shown in Figures 5.3 and 5.5 respectively. Substitute the RTL model for circuit D in the chip-level switch model. The ternary discrepancy vectors that would cause latch L_y to have different outputs with the substituted

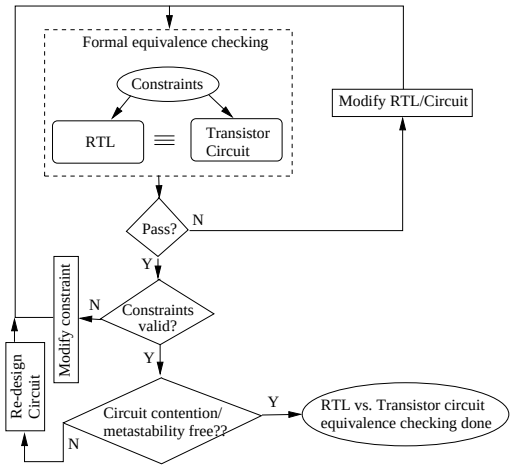


Figure 5.7: Enhanced RTL vs. switch level circuit equivalence checking flow

RTL model is the set of vectors shown in Table 5.1. With the RTL model substituted for circuit D in full-chip simulations, the latch L_y will capture a value of 1 for all the fifteen ternary discrepancy vectors in Table 5.1. If the switch-level model for circuit D was used in full-chip simulations, the latch L_y would capture an X instead. However, if all the X's in these vectors are of the value type X_{unk}, then no real bug is masked by substituting the RTL model as elucidated in Section 5.3.

If the circuit is guaranteed to be free of metastable states, through circuit design methodology for example, then all the circuit blocks in Figure 5.9 can be replaced with their respective RTL models for chip-level simulation. See Figure 5.10 for the substituted RTL full chip model. There will be no logic bug that can go undetected with the substitution of these RTL models for chip-level simulation.

Consider the effect of the RTL model substitution on the simulation of a DFT test pattern. Assume that a test is to be generated for a *s-a-0*

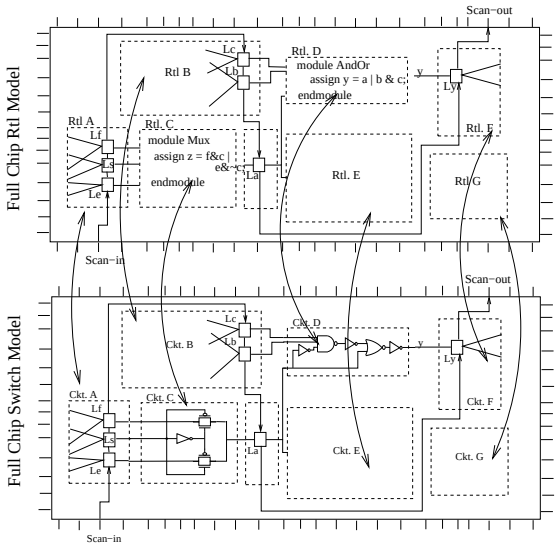


Figure 5.8: Full chip equivalence checking

fault at site y. Let the test $L_aL_bL_c \in \{X11\}$ be generated by an ATPG tool from the full-chip model with all RTL models substituted for the switch-level models. With the substituted RTL full-chip model, the generated test will detect the *s-a-0* fault at fault site y on the actual chip hardware tester. This is because of the assumption that the X in the test vector $L_a = X$, $L_b = 1$, $L_c = 1$ will be treated by the tester to be a value of the type X_{unk}. However, this test vector would have been discarded if the test was simulated on the switch-level model for circuit D, because it would have yielded an X at the output latch L_y.

As part of the testing methodology, it would be prudent to make sure that no test patterns are generated that yield X's in the switch-level model and definite values of 0 and 1 in the RTL substituted model. This can be guaranteed if the ATPG tool generates tests that are purely binary. In cases where the input patterns have $X'es$ in them, they can safely be

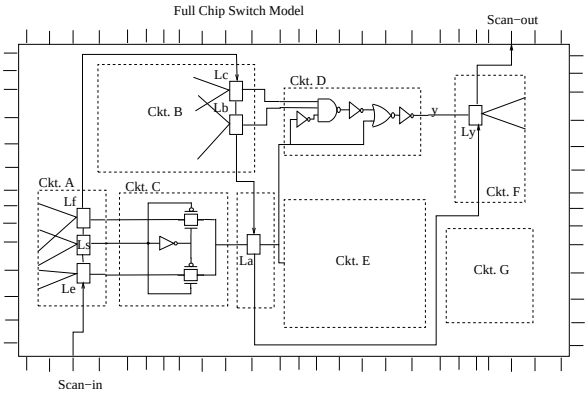


Figure 5.9: Full Chip Switch-level model

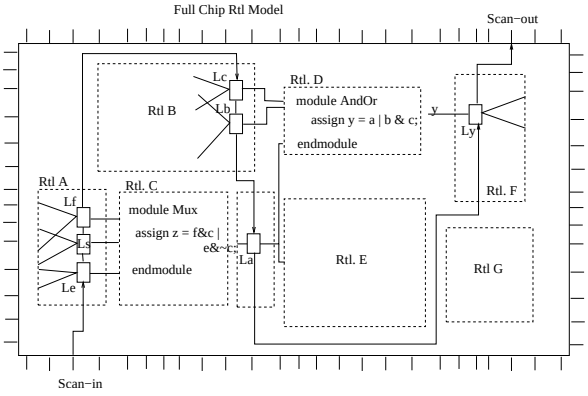


Figure 5.10: RTL chip-level model

replaced with binary values of 0 or 1. If this is the case, then RTL models can be substituted for the switch-level models. The only other possibility is the generation of X_{ms} from within a circuit block with all inputs to the block being binary valued. However, based on our initial assumption about the absence of metastable states, this is not possible.

It seems that it may not be useful to go debug $X'es$ in full-chip switch-level simulations, if it can be guaranteed that no part of the circuit can go into a metastable state (as part of a separate methodology).

Table 5.1: Ternary Discrepancy Vectors for Latch L_y

$L_e L_f L_s L_b L_c$
00X11
01X11
0X011
0XX11
10X11
11X11
1X011
1XX11
X0111
X0X11
X1111
X1X11
XX011
XX111
XXX11

5.4.2 Violation of constraint assumptions

If violation of constraints leads to generation of X_{ms}, then substituted RTL models may mask the propagation of X_{ms}. However, this is again dependent on how the RTL models are coded, how the circuit is implemented and the assumptions made by the circuit designer.

Consider the RTL specification of a 2-input mux shown in Figure 5.11. Assume that the RTL is given by a logic designer to a circuit designer. In addition, the circuit designer gets additional information from a second logic designer designing the constraint logic that the selects are exactly one-hot. The circuit designer then implements the circuit shown in Figure 5.12. The RTL and the circuit are equivalent under the constraint that the selects are one-hot.

RTL specification for a tri−state mux

```
module tsmux (c, d, a, b, y);
    bufif1 (y, a, c);
    bufif1 (y, b, d);
endmodule
```

Figure 5.11: RTL specification for a tri-state mux

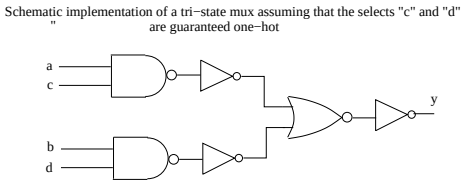


Figure 5.12: Schematic implementation for a tri-state mux assuming selects are one-hot

Assume that the second logic designer makes a mistake while coding up the RTL and codes the constraint logic as shown in Figure 5.13. The complete RTL model is shown in Figure 5.14. Now, a second circuit designer implements the circuit schematic in Figure 5.15 for the constraint logic specification given by the second logic designer. The completed circuit when integrated in the chip environment is shown in Figure 5.16. Now, ver-

```
module onehot_constraint (in, sel1, sel2);
    assign sel1 = in;
    assign sel2 = in;
endmodule
```

Figure 5.13: Incorrect coding of RTL module for 1-hot constraint

Complete RTL specification

```
module tsmux (c, d, a, b, y);
    bufif1 (y, a, c);
    bufif1 (y, b, d);
endmodule

module onehot_constraint (in, sel1, sel2);
    assign sel1 = in;
    assign sel2 = in;
endmodule

module completrtl (in, a, b, y);
    wire c, d;
    I1 tsmux (c, d, a, b, y);
    I2 constraint (in, c, d);
endmodule
```

Figure 5.14: Complete RTL model

Circuit Implementation of RTL constraint module

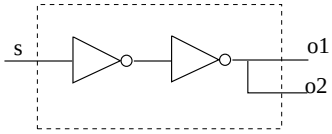


Figure 5.15: Constraint Circuit Implementation

ification engineers formally prove that circuits *I1ckt* and *I2ckt* are boolean equivalent to the RTL modules *onehot_constraint* and *tsmux* respectively. Note that RTL module *tsmux* in Figure 5.11 was proved equivalent to its *and-or* switch-level implementation in Figure 5.12 under the constraint that the selects are exactly one-hot. During full-chip simulation, with the switch-level models in place, the contention case where A is 0 and B is 1 and input select S is a 1 cannot be detected. However, substituting the RTL model *tsmux* for circuit *I2ckt* enables the detection of the metastable

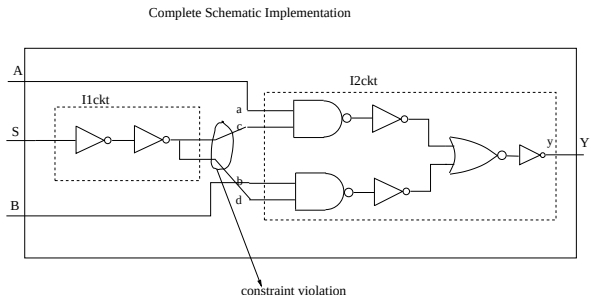


Figure 5.16: Complete Circuit Implementation

state X_{ms}. In Table 5.2, the first three row entries, which correspond

Table 5.2: RTL simulation results

S	A	B	Y
0	-	-	X
1	1	0	X
1	0	1	X
1	0	0	0
1	1	1	1

to six vectors, enable the detection of the constraint violation during full-chip RTL simulations. In Table 5.3, the last three row entries enable the detection of the constraint violation in a full-chip switch-level model simulation. The probability of detection (3/4) of the constraint violation using the full-chip RTL model is twice that of the detection probability (3/8) using a full-chip switch-level model. If all tri-state circuits are modeled using the correct Verilog primitives such as bufif's or notif's in the RTL, then substituting RTL models would be equivalent to having switch-level representations of the circuits in full-chip simulation. The above example illustrates that tackling constraint violations utilizing switch-level models is neither a necessary nor a sufficient condition and that other techniques

Table 5.3: Switch-level simulation results

S	A	B	Y
0	0	0	0
0	0	1	0
1	0	1	1
1	1	1	1
1	0	0	0
1	1	0	1
0	1	0	0
0	1	1	0

or methodologies need to tackle the problem. Moreover, by utilizing the RTL models for chip-level simulations we speed up full-chip simulations tremendously.

