INTEGRATED TECHNIQUES FOR THE FORMAL VERIFICATION AND VALIDATION OF DIGITAL SYSTEMS

Approved by:

Dr. Mitchell A. Thornton (Chair & Dissertation Director)

Dr. Hesham El-Rewini

Dr. Theodore Manikas

Dr. Sukumaran Nair

Dr. John Provence

Dr. Stephen A. Szygenda

INTEGRATED TECHNIQUES FOR THE FORMAL VERIFICATION AND VALIDATION OF DIGITAL SYSTEMS

A Dissertation Presented to the Graduate Faculty of the

School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Doctor of Philosophy

With a

Major in Computer Engineering

By

Lun Li

(B. S. E. E., Beijing Jiatong University)
(M. S. E. E., Beijing Jiatong University)
(M. S. E. E., The University of Tulsa)

May 20, 2006

ACKNOWLEDGEMENTS

So many faculty members, persons, and students helped and influenced my work at Southern Methodist University. First, I gratefully acknowledge Dr. Mitchell A. Thornton, my dissertation advisor, for his guidance, encouragement and support throughout the research phase of this dissertation. Without him, I would not have been able to complete the projects and this dissertation. I have learned a great deal through my years of studying under Dr. Thornton, and highly respect him as both a mentor and a friend. He also helped me publish my work and urged me to attend a variety of national and international conferences. It is a great pleasure and honor to work with him.

I also wish to express my sincere appreciation to Dr. Hesham El-Rewini, Dr. Theodore Manikas, Dr. Sukumaran Nair, Dr. John Provence, and Dr. Stephen Szygenda for their precious time and advice as my committee members.

In addition, I would like to extend my hearty thanks to all my collaborators, Dr. David W. Matula, Dr. Marek Perkowski, and Dr. Rolf Drechsler. I especially enjoyed the discussions and collaborations with Dr. Matula on the integer power operation algorithms and circuits sponsored by Semiconductor Research Corporation.

I greatly appreciate the CAD methods group for providing me such a wonderful environment to conduct my study and research. I really enjoyed the discussions with

Kenneth Fazel, Jason Moore, Mahsan Amoui, Poramate Ongsakorn, David Goodman, and Laura Spenner. I would like to express my sincere appreciation to my friends, Chris Wynne, Ganglei Liu, Yue Jia, Yu Meng, Li Ma, as well as others who helped me a lot in Dallas.

My parents played a vital role in my education. From early grade school, they taught me how to study and the importance of education. They have followed my progress with a great deal of enthusiasm and helped in many ways too numerous to mention.

Finally, I must express my deepest appreciation to my family, especially to my wife, He Jin. From the beginning of our marriage, she has taken many additional responsibilities around our home to support my education. She is the person with whom I can share my feelings and frustrations, and she is the person who knows what to say to help me over hurdles. Without her love, encouragement and support this dissertation would not have been possible. The happiness of completing my degree belongs as much to her as it does to me. Our daughter, Grace Li, brings us lots of fun during and tremendous relief when I am under pressure. I also wish for my coming son to enjoy the happiness of our family. I am truly blessed to have such great wife, daughter and son.

Li, Lun

B. S. E. E., Beijing Jiatong University, 1997
M. S. E. E., Beijing Jiatong University, 2000
M. S. E. E., The University of Tulsa, 2002

Integrated Techniques for the Formal Verification and Validation of Digital Systems

Advisor: Professor Mitchell A. Thornton

Doctor of Philosophy conferred May, 20, 2006

Dissertation completed April, 25, 2006

Chip capacity follows Moore's law, and chips are commonly produced at the time of this writing with over 70 million gates per device. However, ensuring correct functional behavior of such large designs becomes more and more challenging.

Simulation is a predominantly used tool to validate a design in industry. Simulation can validate all possible behaviors of a design in a brute-force manner. However, rapidly evolving markets demand short design cycles while the increasing complexity of a design necessarily dictates that simulation coverage is less and less complete. Formal verification validates the correctness of the implementation of a design with respect to its specification by applying mathematical proofs.