Detection of constraint violations and the detection of contention/metatstable states must be done independently of full-chip switch-level simulation. This must not be a primary objective of chip-level simulation flows. Depending on how the RTL is coded and the structure of the switch level implementation and its corresponding switch model, different simulation results are possible with either of the models. We have shown that switch-level models are not guaranteed to detect constraint violations or metastable states. By enhancing the equivalence checking methodology with a constraints-guarantee mechanism and a contention/metastable prevention circuit design methodology, we can establish that no full-chip switch-level simulations are required.

By enhancing the equivalence checking flow to deal with different switch-level models, it is now possible for designers to use the modified equivalence checking flow results to assist in generating directed SPICE

simulations. For example, each circuit may be verified to be equivalent to the RTL for both the verilog-strength model or the resistance ratio model. If the verilog-strength switch model passes verification and the resistance ratio model does not, the counter-example may be used for generating a directed SPICE simulation. However, due to the switch-level model accuracy problem discussed earlier in Section 5.2, the designer may not choose to. The verification run-times and memory usage for both the resistance ratio model and the verilog-strength model were generated for all custom memories as shown in Table 6.4 in Section 6.3 of the next chapter. From the table, it is clear that the verilog-strength models take less time and memory for simulation as compared to the resistance ratio switch model.

X-state semantics at different modeling levels of abstraction must be tackled at the circuit vs. RTL verification level and tools need to be in place to enable designers to do a more rigorous analysis of their circuits. Utilizing RTL models will speed up full-chip simulation tremendously and will enable faster time-to-market, provided all circuits that make up the chip model have been formally verified to be boolean equivalent to their switch-level models. In the next chapter, we present the design, development, and deployment aspects of an industrial strength formal verification tool that incorporates some of the techniques we have described so far.

Chapter 6

A Practical Equivalence Checking Tool

Design flows use a myriad of CAD tools and software products to validate and analyze their designs. Many of these tools employ formal techniques [14][25][36][78] [96] discussed in Section 1.2 in Chapter 1. These software tools are either proprietary or are supplied by specialty CAD tool vendors. In some cases, the front-end of the tool is designed in-house and the back-end is some proprietary interface that is supplied by a vendor. In the past few years, there have been a considerable number of vendors specializing in formal analysis and formal verification tools. These tools have been successfully used in mainly ASIC flows and work on small designs. More recently, gate-level logical equivalence checking tools have matured to the point that they are now being used by designers. However, property checking and/or symbolic simulation tools are typically used by only a few experts on the project or by specialist support personnel. They hold advanced degrees in the tools subject matter and are specifically hired for the purpose. It is very rare that the logic or circuit designers use the tools themselves. There are numerous reasons that are cited for the lack of use of these tools by the design teams, even though it is well understood that design quality improves with their use.

In this chapter, we present a few fundamental design decisions behind the successful deployment of a a second-generation formal custom

memory equivalence checking tool Versys2 at Motorola. Versys2 is a symbolic simulator for validating RTL vs. transistor implementations [71]. Versys2 was jointly developed by a team in the High Performance Tools and Methodology group at the PowerPC design center in Motorola [1]. The tool is currently being used by circuit and logic designers on all current PowerPC microprocessor projects. We present some of the tool design changes we had to incorporate along the way and also how the target audience for this tool influenced many of our design decisions. We try to bring out the importance of achieving a threshold design quality level without the expense of additional time-to-market. We stress the importance of the design of the user-interface and its simplicity which can make or break a project's decision to use a tool. Finally, it is our goal to show that for any successful CAD tool design flow, both the technical and economic concerns must be addressed.

6.1 Classical design and use of formal tools

Formal tools have tended to be either too labor-intensive or have been completely impractical for industrial-size problems. This in turn has crippled their abilities to challenge current design verification practices, unsatisfactory as they may be.

The design of a formal CAD tool goes through a series of phases. Typically, a formal verification expert team designs the tool based on their view of the verification problem. The requirements are usually obtained either through a document or through informal meetings between the project teams and the tool development teams. The development team then writes software focusing on the core algorithms and exploiting clever mathemat-

ical algorithms. A prototype is built and tried out on a few small designs by the tool development team. The tool is debugged on these examples and then released to be used as a software application. The users of the tool are the tool developers and possibly a few other members of the CAD tools team. For the tool to be used in the project, some of the developers have to join the project team. More often than not, the tool does not make it into the tape-out criteria of the design teams. Project managers need to see a clear and compelling advantage to adopting formal tools, beyond just enhanced coverage and incremental efficiency. This is primarily due to the conception by design teams that the tool can only be used by the developers or experts specifically hired for the purpose.

This software design and use methodology for a formal tool is typical of most formal verification software projects. The software blocks that are typically overlooked in formal tools are the debug and user interfaces. These modules are particularly important since the target users of these tools are the logic and circuit designers and it is their productivity that determines a major fraction of the design cycle time. What is usually overlooked by the CAD tool development team is that their method of debug and usage of the tool is quite often different from that of the designers. This is a major obstacle to the adoption of these tools by the designers. Because formal tools are part of a larger engineering process, they must operate within the context of not only a technical design, but also business strategies.

6.1.1 Our initial experience

Symbolic simulation is a simulation-based approach that combines traditional simulation with formal symbolic manipulation [25][7][104]. Prior work in symbolic simulation and its industrial application to microprocessor custom memories had established the benefits of symbolic simulation technology [49][91][93]. However, none of the previous approaches overcame the general problem of methodology development, tool deployment, and automation for a project design flow. To gain a better understanding of past custom memory verification efforts and its associated difficulties, two engineers were deputed to the project team. The goal was to solve the custom memory verification problem and develop a custom memory verification flow using symbolic simulation.

The CAD tool developers were part of the design team and reported their progress to the project and CAD tools teams. An elaborate but ad-hoc custom memory verification flow using symbolic simulation was developed and all custom memories in a PowerPC project were verified [71][103]. The old symbolic simulation verification flow is shown in Figure 6.1. A whole suite of verification support libraries and code specific to each custom memory was written in PERL and a specialized language FL. A number of bugs were detected in the equivalence checking between RTL and transistor schematics for these custom memories. Some of these bugs would have been detected close to or only after tape-out. However, the manual effort involved was extremely high [71] and the users of the tool were only the two CAD tool developers. Moreover, one of the users had to learn the specialized language FL (which was not central to the verification task) to program the assertions in it.

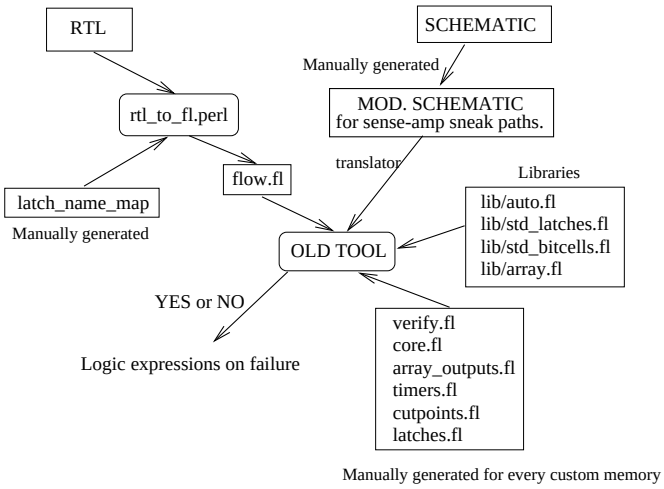


Figure 6.1: Old custom memory verification flow

Realizing the importance and the payoff in using symbolic simulation to verify custom memories, and noting that the current flow was inadequate for fitting into existing design flows, it was decided to rethink the design of the symbolic simulation tool and to automate the entire flow as much as possible.

6.2 Versys2 Design

Versys2 was jointly developed by a team in the High Performance Tools and Methodology group at the PowerPC design center in Motorola, and was completely written from scratch with the sole objective of automating the custom memory verification flow and to minimize designer effort in using the tool. The tools primary focus was to do equivalence checking between RTL and custom transistor schematics and had three main objectives

- The tool would have to have minimal user input (get as close as possible to *push-button* technology) to complete the verification task.
- The tool would have to be used by circuit and logic designers and this additional task would not exceed 10-15% of their total task time.
- Designers would not have to learn a completely new specialized programming language for carrying out the verification. This would eliminate verification coverage variability across different users of the tool.

Before the start of the project, we decided to list all the features that the tool would ideally have in addition to equivalence checking, given unlimited resources. Table 6.1 shows a matrix of a subset of the tool features and their relevance to four key metrics. It was agreed that the features would then be ranked in order of priority and the feature list would be pruned based on our customer's requirements. Features that would enable its use by circuit and logic designers would have higher priority over features that could be implemented after the first prototype was built and released. Not all features were implemented. For example, waveform and PLA debug features were implemented and took precedence over hazard or contention analysis. Features that contributed to enhancing both the functionality and usability were implemented. The RTL parser, assertion management, constraint specification and reduction features were all implemented upfront. This feature list would be pruned as and when required during the implementation stages and would be purely driven by the project's deadlines. Our first criteria was to identify the key functional blocks of the tool and to design it as separate modules. The tools architecture was to be

Table 6.1: Tools Feature Matrix

Feature List	Enhance Functionality	Enhance Usability	Enhance Maintainability	Enhance Performance
Property verification	High	Low	Low	Low
Hazard Analysis	High	Low	Low	Low
Flexible cmd-line	Low	High	Low	Low
Auto state mapping	Low	High	Low	Low
Auto temporal mapping	Low	High	Low	Low
Assertion management	High	High	Medium	Low
Var order checkpoint	Low	Low	Low	High
Internal Dynamic Node handling	Low	High	Low	Low
RTL parser	High	High	Low	High
Constraint specification	High	High	Low	Low
Checkpoint mgmt	Low	High	Low	Low
Waveform debug assistance	Low	High	Low	Low
PLA debug assistance	Low	High	Low	Low
Antecedent/Consequent display	Low	High	Low	Low
Delay file auto-gen	Low	High	Low	Low
Reductions	High	High	Low	High
Auto transistor modif.	Low	High	High	Medium
Clean BDD/Sim Interface	Low	Low	High	Medium
Contention analysis	High	Low	Low	Low
Verilog Pattern Sim	High	Low	Low	Low
Cutpoint mgmt	High	High	Low	High
Reduced false -ves due to X	High	High	Low	Low

designed to enable it to be enhanced and modified based on a plug-and-play framework.

6.2.1 Assertion Manager Interface

In the earlier tool, the STE assertions [104] were generated manually and had to be written in a specialized language. This would involve the training of the designers to write temporal logic assertions in that language. This would be an additional task to an already tight schedule for the project. The quality of the verification was directly dependent on the user's knowledge of the design and his/her expertise in generating the correct set of assertions. The number of bugs exposed would vary with the user of the tool. The resulting verification coverage would be highly dependent on the designer's knowledge of the specialized programming language.

Based on this observation, it was decided that manual assertion writing was a big hurdle to designers using the tool themselves. Since the RTL encoded all the properties that were to be satisfied by the schematic, a fully automatic assertion generator was proposed. This assertion generator would automatically parse the RTL and generate the correct set of STE assertions to be checked on the schematic. Since temporal logic assertions have a temporal component in them, the design of the automatic assertion generator would be a non-trivial task. Since the definition of these timing windows for these assertions was a considerable fraction of the assertion writing time, automating this task was a high priority and would directly impact usability. This is analogous in principle to Amdahl's law for computer design *"In making a design tradeoff, favor the frequent case over the infrequent case"*. Here the principle is applied to the design of tool defaults

and tool input for reducing user-tool interaction time of user tasks that are routinely performed.

To automate the generation of timers (the temporal component of an assertion), we setup a database of default timers for the most commonly generated assertions for custom memories. A database *project_defaults.versys* with all the default timer definitions was created for all checkpoints such as latches, primary outputs, cutpoints and bitcells [71]. Cutpoints are equivalent nodes in the RTL and schematic and bitcells are the basic storage elements that make up the memory core. This file is included in the main configuration file (See Section 6.2.4) that the tool reads. Every PowerPC project was set up with its own specific project defaults. The organization of the default timer database is explained in the following sections.

6.2.2 Phase Definitions and Phase Maps

Phase definitions are definitions that characterize the behavior of a signal in a cycle. Phase maps associate specific signals with a specific phase definition. By using a phase map for a signal in conjunction with a phase definition, we determine exactly what the phase behavior of the signal is within a cycle. Based on the phase map, a specific waveform is associated with each signal.

6.2.3 Timer Definition Defaults and Timer Rules

The timers were defined for both the antecedent stimulus and the consequent checks for all checkpoints. This involved getting the right defaults for setup and hold timing windows for latches, precharge and evaluate timing windows for dynamic inputs and outputs, bitcell antecedent and

consequent timing windows, primary output consequent timing windows, etc. The default timers were all defined with respect to an established clocking scheme. The timers were parameterized with default arguments and the users have the option of overriding the default timers with their own defined timers. A timer rule associates a default timer definition with a particular checkpoint and its phase behavior. Each checkpoint gets its own timer for both the antecedent and consequent.

A lot of effort went into getting the default timers right. The defaults enabled new users to quickly boot-strap themselves in running the tool. Default timers have worked very well in at least 90% of the designs that were verified. This dramatically cut down the verification effort by about 75%.

6.2.4 Project Defaults and Phase Map Annotations

The tool gets all its input from a single file called the configuration file. All information about the custom memory is specified in this file. This was in contrast to our earlier verification flow where all the information about the memory was distributed over different files. The configuration file is divided into different sections as shown below. The project defaults file, described in Section 6.2.1, containing all the default timers and phase definitions is included in the main configuration file using the *include* directive. The designer then specifies phase maps for the primary inputs, based on the default phase definitions available in the project defaults file. This is an important step since the waveforms applied to the inputs is based on this annotation.

On reading the phase maps of all the inputs from the configuration

Versys2 Configuration File

include "project_defaults.versys";
Phase_map definitions for primary inputs
•
•
New Timer definitions (if defaults don't suffice)
•
•
State_map definitions
    Timer Instantiations (if defaults don't suffice)
    Required Clauses (for schematic X elimination)
    Checkpoint grouping (for quick checkpoint grouping and verification)
•
•
Constraint specification (if needed)
•
•
Cutpoint specification (if needed)
•
•
Schematic Reduction specification (if working on reduced-size schematic models)
•
•

file, the tool calculates the phase definitions of all the checkpoints based on the supplied input phase definitions by the designer. All checkpoints then automatically get their default timers based on the computed phase definitions. Designers can define their own timers and can override the computed default timers for the checkpoints. An important feature that enabled quick construction of new timing windows for failed checkpoints was the ability to invoke a parameterized default timer with different arguments. This allowed the designer to move certain signal edges forward

or backward based on the failure condition.

6.2.5 State mapping specification

State mapping between the RTL and the schematic is then specified. This involves mapping both latches and bitcells by specifying their names based on the transistor-level schematic hierarchy. Specification of state mapping was made easy by allowing the following syntax in the configuration file.

- *For loops*: Loops provided a convenient iteration mechanism that allowed quick mapping of iterated instances of latches and bitcells.
- *Function definitions*: Functions could be defined by users so that hierarchical mapping could be done bottom up. These functions were used by designers to define certain primitive cell mappings and were included in all other project team member configuration files.
- *Verilog syntax and quoted strings for signal names*: The users could specify signal names using verilog syntax or quoted strings and could refer to both RTL and schematic names.

State mapped checkpoints automatically got their default timers. Designers could define their own timers and then specify the newly defined timers as part of the state maps. This enabled the designer to customize a timer for a specific checkpoint. A command line option was provided to have a textual printout of the user specified state map that made debugging the state map easier.

6.2.6 X elimination

An important feature that was allowed to be specified as part of the state maps were *required* clauses. The designer could specify any RTL node to be driven in addition to the driven nodes computed by the tool for an assertion. From these additional RTL *required* nodes, a set of schematic checkpoint nodes are computed and then driven. This minimized the occurrence of false negatives by eliminating the propagation of X's in the schematic. If any of the computed additional schematic checkpoint nodes happened to be in the logical support of the schematic checkpoint being verified, then a failure would occur. This is because the consequent of the assertion check derived from the RTL would not include these *required* nodes. If the additional schematic checkpoints were not in the logical support of the checkpoint being verified, then no failure would occur. In both cases, no false positive is possible.

The propagation of X's occurs due to the many timing and sneak paths present in the schematic but not reflected in the RTL. By utilizing symbols instead of X's on the *required* nodes, the inherent pessimism in the simulation algorithm for X's is overcome. This feature was added as a result of our experience using the earlier flow.

6.2.7 Checkpoint Partitioned Verification and Reduction Options

A command-line option to verify a specific checkpoint or groups of checkpoints was provided. This allowed the designer to work on only one or two checkpoints at a time. This allowed faster debug and verification run-times during the initial stages of verification. Each state map had

a keyword *group* that could be used to associate a name with the set of checkpoints in the state map. This name could then be used to verify all the checkpoints in the map in one symbolic simulation run.

Using the reduction option, the designer could specify reduced size models of the schematic to verify. This option in conjunction with the checkpoint group option enabled designers to verify their control logic quickly. Designers could specify the reductions in the configuration file or as part of the command-line arguments. Since the debug times were very fast, designers relied heavily on this option to pinpoint initial problems with their schematics.

6.2.8 Tool Logs and Result Output

Command-line options to capture the tools stdout and stderr in a log file were provided. In addition, a debug output option was provided that dumped verbose debug information in addition to the log output. The verbose debug information included the antecedents and consequents for all the STE properties that were generated for the checkpoints. In case of failures, the failed PLA tables would be output to the debug file.

6.2.9 Simulator Options

An option was provided to stop the symbolic simulation at any time the designer wanted to. This allowed designers to quickly stop and debug a simulation run based on their knowledge of the previous simulation runs. In addition to the time, the designer could also specify the number of checkpoint failures after which simulation would be aborted. Using both these options, the designer could quickly debug the failures.

To prevent oscillations from slowing down the simulation, a maximum oscillation count option was provided that enabled designers to set the oscillating nodes to an X after the number of oscillations exceeded the specified count. This enabled faster verification runs at the expense of potential false negatives.

6.2.10 Visualization and Debug Interface

For any verification tool to be incorporated into an existing design flow, it is mandatory that an efficient debug interface be developed. Based on our initial experience of debugging failures using the old tool, we decided on designing a debug interface that was both textual and visual. For the textual representation, we chose a PLA format with which most designers were familiar with. For a visual representation of the failures, we interfaced Versys2 to a commercial waveform browser.