Image/Pre-Image computation is a core algorithm in formal verification. *Binary Decision Diagram* (BDD) -based methods are usually faster but can exceed memory capacity for some types of designs which therefore limits scalability. *Satisfiability* (SAT) solvers are less vulnerable to memory explosion but slow when all the satisfied solutions are required in image computation. In this work, a genetic algorithm based conjunctive scheduling solution is presented to enhance BDD-based image computation. A way of combining BDD and SAT approaches for image computation is also presented to solve the state space explosion in image computation. A BDD-based approximation method is used to calculate the over- and under- boundaries of reachable states. A SAT solver is used to find the remaining states. The SAT solver is enhanced by techniques referred as “early detection” and “expansion” to find a satisfiable assignment containing more don’t cares.

Formal verification itself cannot solely accomplish the validation task. Thus, combining different approaches together to serve the purpose of validation of digital circuits attracts our attention. The third part of this work focuses on the *Integrated Design Validation* (IDV) system that develops an integrated framework to the design validation and takes advantage of current technology in the areas of simulation and formal verification resulting in a practical validation engine with reasonable runtime. To demonstrate the ability of the IDV system, IDV is applied to two practical application circuits designed in our lab for the SRC sponsored arithmetic circuit project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

VLSI design sizes grow as VLSI fabrication technology can handle smaller and smaller feature sizes. With the success of CAD tools in logic synthesis and the use of hardware description languages, such as Verilog and VHDL, chip capacity (in terms of the number of transistors per chip) follows Moore's law and chips are commonly produced at the time of this writing with over 70 million gates per device. Large gate counts and high operating frequencies allied with new chip architectures lead to considerable increases in processing power.

However, ensuring correct functional behavior of such large designs becomes more and more challenging. Simulation, emulation, and formal verification are three techniques available for validating a design.

Simulation is a predominantly used tool to validate a design in industry. Simulation can validate all possible behaviors of a design in a brute-force manner where input patterns are applied to the design and the resulting behavior is compared with expected behavior. Simulation allows some measure of functional and timing validation, and offers ease of use, relatively low cost, and sophisticated debugging. It can also handle very large circuits. However, rapidly evolving markets demand short design cycles while the increasing complexity of a design necessarily dictates that simulation coverage is less and less complete. A design with n inputs has 2^n possible input vectors, which is clearly too complex for using simulation for the purposes of verification. This exponential growth of simulation requirements results in significant simulation times making simulation an impractical approach even for the validation of the absence of specific design errors.

Emulation made its appearance to accelerate simulation by hardware in the 1980's. Instead of simulating a software model of the design, a designer could run the stimuli on a hardware prototype of the design. Programmable logic, such as *Field Programmable Gate Arrays* (FPGAs), enabled emulation by programming/implementing any arbitrary logic design into these devices. Although emulation provides two to four orders of magnitude speedup over software simulation on average, it comes at a cost. Fitting the design under validation into FPGAs is a complex task. Even with the speed benefits, emulation retains the important limiting characteristic of simulation resulting in the validation of only a small fraction of the possible input stimuli. Therefore, it is very hard for simulation/emulation to find the so-called corner-case bugs.

Formal verification validates the correctness of the implementation of a design with respect to its specification by applying mathematical proofs. Hence, formal verification conducts an exhaustive exploration of all possible behavior implicitly instead of explicit enumeration of an exponential number of possible behaviors that simulation and emulation require. Formal verification has attracted a lot of attention in the past few years. The most successful methods to date are equivalence checking [HC98], model checking [CE81], theorem proving [Hoa69], and symbolic trajectory evaluation [HS97].

Equivalence checking methods have led to significant success in industry. Correctness is defined as the functional equivalence of two designs (i.e. a gate-level design matches its desired behavior as specified at the *Register Transfer Language* (RTL) level). Because of the computational complexity of formal equivalence checking, a design methodology typically adopts specific rules to make the problem tractable for large designs. In practice, the specification and implementation of a design often have a large degree of structural similarity in terms of internal nets that implement the same function. For example, equivalence checking can check if the designs have corresponding latches. Once the correspondence between latches of a reference design and an implementation has been discovered, equivalence checking is just a matter of showing that corresponding latches have the same next-state function. This has proven to be very valuable in validating that an implemented gate-level design matches its desired behavior as specified at the RTL.