On a failure, a specific scalar vector is produced that exposes the discrepancy between the RTL and the schematic. Both the reference model (RTL) signals and the schematic model signals could be viewed in the browser. A feature that was provided was the ability to view both the antecedent and the consequent in the waveform browser as shown in Figure 6.2. The antecedent and consequent for a signal are plotted as *signal$ant* and *signal$cons*. This aided the designer in understanding the type of assertion that was generated. Compared to the earlier debug method of examining detailed ternary logic expressions, the visualization interface enabled the designer to pinpoint the exact problem very quickly. In fact, this design decision to debug failures using a waveform browser proved to be a key feature that designers found very useful. It is estimated that this

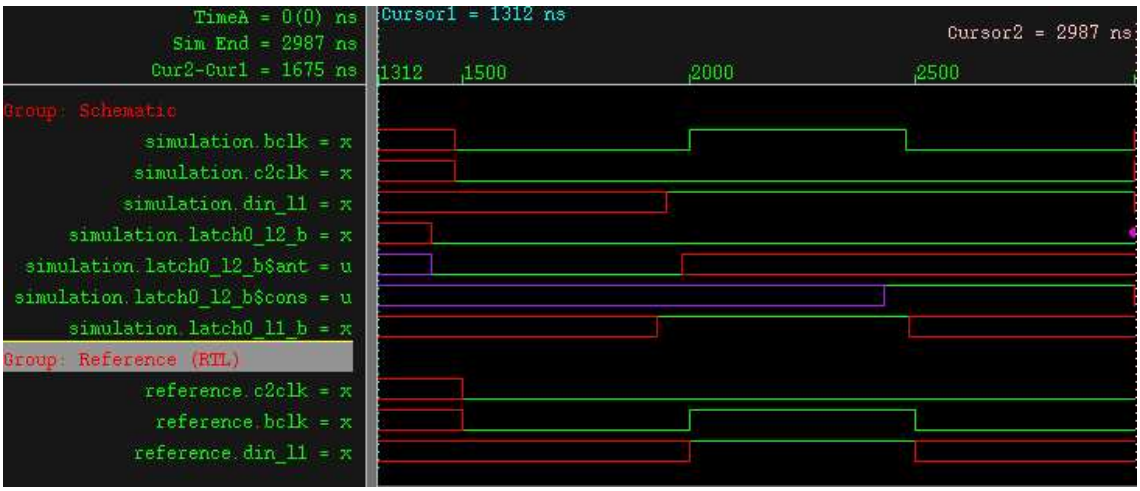


Figure 6.2: Failed Scalar Vector showing Discrepancy

has brought down the debug and diagnosis effort by at least 70-80%.

6.2.11 Switch Model Translator

The default switch-level model for the transistor circuit is bidirectional. This can create an oscillation in the simulation of certain classes of custom circuits such as the sense-amps at the output of the memories. In the old flow, our previous solution to avoid this problem was to make manual structural edits to the sense-amp transistor netlist. Since most custom memories have sense-amp transistors, this was a bottleneck to complete automation.

To automate the attribution of transistors with uni-directional and different strength attributes, we allowed the designer to specify the required transistor modifications in a separate transistor attribute file that could be read by the translator. The file contained a list of the cells and transistors whose attributes were to be modified. This enabled an on-the-

fly generation of a modified switch-level model based on the attribute file and the input schematic and eliminated the manual edits. By eliminating the manual modification of schematics, the tool could now directly work of the project's data repositories.

The translator is also capable of generating a verilog-strength switch-level model from the same circuit implementation. This allowed our methodology to be able to verify two types of switch models as we had elaborated earlier in Section 5.2 of Chapter 5. For example, transistor strength related problems could be identified by first doing a verification run on a verilog-strength switch model and then on a Resist-ratio type (using lengths/widths of transistors) switch model. If the first verification passed and the second did not, this usually meant the failure was in a ratioed circuit and it was up to the circuit designer if further investigation was necessary.

6.2.12 Creation of reduced-size switch-level models

Custom memories have a large number of bitcells and complex control logic in them. With the increase in pipeline depths, these memories are also being designed with a lot of additional latches in them. With this exponential increase in the state space and associated complexity, there is a need for a quick initial analysis and verification.

To enable quicker debug and initial turnaround times, we developed a mechanism that would allow designers to work with reduced-size switch-level models. Designers could specify the instances of cells that would be extracted out when creating the switch-level model. A reduced switch-level model of the schematic is created which is read by the tool. Typically,

the reduced switch-level models consist of a single bitcell or a single row or column of bitcells. The designer works with the reduced-size model, debugs the control logic and establishes all the delays required for correct event ordering in the schematic [71].

Once the reduced-size model is verified, the full-size model is then verified. By combining the ability to verify groups of checkpoints and the ability to work with reduced-size models, the designer is able to verify a custom memory within two to three days.

6.2.13 Software Caches

During the design phase of the tool, it was realized that once the RTL design became stable and was not changing, it was usually the schematic that underwent minor tweaks and changes as tape-out approached. Since the initial analysis of the RTL during automatic assertion generation is compute intensive, it was decided to cache the RTL analysis and intermediate results in a software cache. This enabled the tool to work off the caches when only the schematic had changed. The generation of the software caches is a one time cost that is amortized over subsequent runs. For some memories, this saved us at least 80-90% of the time it would have taken if the caches were not present.

6.2.14 Usage Tracking and User Bug Reporting

In order to keep track of users of the tool, we integrated the tool with an existing usage tracking library. By keeping track of first time users, it was possible for us to be more pro-active in responding to their queries and to assist them in their verification tasks. The tool was also integrated

with a bug reporting library that was developed in-house. A command-line option was provided to Versys2 that enabled users to report software bugs related to Versys2 quickly. This option captured the user's complete run-time environment and the exact command-line arguments that were used to run the tool by the designer. This created a compressed tar file that was then sent to us to help debug the software.

6.2.15 Training

A hands-on training class with workstations was organized and a tutorial on the use of the CAD tool for verifying real designs was presented. A Versys2 manual was created with a FAQ section that helped new users get up to speed with the tool. This class proved to be very valuable for the users and especially for the new designers who had very little or no background on the old verification flow.

6.2.16 BDD and Simulation Engine

Versys2 was interfaced to the public domain CUDD BDD package. This interface was built as a set of layers. This layered interface would enable seamless integration of the Versys2 simulation engine to other BDD packages and would enable it to use the enhancements with new releases of BDD packages. With this BDD interface, there was a performance improvement in the symbolic simulation runs compared to the earlier tool.

6.3 Formal Tool Deployment Approaches and Results

There are two approaches that can ensure the use of formal CAD tools in existing design flows. Either the designers use the tool as part of their daily task or experts are hired into the project for the purpose. The type of deployment is highly dependent on the task accomplished by the formal tool.

Typically, it is argued by verification teams that a second pair of eyes is beneficial to exposing bugs. This is true for formal tools that provide only a yes or no result to the end user. If the tools result is highly dependent on the user input and false positives could occur because of it, then formal verification experts are required to use the tool. If the result of the tool is negative, then the expert has to consult with the designer in debugging the failure. This is because the expert does not have the knowledge of the design as the designer does. The expert has to understand the design and interacts with the designer to identify the source of the problem. Such tools are typically used just before tape-out to ensure that two representations are equivalent in some sense.

Formal tools that enable not only verification of the designs but also design analysis are extremely beneficial to the designers if used by themselves. These tools are used up front in the design cycle since they also enable designers to analyze their design. This added functionality of design analysis in addition to the verification task enables the designer to improve and enhance the design quality. Versys2 has been used by circuit designers to actually modify their schematics based on hazard, contention and race condition failures observed during equivalence checking of RTL and

schematics for custom memories. Circuit designers in PowerPC projects have used the results of Versys2 as an aid to SPICE their circuits and check race condition paths.

Using the earlier tool and flow, it had taken about three person-years for validating over 20 custom memories in one project. Even though there were a number of bugs that were found, the verification effort was still quite high and the tool users were not the designers. All the run-times and memory usages reported below were measured on an UltraSparc machine running SunOS 5.8 and having 1G of RAM memory. Table 6.2 lists some of the memories that were verified. The control logic transistors

Table 6.2: Old flow: Custom memory verification results

Custom Memory	Bitcells	Latches	Control logic transistors	Assertions Runtime (hrs)
A	73728	1346	69000	11-12
B	24576	1612	87500	15-16
C	3968	357	31000	15-16
D	24576	935	44000	16-17
E	131072	330	177500	19-20
F	88704	445	39500	12-13
G	50688	706	156000	11-12
H	71680	276	60000	15-16
I	21824	1192	27000	2-3
J	1024	0	8950	1.5-2
K	4096	0	7900	5-6
L	8512	0	8800	5-6
M	256	0	4500	6-7
N	512	0	1250	1-2
O	2096	0	32400	1-2
P	4192	0	6050	4-5

are all the transistors in the array that do not comprise the bitcells and

latches. The run-times represent the time that it takes for the assertion set to pass successfully on the schematic. From Table 6.2, we can infer that the complexity of the control logic is an important factor influencing assertion run-times in addition to the bitcells and latches. For example, array B takes the same amount of time as array C even though array C has fewer bitcells, latches and control logic transistors as compared to array B. Moreover, all the assertion run-times are less than a day. In contrast, random verilog simulation of these arrays would take anywhere from weeks to months and it would be extremely difficult to quantify the coverage. From the perspective of CPU usage and coverage, our technique is far better than conventional validation methods for these memories. Table 6.3 presents the

Table 6.3: Validation effort

Array Block	Validation Time (man-months)	Discrepancies
A	3	4
B	3	6
C	3	5
D	3	5
E	3	9
F	2	6
G	2	5
H	3	4
I	2	2
J	1	1
K	2	4
L	2	5
M	2	5
N	1	3
O	1	0
P	1	2

validation time and discrepancies that were uncovered using our methodology. Approximately, three person years were spent in the validation of over 20 custom arrays uncovering 66 discrepancies using this validation methodology. The validation time comprised methodology development, partial implementation of the automatic assertion generator, manual state node and timer mapping, assertion generation for bitcells and sense-amp outputs, development of the verification support libraries, establishing the BDD variable order and debugging of assertion failures. About half the validation time was spent on debugging of assertion failures and getting the timer and state node mapping right.

To further reduce the verification effort and enhance designer's productivity, we developed a second-generation custom memory equivalence checker Versys2. This has been used to verify custom memories in two PowerPC projects and is currently being used as the mainstream custom memory RTL vs. schematic verification tool at Somerset, Motorola. In addition to custom memories, Versys2 has also been used to verify custom combinational logic in another PowerPC project. The tool is now being used by circuit designers and has been adopted as the custom memory equivalence checker for tape-out. With the design teams taking control of the tool, the verification effort for these memories has drastically come down by at least $3X$. This is primarily due to quicker debug-edit-run cycles, the simplicity of the initial setup and user-interface to the tool and a better implementation of the symbolic simulation engine. The new flow is shown in Figure 6.3. Versys2 has been integrated into the projects data management framework and is now part of the project's verification regression mechanism for custom memories when either the RTL or schematic

Table 6.4: New flow: Custom memory verification flow using Versys2

Custom Memory	Bitcells	Latches	Control logic transistors	Time (hrs)/ Mem (MB) (w/l)	Time (hrs)/ Mem (MB) (Verilog)
A	73728	1346	69000	8.9/535	5.3/323
B	24576	1612	87500	4.6/335	2.3/321
C	3968	357	31000	6.0/131	3.7/91
D	24576	935	44000	5.2/223	2.1/210
E	131072	330	177500	2.6/404	2.2/337
F	88704	445	39500	1.7/426	0.7/363
G	50688	706	156000	1.2/415	0.5/221
H	71680	276	60000	2.7/416	1.7/298
I	21824	1192	27000	1.4/294	1.2/288
J	1024	0	8950	1.1/95	0.9/86
K	4096	0	7900	0.1/121	0.03/51
LMN	9280	0	14550	4.9/164	3.4/122
O	2096	0	32400	0.6/197	0.25/118
P	4192	0	6050	0.9/146	0.6/99

is modified. Table 6.4 lists the memories that were verified using Versys2 and the new flow. The results shows that the verification runs using the verilog-strength switch model is faster and takes less memory than the resist-ratio switch models. This is because the verilog switch models are smaller in size since the number of strength classes is typically 3, whereas the resist-ratio models based on lengths and widths of transistors could have strength classes anywhere from 3 to 8 leading to a much larger switch-level model. In our experience, the resist-ratio model has worked very well without the circuit designer having to worry about the correct usage of verilog primitives. The new tool was also able to verify the L, M, and N memories as part of a single custom memory that instanti-

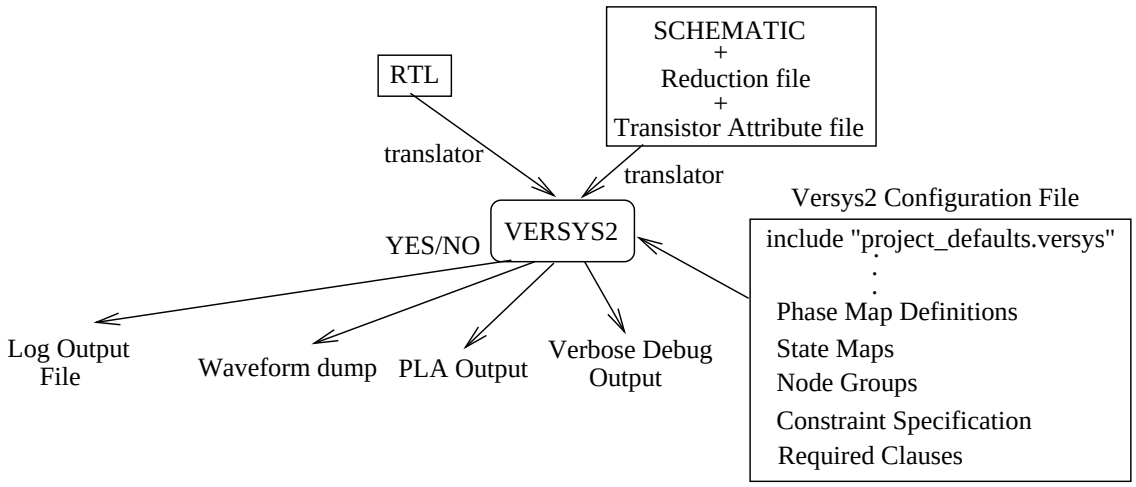


Figure 6.3: Modified Flow with New Tool

ated all of them. The memory validation methodology discovered many logic discrepancies and circuit related problems. These included incorrect clock regenerators feeding the wrong sets of latches, control logic errors in the array read and write enables, incorrect modeling of sense-amp output precharge in the RTL, incorrect hookup of the scan chain in the implementation, incorrect modeling of the primary outputs in the RTL when certain clocks were de-asserted, etc. Many of these bugs would manifest when switching between different modes of operation during test, debug, power-on-reset, etc. These modes are essential for testing and debugging the chip. Validation techniques that only look at the functional behavior will not find these errors. In addition to logical bugs, a number of potential circuit related problems such as shorts between VDD and GND for small periods of time, glitches, and race problems were identified and conveyed to the designers.

We also applied our technique to a number of custom designed

Comb. Ckt.	# PIs	POs	Transistors	Time (secs) / Memory (MB)
A	210	53	5263	15329/145
B	20	1	124	1.4/3
C	132	35	4093	948/55
D	101	54	4644	16.4/3
E	7	2	18	1/3
F	150	92	5994	1960/77
G	13	8	464	1.4/3
H	22	16	669	2.8/3
I	44	32	1753	9.8/7
J	215	52	4027	7.8/3
K	145	8	832	8.9/7
L	206	253	5236	485/39
M	13	2	26	0.86/3
N	13	8	92	0.91/3
O	42	10	643	3.7/3

Table 6.5: Results on Combinational Circuits

dynamic circuits from the Motorola MPC7455 microprocessor, to verify equivalence between respective RTL and transistor level descriptions. Table 6.5 contains verification data relating to fifteen custom combinational circuits. The times shown in the tables include the time taken to generate assertions from the RTL model and the time taken to symbolically simulate the assertions on the switch-level model. Due to differences in circuit structures and/or complexities and the fact that we do not provide any user control over variable ordering, similar sized circuits might take widely varying times. For example, although circuits A and J are both dynamic adders – 52 and 50 bits wide, respectively, the former takes several orders of magnitude more time to verify. Using the constraint derivation algorithm in Chapter 4, both the auto-constraint c_a and the weakening-factor

Comb. Ckt.	# PIs	POs	Transistors	Time (secs) / Memory (MB)			% Time saved
				c_u	c_a	c_{weaken}	
A	210	53	5263	15329/145	13559/104	462/64	96
B	20	1	124	1.4/3	1.5/3	1.36/3	9
C	132	35	4093	948/55	2767/87	746/38	73
D	101	54	4644	16.4/3	91/23	14.9/3	84
E	7	2	18	1.0/3	1.1/3	0.7/3	36
F	150	92	5994	1960/77	5227/80	305/40	94
G	13	8	464	1.4/3	1.6/3	1.45/3	9
H	22	16	669	2.8/3	3.5/3	2.86/3	18
I	44	32	1753	9.8/7	12.6/7	8.6/7	32
J	215	52	4027	7.8/3	8.8/3	7.1/3	19
K	145	8	832	8.9/7	14.7/7	8.6/7	41
L	206	253	5236	485/39	515/37	35/24	93
M	13	2	26	0.86/3	1.24/3	0.91/3	26
N	13	8	92	1.46/3	1.7/3	0.91/3	46
O	42	10	643	3.7/3	4.8/3	2.18/3	54

Table 6.6: Equivalence checking results using c_u, c_a and c_{weaken}

c_{weaken} were generated for the fifteen custom combinational circuits. The verification run-times using c_a and c_{weaken} are shown in Table 6.6. All the combinational circuit verification runs using c_{weaken} as the constraint have a lower run-time than the run-times using the auto-constraint c_a as the constraint. This is because c_{weaken} allows a very small set of behaviors on the primary inputs as compared to c_a, and therefore the simulation trajectories that the circuit goes through is very small. An interesting observation is that for some combinational circuits such as G, H, and M, the c_{weaken} run-times were higher than the run-times using c_u. This can be attributed to the fact that c_{weaken} is comparable to c_u in terms of the size of the set of possible behaviors that can be applied to the primary inputs.

To compare the efficiency of our logarithmic inertial-delay circuit model construction with respect to the linear inertial-delay circuit model construction, we modified the symbolic simulation engine to generate five symbolic simulation engines such that every node in the circuit would get a delay of

$$(d_R, d_F) = \{(3,3), (4,4), (8,8), (16,16), (32,32)\}$$

units respectively. Using the formulation in Section 2.2.3, we automatically generated a state machine model in verilog and then synthesized it without any optimizations to a boolean gate and latch model using the DC synthesis tool from Synopsys [108]. In the simulation engine, the latches were modeled using a single dimensional buffer and the synthesized gate level model was used to implement the next state function that updates the buffer every simulation time tick. This resulted in a circuit model size that is logarithmic in terms of the combined inertial delay of all the nodes in the circuit. A number of symbolic simulation experiments were conducted on custom memories to evaluate the resource usage using both types of inertial-delay circuit constructions. The time and memory usage results are shown in Tables 6.7 and 6.8 respectively. Using the linear inertial-delay circuit construction, circuits B, D, E and G took more than three days of simulation time (T_{lin}) for rise and fall inertial delays of 32 units for each node, whereas the simulation (T_{log}) was able to complete using the logarithmic inertial-delay circuit for each node. T_{lin} for circuits H and I was more than a day but T_{log} was less than six hours. This is because the logarithmic inertial-delay circuit behaves like a counter and therefore requires only 6 delay latches to implement the counter unlike the shift-register delay-buffer like construction which requires 32 delay-latches. Moreover,

Table 6.7: Simulation Time for Linear/Logarithmic Delay Circuit

Custom Mem.	T_{lin}/T_{log} (hrs/hrs) For Rise and Fall Inertial Delays of				
	3	4	8	16	32
A	5.1/5.4	5.3/5.4	5.8/5.5	7.1/5.6	9.9/6.0
B	2.4/2.5	2.5/2.5	3.1/2.6	6.7/3.0	A[1]/4.3
C	3.7/3.6	3.7/3.7	4.1/3.8	5.2/3.9	8.6/5.1
D	2.4/2.4	2.6/2.6	3.0/2.8	4.8/3.7	A/5.3
E	5.3/5.4	5.5/5.4	6.0/5.6	A/7.1	A/8.4
F	0.7/0.8	0.8/0.8	1.2/0.9	1.7/1.0	4.1/1.6
G	1.2/1.2	1.3/1.2	1.7/1.3	4.8/1.9	A/4.3
H	2.4/2.3	2.5/2.4	2.9/2.4	5.3/2.8	37/5.6
I	1.5/1.6	1.6/1.6	1.9/1.7	4.9/2.0	39/5.3
J	1.0/1.0	1.0/1.0	1.3/1.1	2.5/1.4	5.8/2.0
K	.03/.03	.04/.04	.05/.04	.07/.05	.1/.06
LMN	3.9/3.8	4.0/3.8	4.2/3.9	5.4/4.3	7.0/4.7
O	.27/.28	.28/.28	.31/.31	.84/.42	1.4/.55
P	.6/.7	.8/.7	1.0/.8	1.6/.8	2.5/.9

the shift register requires shifting every simulation time tick. The inertial delay counter construction may require more next-state function logic but this has only a minimal impact on the simulation time. As we go from 16 to 32 units of inertial delay per node, the simulation time T_{lin} increases drastically for the shift-register delay circuit whereas the simulation time T_{log} for the counter-like circuit construction does much better. Also, the BDD reordering occurs much more often in the shift-register model which is one of the contributors to T_{lin}. The memory usage also deteriorates as we go from 16 to 32 units of inertial delay but is still within the RAM size of the machine. The effect of the number of delay-latches on simulation