Another promising direction is model checking. The focus of model checking is to check whether the design, either in its specification or its implementation form, satisfies certain properties. Unlike equivalence checking, which limits itself to one step in time, model checking considers all time steps in the future. For example, checking that a design never deadlocks, or that each request is eventually responded to, requires the consideration of sequences having an unbounded length. Recently, model checking is beginning to be accepted by industry for integration into the verification flow. It is typically used in establishing the correctness of interfaces between components, as well as in uncovering difficult-to-find corner case bugs in designs [PM04].

Three leading *Electrical Design Automation* (EDA) vendors, Cadence, Synopsys and, Mentor Graphics, offer equivalence checking tools as a mature technology. They started to market their model checking tools recently. There are a number of smaller companies specializing in formal tools, all established within the last several years. Typically these companies specialize in property verification. Lar [Phi01] conducted a survey on commercial tools in equivalence checking and model checking in 2001. Tables 1.1 and 1.2 describe commercial equivalence checking tools and property checking tools. Some information is extracted from [Phi01] and is updated in the following tables.

Table 1.1 Comparison of Equivalence Checking Tools

Basic	Product	Formality	Encounter Conformal	Formal Pro
	Manufacturer	Synopsys	Cadnece	Mentor Graphics
	Website	www.synopsys.com	www.cadence.com	www.mentor.com
Data Structure	BDD	*	*	*
	SAT	*	*	*
	Symbolic methods		*	
	ATPG	*	*	*

Table 1.2 Comparison of Model Checking Tools

Basic	Product	Verifier	Design Verity-Check	imPROVE-HDL	Solidify	Verix
	Manufacturer	HDL	Veritable	TransEDA	Averant	RealIntent
	Website	www.athdl.com	www.veritable.com	www.transeda.com	www.averant.com	www.realintent.com
Data Structure	Design decomposition	*	*	*	*	*
	Property decomposition		*		*	*
	Abstraction	*	*		*	*
Property Specification	Language name	Verilog		PEC	HPL	
	Language style	Verilog	Forms	Keywords	Verilog	Keywords
	Inline	*			*	*
	Separate from design	*	*	*	*	
	Simulatable	*				*

In the above introduction, three techniques for the validation of digital designs are mentioned. The first part of this research is focused on formal methods where the existing methods are extended to improve efficiency. In the second part of the research we will concentrate on building the *Integrated Design Validation* (IDV) platform that combines formal verification methods and simulation techniques to provide a reliable environment for the validation of digital designs.

The remainder of this dissertation is organized as follows: Chapter 2 provides details regarding the fundamental data structures and algorithms used in equivalence and model checking, such as Boolean *Satisfiability* (SAT), and function representations such as B*inary Decision Diagrams* (BDDs). Chapter 3 reviews existing techniques for equivalence checking and model checking first, then two approaches are presented to leverage the memory usage and run-time for image computation resulting in a core algorithm for both equivalence checking and model checking. Chapter 4 describes the IDV system that combines formal verification methods and simulation techniques to provide a reliable environment for the validation of some types of designs. In Chapter 5, two designs are validated using the IDV system that can not be validated by any single tool. We conclude and discuss future work in Chapter 6.

CHAPTER 2

BACKGROUND

In this chapter, basic data structures and algorithms, such as Boolean functions, *Binary Decision Diagrams* (BDDs), and the Boolean *Satisfiability* Problem (SAT), are introduced.

2.1 Fundamental Data Structures and Algorithms

2.1.1 Boolean Functions and Finite State Machines

A Boolean function with *n*-inputs and one output is a mapping, $f : B^n \rightarrow B$ where $B = \{0,1\}$. The support of a Boolean function *f*, denoted as *supp*(*f*), is defined as the set of variables on which the function *f* depends.