[1]A: Abort, ran for > 3 days

Table 6.8: Memory Usage for Linear/Logarithmic Delay Circuit

Custom Mem.	M_{lin}/M_{log} (MB/MB) For Rise and Fall Inertial Delays of				
	3	4	8	16	32
A	327/326	327/326	330/326	351/327	360/328
B	323/324	323/324	335/325	350/327	357/328
C	90/91	90/91	95/91	101/93	106/95
D	211/210	211/210	216/211	231/214	234/218
E	350/354	350/354	359/353	441/358	451/359
F	362/363	362/363	363/362	376/363	380/368
G	210/212	210/212	216/213	253/216	258/221
H	302/300	302/300	307/300	323/304	331/305
I	290/290	290/290	291/291	312/294	320/299
J	87/86	87/86	91/87	92/88	93/89
K	51/51	51/51	51/51	52/52	52/54
LMN	122/124	122/124	125/124	131/125	134/125
O	121/121	121/121	123/122	133/123	135/124
P	99/100	99/100	100/100	103/100	104/100

performance is exacerbated as the inertial-delay increases. For example, a million-unit inertial delay buffer would require a shift-register construction that has a million delay-latches, whereas a counter-like construction would require only twenty-one delay-latches. This deterioration in simulation performance is also dependent on the machine architecture since the update of the delay-buffer can be really fast if the delay-buffer fits into a register or a cache line. However, for small circuits such as J, K, P and LMN, where the number of nodes that are annotated with inertial delays is small, the counter-like construction does not seem to have any added benefit. For example, circuit P takes more time and memory to simulate for a 3-unit inertial delay counter circuit. A possible solution would be to construct shift-register like inertial delays for small values of delays and

counter-like delay circuits for large values of delays based on a threshold number.

Chapter 7

Conclusions and Future Research

This dissertation has presented a rigorous and sound verification methodology for implementation verification between an RTL specification and its corresponding transistor circuit implementation using symbolic simulation. We have demonstrated the inadequacy of current boolean function extraction techniques. We established that the notion of state transitions in formal tools and time progress in simulation are equivalent formulations of computation progress. Ultimately, it is the recognition of state that influences the simulation model extracted from an implementation. Simulation plays an important role in the design flow primarily because it allows designers to analyze and experiment with specifications and implementations. By abstracting out the irrelevant details of current symbolic simulation engines, this dissertation presented a general method of deriving an efficient data-independent event-driven symbolic simulator from an existing event-driven simulator for simulating circuits with transport and inertial delays.

We presented a rigorous methodology for verifying custom memories and custom logic and showed how some of the complexity issues were dealt with. We showed how our methodology is an improvement over current state-of-the-art commercial tools. From the perspective of bug coverage and verification complexity, we analyzed the benefits of state

mapping over that of the product machine approach. By defining a new class of logic bugs, we have shown that our implementation verification methodology is more robust and has a higher coverage compared to PMA methods. We introduced a fundamental problem in the implementation verification of dynamic circuits and showed how closely tied it is to the latter stages of the design flow. We presented an automatic technique for deriving constraints and provided an enhanced framework for implementation verification utilizing these constraints. This dissertation also made the case for eliminating gate/switch-level simulations for functional simulation provided our implementation verification methodology is adopted. We finally presented the features of an industrial-strength implementation verification tool incorporating these ideas.

From a technical standpoint, symbolic simulation is a viable implementation verification technique and is capable of dealing with a wide variety of models. Symbolic simulation can benefit directly from the enhancements to the switch-level models and can utilize better delay models. It is also effective in finding bugs and analyzing circuit-related problems such as races and glitches. Uncovering bugs is computationally easier than proving correctness. Assuming a potential cost for each bug, it is easy to see the payoff for symbolic simulation. From an economic perspective, validation and analysis using symbolic simulation can fit neatly into existing design and validation methodologies. The potential benefits of improved product quality (fewer errors), reduced time-to-market and lower validation costs (smaller number of highly-skilled people) seem to warrant further investment and effort in this area. Whereas other approaches discussed in Chapter 1 do not meet all of the three criteria, our experience has shown

that symbolic simulation does pay off on all of the three metrics. Moreover, this approach ties together the thought processes of two sets of people in a design company, the circuit designers and the design architects. The work presented here shows that a symbolic simulation methodology can address both the economic and technical concerns.

7.1 Future Research Directions

The verification methodology presented in this dissertation provides a firm foundation for implementation verification. Much more work is needed before the full potential of symbolic simulation can be realized in other areas such as timing analysis, noise analysis, power estimation, behavioral specification property checking, etc. The other challenge is to be able to fit the methodologies into existing design flows seamlessly. Some of the problems that we intend to pursue are

- Identify techniques for adaptive and more accurate switch-level model generation for correct behavior modeling of custom transistor circuits. Develop calibration techniques so that the switch-level models can accurately track different process technologies.

- Develop algorithms to enable use of a mix of PMA and SMA techniques for behavioral vs. RTL equivalence checking.

- Investigate automatic symbolic simulator derivations from a specification and explore data structures that will enable efficient higher-order symbolic simulators to be automatically generated. Develop techniques that can encode infinite domains by restricting them to

finite domains and then using mathematical induction techniques to prove the properties.

- Investigate algorithms that will enable use of ATPG and SAT-based techniques for symbolic simulation.
- Explore higher-order type theory and develop algorithms for generating efficient simulation engines.

The ultimate objective for EDA tools and computer-aided design is to be able to develop push-button methodologies for designing correct-by-construction designs from high-level specifications. This reduces the design cycle time and improves time-to-market. Thus developing revolutionary EDA tools and breakthrough methodologies poses a great and exciting challenge.

Bibliography

[1] M. S. Abadir, K. L. Albin, J. Havlicek, N. Krishnamurthy, A. K. Martin "Formal Verification Successes at Motorola" *Formal Methods in System Design,* Vol. 22, pp. 117-123, Kluwer Academic Publishers, March 2003

[2] J. A. Abraham "Functional Level Test Generation for Complex Digital Systems" *IEEE International Test Conference,* 1981

[3] S. B. Akers "Binary Decision Diagrams" *IEEE Transactions on Computers,* C-27(6), pp. 509-516, June 1978

[4] M. Abramovici, Melvin A. Breuer and Arthur D. Friedman "Digital Systems Testing and Testable Design" *Revised Printing, IEEE Press* 1990

[5] Derek L. Beatty, Randal E. Bryant, and Carl-Johan H. Seger "Synchronous Circuit Verification by Symbolic Simulation: An Illustration," *Advanced Research in VLSI: Proceedings of the 6th MIT Conference,* pp. 98-112, MIT Press, March 1990.

[6] Derek L. Beatty "A Methodology for Formal Hardware Verification with Application to Microprocessors" *Ph.D. Thesis, published as Technical Report CMU-CS-93-190, August 1993,* School of Computer Science, CMU.

[7] Derek L. Beatty and Randal E. Bryant "Formally verifying a microprocessor using a simulation methodology" *Design Automation Conference, 1994*

[8] L. Bening, H. Foster, "Principles of Verifiable RTL Design" *www.verifiableRTL.com* 2nd Edition, Kluwer 2001

[9] J. Bhadra and N. Krishnamurthy "Automatic Generation of Design Constraints in Verifying High Performance Embedded Dynamic Circuits" *International Test Conference* 2002

[10] G. Birkhoff "Lattice Theory" *The American Mathematical Society Colloquium Publications,* Vol. XXV, American Mathematical Society, 3rd Edition, 1967

[11] D. T. Blaauw, R. B. Mueller-Thuns, D. G. Saab, P. Banerjee, J. A. Abraham "SNEL: a switch-level simulator using multiple levels of functional abstraction" *IEEE International Conference on Computer-Aided Design (ICCAD)* pp. 66-69, Nov. 1990

[12] D. T. Blaauw, D. G. Saab, P. Banerjee, J. A. Abraham "Functional Abstraction of Logic Gates for Switch-Level Simulation" *, European Conference on Design Automation (EDAC)* pp. 329-333, Feb. 1991

[13] M. Boehner "LOGEX - An Automatic Logic Extractor from Transistor to Gate Level for CMOS Technology" *25th Design Automation Conference,* pp. 517-522, 1988

[14] R. S. Boyer and J. S. Moore "A Computational Logic" *ACM Monograph Series. Academic Press, New York,* 1979

[15] D. Brand "Verification of Large Synthesized Designs" *IEEE/ACM International Conference on Computer-Aided Design,* pp. 534-537, Nov. 1993

[16] Randal E. Bryant "A Switch-Level Model and Simulator for MOS Digital Systems" *IEEE Transactions on Computers,* vol. C-33, no. 2, Feb 1984, pp. 160-177.

[17] Randal E. Bryant "Can a simulator verify a circuit" *Formal Aspects of VLSI Design,* pp. 125-6, G. J. Milne and P. A. Subrahmanyam, editors, North Holland, Amsterdam, 1986

[18] Randal E. Bryant "Algorithmic Aspects of Symbolic Switch Network Analysis" *IEEE Transactions on Computer-Aided Design,* CAD-6(4), July 1987, pp. 618-633

[19] Randal E. Bryant "Boolean Analysis of MOS Circuits" *IEEE Transactions on CAD of Integrated Circuits,* CAD-6(4), July 1987.

[20] Randal E. Bryant "Graph-Based Algorithms for Boolean Function Manipulation" *IEEE Transactions on Computers,* Vol. C-35, No. 8, pp. 677-691, August 1986

[21] Randal E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler "COSMOS: a compiled simulator for MOS circuits" *24th Design Automation Conference,* pp. 9-16, 1987

[22] Randal E. Bryant "Verifying a Static RAM Design by Logic Simulation" *Fifth MIT Conference on Advanced Research in VLSI,* 1988, pp. 335-349.

[23] Randal E. Bryant "Verification of synchronous circuits by symbolic logic simulation" *Hardware Specification, Verification and Synthesis: Mathematical Aspects,* no. 408 in LNCS, Miriam Leeser and Geoffrey Brown, editors, Springer-Verlag, Berlin, 1989

[24] Randal E. Bryant "Symbolic Simulation - techniques and applications" *Design Automation Conference, 1990*

[25] Randal E. Bryant and Carl-Johan H. Seger "Formal Verification of Digital Circuits Using Symbolic Ternary System Models" *Computer-Aided Verification '90, E. M. Clarke, and R. P. Kurshan, eds.,* American Mathematical Society, 1991, pp. 121-146.

[26] J. A. Brzozowski and C-J. Seger "A Unified Framework for Race Analysis of Asynchronous Networks" *Journal of the Association for Computing Machinery,* Vol. 36, No. 1, pp. 20-45, January 1989

[27] J. A. Brzozowski, Z. Esik and Y. Iland "Algebras for hazard detection" *31st IEEE International Symposium on Multiple-Valued Logic,* Proceedings, pp 3-12, May 2001

[28] Jerry R. Burch "Trace Algebra for Automatic Verification of Real-Time Concurrent Systems" *PhD Thesis, published as Tech Report CMU-CS-92-179,* CMU, Pittsburgh, PA, Aug 1992

[29] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill "Sequential circuit verification using symbolic model checking" *ACM/IEEE Design Automation Conf, Orlando, FL.* June 1990, pp. 46-51

[30] J. P. Burgess "Basic Tense Logic" *Handbook of Philosophical Logic. Volume II: Extensions of Classical Logic,* editors D. Gabbay and F. Guenthner, pp. 89-134, D. Reidel, 1984

[31] G. Cabodi and P. Camurati "Symbolic FSM Traversals Based on the Transition Relation" *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* May 1997, Vol. 16, No. 5, pp 448-457

[32] W. C. Carter, W. H. Joyner Jr., and D. Brand "Symbolic Simulation for Correct Machine Design" *16th ACM/IEEE Design Automation Conference,* pp. 280-286, 197

[33] E. Cerny, J. P. Hayes, and N. C. Rumin "Accuracy of Magnitude-Class Calculations in Switch-Level Modeling" *IEEE Transactions on Computer-Aided Design,* pp. 443-452, April 1992

[34] S. G. Chappell and S. S. Yau "Simulation of Large Asynchronous Logic Circuits Using an Ambiguous Gate Model" *Proceedings Fall Joint Computer Conf.* pp. 651-661, 1971

[35] E. M. Clarke and E. A. Emerson "Synthesis of synchronization skeletons for branching time temporal logic" *Logic of Programs: Workshop, Yorktown Heights, NY* volume 131 of LNCS. Springer-Verlag, May, 1981

[36] E. M. Clarke, E. A. Emerson and A. P. Sistla "Automatic verification of finite-state concurrent systems using temporal logic specifications" *ACM Transactions on Programming Languages and Systems* 8(2):244-263, 1986

[37] E. M. Clarke, O. Grumberg, and D. A. Peled "Model Checking" *The MIT Press* 1999

[38] W. E. Cory "Symbolic Simulation for Functional Verification with ADLIB and SDL" *18th Design Automation Conference* pp. 82-89, 1981

[39] Oliver Coudert, Christian Berthet, and Jean Christophe Madre. "Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams" *CAV' 90* Rutgers, NJ, June 1990

[40] Oliver Coudert, Christian Berthet, and Jean Christophe Madre. "Verification of Synchronous Sequential Machines based on Symbolic Execution" *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France* vol. 407 of Lecture Notes in Computer Science, Springer-Verlag, June 1989.

[41] O. Coudert, C. Berthet, and J. C. Madre "Verification of sequential machines based on symbolic execution" *, Lecture Notes in Computer Science 407., Springer-Verlag* 1989, pp. 365-373

[42] Clifford E. Cummings "Correct Methods for Adding Delays to Verilog Behavioral Models" *HDL Conference* 1999

[43] Clifford E. Cummings, "RTL Coding Styles That Yield Simulation and Synthesis Mismatches" *www.sunburst-design.com/papers/ CummingsSNUG1999SJ_SynthMismatch_rev1_1.pdf* , San Jose SNUG 1999

[44] J. A. Darringer "The Application of Program Verification Techniques to Hardware Verification" *16th ACM/IEEE Design Automation Conference,* pp. 375-381, 1979

[45] S. Devadas, H. K. Ma, and R. Newton "On the Verification of Sequential Machines at Differing Levels of Abstraction" *IEEE Transactions on CAD* vol. 7, No. 6, June 1988

[46] David L. Dill "Trace Theory for Automatic Hierarchical Verification of Speed Independent Circuts" *PhD Thesis, published as Tech Report CMU-CS-88-119,* CMU, Pittsburgh, PA, Feb 1988

[47] E. B. Eichelberger "Hazard Detection in Combinational and Sequential Circuits" *IBM Journal of Research and Development,* Vol. 9, pp. 90-99, March, 1965.

[48] H. Foster "Understanding Semantic Inconsistency" *www.demosondemand.com/verplex/goldensilicon.asp* White Paper, Verplex Systems Inc., 26th Feb 2002

[49] Neeta Ganguly, Magdy Abadir and Manish Pandey "PowerPC Array Verification Methodology using Formal Techniques" *International Test Conference,* Washington DC., 1996, pp. 857-864.

[50] Evgueni I. Goldberg, Mukul R. Prasad, and Robert K. Brayton "Using SAT for Combinational Equivalence Checking" *Design Automation and Test Conference,* pp. 114-121, March 2001

[51] Michael J. C. Gordon "HOL: a proof generating system for higher-order logic" *VLSI Specification, Verification and Synthesis* editors G Birtwistle and P. A. Subrahamanyam, pp. 73-128, Kluwer Academic Publishers, Boston, 1987

[52] Aarti Gupta "Formal hardware verification methods: A survey" *Formal Methods in System Design* 1, 2/3 pp. 151-238, Oct. 1992

[53] J. P. Hayes "Uncertainity, Energy, and Multiple-Valued Logics" *IEEE Transactions on Computers,* Vol. C-35, No. 2, pp. 107-114, Feb 1986.

[54] Scott Hazelhurst "Compositional Model Checking of Partially Ordered State Spaces" *PhD Thesis, The University of British Columbia,* Vancouver, Jan 1996

[55] Frederick C. Hennie "Finite-State Models for Logical Machines" *Massachusetts Institute of Technology,* Library of Congress Catalog Card Number: 67-29935, John Wiley and Sons, Inc., 1968

[56] H. Howe, "Pre- and Postsynthesis Simulation Mismatches" *Proceedings of the 6th International Verilog HDL Conference* March, 1997

[57] D. A. Huffman "The synthesis of sequential switching circuits" In *Sequential Machines: Selected Papers*, E. F. Moore, ed. Addison-Wesley, Reading, Mass., pp. 3-62, 1964

[58] G. E. Hughes and M. J. Creswell "Introduction to Modal Logic" *Methuen, London, 1977*

[59] E. V. Huntington "Sets of independent postulates for the algebra of logic" *Transactions. American Mathematical Society,* 5, pp. 288-309, 1904

[60] S. Napper "Equivalence Checking for SOC Custom Blocks" *Cover Story: Integrated System Design* January 2001

[61] P. Jain, P. Kudva and G. Gopalakrishnan "Towards a verification technique for large synchronous circuits" *CAV, 1992*

[62] T. Kam and P. A. Subrahmanyam "Comparing Layouts with HDL models: A Formal Verification Technique" *IEEE Transactions on Computer-Aided Design,* pp. 503-509, April 1995

[63] M. Kaufmann, A. K. Martin, and C. Pixley "Design Constraints In Symbolic Model Checking" *Computer Aided Verification, 1998.*

[64] C. Kern and Mark R. Greenstreet "Formal Verification in Hardware Design: A Survey" *ACM,* 1997

[65] Z. Kohavi "Switching and Finite Automata Theory" *2nd ed., Comput. Sci. Ser. New York: McGraw-Hill* 1978

[66] Alfred Kölbl, James Kukula, Kurt Antreich, and Robert Damiano "Handling Special Constructs in Symbolic Simulation" *Design Automation Conference* pp. 105-110, June 2002

[67] S. A. Kripke "Semantical Considerations on Modal Logic" *Proceedings of a Colloquium: Modal and Many Valued Logics,* Vol. 16 of Acta Philosophica Fennica, pp. 83-94, August 1963

[68] S. A. Kripke "Outline of a theory of truth" *Journal of Philosophy,* 72:690-716, 1975

[69] Narayanan Krishnamurthy, Andrew K. Martin, Magdy S. Abadir, Jacob A. Abraham "Equivalence Checking for PowerPC Custom Memories using Symbolic Trajectory Evaluation" *IEEE International Workshop on Microprocessor Test and Verification, MTV99,* 1999

[70] Narayanan Krishnamurthy, Andrew K. Martin, Magdy S. Abadir, Jacob A. Abraham "Validation of PowerPC Custom Memories using Symbolic Simulation" *18th IEEE VLSI Test Symposium,* April 2000, pp. 9-14

[71] N. Krishnamurthy, A. K. Martin, M. S. Abadir, J. A. Abraham "Validating PowerPC Microprocessor Custom Memories" *IEEE Design and Test of Computers,* Vol. 17, No. 4, Oct-Dec 2000, pp. 61-76

[72] N. Krishnamurthy, A. K. Martin, M. S. Abadir, J. A. Abraham "Design and Development Paradigm for Industrial Formal Verification CAD Tools" *IEEE Design and Test of Computers,* Vol. 18, No. 4, Jul-Aug 2001, pp. 26-35

[73] N. Krishnamurthy, J. Bhadra, M. S. Abadir, J. A. Abraham "Is State Mapping Essential for Equivalence Checking Custom Memories in Scan-Based Designs" *20th IEEE VLSI Test Symposium,* April 2002, pp. 275-280

[74] N. Krishnamurthy, J. Bhadra, M. S. Abadir, J. A. Abraham "Towards the Complete Elimination of Gate/Switch-Level Simulations" *to appear in the proceedings of the 17th IEEE International Conference on VLSI Design,* 2004

[75] Thomas Kropf "Introduction to Formal Hardware Verification" *Springer-Verlag,* 1999

[76] Sandip S. Kundu "Gatemaker: A Transistor to Gate Level Model Extractor for Simulation, Automatic Test Pattern Generation and Verification" *IEEE International Test Conference, 1998*

[77] R. P. Kurshan "Reducibility in analysis of coordination" *In P. Varaiya and A. B. Kurzhanski, editors, Discrete Event Systems: Models and Applications,* Vol. 430 of LNCIS, pp. 414-453, HASA, Springer-Verlag, Aug 1987

[78] R. P. Kurshan "Analysis of discrete event coordination" *In J. W. de Bakker and W. P. de Roever and G. Rozenberg, editor, Stepwise Refinement of Distributed Systems,* Vol. 430 of LNCS, pp. 414-453, REX Project, Springer-Verlag, May 1989

[79] D. E. Long. "Model Checking, Abstraction, and Compositional Verification" *PhD Thesis, School of Computer Science, Carnegie Mellon University* CMU-CS-93-178, July, 1993.

[80] Jean C. Madre "Original concepts of Priam, an industrial tool for efficient formal verification of combinational circuits" *The Fusion of Hardware Design and Verification* pp. 487-501, G. Milne, editor. Elsevier Science Publishers, 1988.

[81] Andrew K. Martin "Trace-Automata: A Formal Framework for Using Abstraction to Verify Hybrid Systems" *PhD Thesis, The University of British Columbia,* Vancouver, Nov 1996

[82] Clayton B. McDonald and Randal E. Bryant "Symbolic Functional and Timing Verification of Transistor-Level Circuits" *International Conference on Computer-Aided Design, ICCAD 99,* Nov. 1999, pp. 526-530

[83] Clayton B. McDonald and Randal E. Bryant "Symbolic Timing Simulation using Cluster Scheduling" *Design Automation Conference,* June 2000, pp. 254-259

[84] Clayton B. McDonald and Randal E. Bryant "Computing Logic-Stage Delays using Circuit Simulation and Symbolic Elmore Analysis" *Design Automation Conference,* June 2001, pp. 283-288

[85] Michael C. McFarland "Formal Verification of Sequential Hardware: A Tutorial" *IEEE transactions on Computer-Aided Design of Integrated Circuits and Systems* vol. 12, no. 5, May 1993, pp. 633-654.

[86] Kenneth L. McMillan "Symbolic Model Checking: An approach to the state explosion problem" *PhD thesis, published as Technical report CMU-CS-92-131* School of Computer Science, CMU, Pittsburgh, PA, May 1992

[87] G. H. Mealy "A method for synthesizing sequential circuits" *Bell System Technical Journal* 34(5), pp. 1045-1079, 1955

[88] Marvin L. Minsky "Form and Content in Computer Science" *ACM Turing Award Lecture* Journal of the ACM, Vol. 17, No. 2, April 1970

[89] E. F. Moore "Gedanken-experiments on sequential machines" *Automata Studies,* Shannon C. E. and McCarthy, J., Eds., Princeton University Press, 1956

[90] S. Owre, J. M. Rushby and N. Shankar "PVS: A prototype verification system" *In Kapur, D. (ed.), 11th International Conference on Automated Deduction,* Volume 607 of Lecture Notes in Computer Science, Saratoga, NY, pp. 748-752. Springer-Verlag.

[91] Manish Pandey, Richard Raimi, Derek L. Beatty and Randal E. Bryant "Formal Verification of PowerPC arrays using symbolic trajectory evaluation" *33rd ACM/IEEE Design Automation Conference,* June 1996, pp. 649-654

[92] Manish Pandey "Formal Verification of Memory Arrays" *Ph.D. Thesis, published as Technical Report CMU-CS-97-162, May 1997,* School of Computer Science, CMU.

[93] Manish Pandey, Richard Raimi, Randal E. Bryant, and Magdy S. Abadir "Formal Verification of Content Addressable Memories using Symbolic Trajectory Evaluation" *34th ACM/IEEE Design Automation Conference,* June 1997

[94] G. Parthasarathy, C.-Y. Huang, and K.-T. Cheng "An Analysis of ATPG and SAT Algorithms for Formal Verification" *High-Level Design Verification and Test Workshop,* pp. 177-182, Nov. 2001

[95] V. Paruthi and A. Kuehlmann "Equivalence Checking using a Structural SAT-solver, BDDs, and Simulation" *International Conference on Computer Design,* Sept. 2000

[96] Carl Pixley "Introduction to a Computational Theory and Implementation of Sequential Hardware Equivalence" *2nd International Conference, CAV 90* editors E. M. Clarke and R. P. Kurshan, June 1990, pp. 54-63

[97] A. Pnueli "A temporal logic of concurrent programs" *Theoretical Computer Science, 13:45-60, 1981*

[98] D. Price "Pentium FDIV flaw-lessons learned" *IEEE Micro* Vol. 15, Issue 2, pp. 86-88, April 1995

[99] J. P. Roth "Diagnosis of Automata Failures: A Calculus and a Method" *IBM Journal of Research and Development,* Vol. 10, No. 4, pp. 278-291, July 1966.

[100] J. P. Roth "Computer Logic, Testing, and Verification" *Computer Science Press, 1980*

[101] D. G. Saab, J. A. Abraham and V. M. Vedula "Formal verification using bounded model checking: SAT versus sequential ATPG engines" *16th International Conference on VLSI Design,* pp. 243-248, Jan 2003

[102] Carl-Johan H. Seger and R. E. Bryant "Modeling of Circuit Delays in Symbolic Simulation" *Formal VLSI Correctness Verification,* Vol. 2, Elsevier Science Publishers, 1990

[103] Carl-Johan H. Seger "Voss-a formal hardware verification system: user's guide," *Technical Report 93-45, Dept. of Computer Science, University of British Columbia, 1993*

[104] Carl-Johan H. Seger and Randal E. Bryant "Formal Verification by Symbolic Evaluation of Paritially Ordered Trajectories," *Formal Methods in System Design,* vol. 6, 1995, pp. 147-189.

[105] K. J. Singh and P. A. Subrahmanyam "Extracting RTL Models from Transistor Netlists, " *IEEE/ACM International Conference on Computer-Aided Design,* Nov 1995, pp. 11-17

[106] K. J. Supowit and S. J. Friedman "A new method of Verifying Sequential Circuits" *Proceedings of teh 23rd Design Automation Conference,* 1986

[107] Formality Equivalence Checker "http://www.synopsys.com/products/verification/formality_ds.html"

[108] Design Compiler "http://www.synopsys.com/products/logic/logic.html"

[109] "A Lattice-Theoretical Fixpoint Theorem and its Applications" *Pacific J. Math.,* pp. 285-309, 1955

[110] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli "Implicit enumeration of finite state machines using BDD's" *Proc. IEEE ICCAD* Nov. 1990, pp. 130-133

[111] R. B. Mueller-Thuns, D. G. Saab, J. A. Abraham "Parallel switch-level simulation for VLSI" *European Conference on Design Automation (EDAC)* pp. 324-328, Feb. 1991

[112] E. G. Ulrich "Exclusive Simulation of Activity in Digital Networks" *Communications of the ACM,* Vol. 13, pp. 102-110, Feb 1969

[113] E. G. Ulrich and Dennis Hebert "Speed and Accuracy in Digital Network Simulation based on Structural Modeling" *19th Design Automation Conference,* pp. 587-593, 1982

[114] S. T. Unger "Asynchronous Sequential Switching Circuits" *Wiley-Interscience,* New York, 1969

[115] Jean G. Vaucher and Pierre Duval "Programming Techniques: A Comparison of Simulation Event List Algorithms" *Communications of the ACM,* Vol. 18, No. 4, pp. 223-230, April 1975

[116] Design Automation Standards Committee "IEEE Standard Verilog Hardware Description Language: IEEE Std 1364-2001" *IEEE-SA Standards Board,* pp. 84-103, 17th March 2001

[117] Conformal Logic Equivalence Checker (LEC) "http://www.verplex.com/products/lec_brochure.html"

[118] Li-Chung Wang, Magdy Abadir and Narayanan Krishnamurthy "Automatic Generation of Assertions for Formal Verification of PowerPC Microprocessor Arrays Using Symbolic Trajectory Evaluation" *35th Design Automation Conference, 1998*

[119] Neil H. Weste, K. Eshraghian "Principles of CMOS VLSI Design: A Systems Perspective" Revised 2nd Edition, Addison Wesley 1993

[120] Jin Yang and Carl-Johan H. Seger "Introduction to Generalized Symbolic Trajectory Evaluation" *International Conference on Computer Design* pp. 360-365, Sept. 2001

Vita

Narayanan Krishnamurthy (Nari) was born to B. S. Krishnamurthy and Pratima Krishnamurthy on the 23rd of March 1967. He graduated with a B. Tech (Hons.) in Instrumentation Engineering (EE) from the Indian Institute of Technology, Kharagpur, India in 1988. From 1988 to 1991, he worked as a Computer Control Engineer at Tata Steel, Jamshedpur, India and from 1991 to 1994, he was employed as a Software R&D Engineer at Sanmar Electronics, Madras, India. He obtained his M.S.E in Electrical and Computer Engineering from the University of Texas at Austin in 1996. From 1997 onwards, he has been working as a CAD tools and methodology developer at the PowerPC Design Center in Motorola, Austin, while simultaneously pursuing his PhD at the University of Texas at Austin. He is currently a Principal Software Staff Engineer in the High Performance Tools/Methodology group at the PowerPC Design Center in Motorola.

Permanent address: 8107 Tallyho Trail
Austin, Texas 78729

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.