Multilevel circuits are typically represented as a Boolean network. A *Directed Acyclic Graph* (DAG) whose nodes represent Boolean functions is defined in the following paragraph.

A Boolean network *N* with *n* primary inputs $X = \{x_1, ..., x_n\}$ and *m* primary outputs $Z = \{z_1, ..., z_m\}$ can be viewed as a set of *m* single output Boolean functions defined as $f: B^n \rightarrow B^m$. Let λ be $\{\lambda_1(X), ..., \lambda_m(X)\}$ where λ_i is an output function. A characteristic function of a Boolean network *N* is defined as a Boolean function $C(X, Z, \lambda)$ such that

C(X , Z , λ) = 1 $\Leftrightarrow z_i \equiv \lambda_i(X)$. In other words, a characteristic function maps every valid input/output combination to ‘1’, and every invalid combination to ‘0’. Computationally, the characteristic function can be derived by the following formula [HC98]:

$$\mathrm{C}(X, Z, \lambda) = \prod_{i=1}^{n} C_i(z_i, X, \lambda_i) = \prod_{i=1}^{n} (z_i \equiv \lambda_i(X))$$

where ($a \equiv b$) corresponds to $(ab + \overline{a}\overline{b})$, C$_{\mathrm{i}}$($X$, z_i , λ_i) is also called a “bit function”.

A synchronous sequential circuit or machine can be represented as an *Finite State Machine* (FSM). An FSM is a quintuple, $M = \{S, X, Y, \lambda, \delta\}$, where X denotes input wires, Y denotes output wires, S is a set of states, δ is the next state function, and λ is the output function. The next state function is a completely-specified function with domain ($X \times S$) and range S . A Huffman model [HC98] is shown in Figure 2.1.

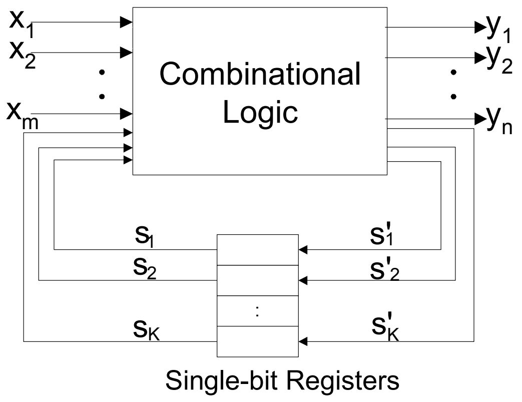


Figure 2.1 Huffman Model

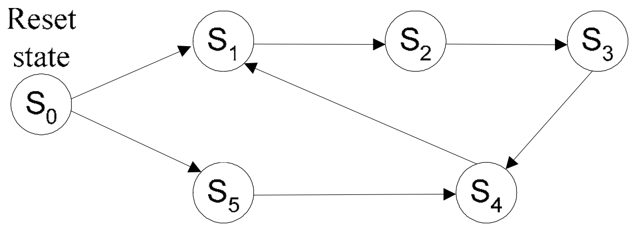
State space traversal is the basic procedure for equivalence and model checking. State space traversal can be performed explicitly by traversing the *State Transition Graph*

(STG) in either a depth-first or a breadth-first manner. Figure 2.2 illustrates the breadth-first strategy for FSM traversal.

```
FSM_traveral()
{
 // continue loop until fixed point
 $R_i = R_0$;
 while($R_{i+1}$!=$R_i$){
    // Breadth-first search
    $N_{i+1}$=Breadth_First_Search($R_i$);
    $R_{i+1}$ =$N_{i+1} \cup R_i$ ;
}
```

Figure 2.2 FSM Traversal Using Breadth-First Search

R_i represents the set of all reachable states at the i^{th} iteration. The iteration procedure begins at a reset state R_0 and stops at a *fixed-point* [Tar55] where the reachable states in two consecutive iterations are identical, i.e. $R_{i+1} = R_i$. At each iteration the next set of reachable states of R_i is computed and denoted as N_{i+1}. The set of reachable states in iteration i+1 will be $R_{i+1} = R_i + N_{i+1}$. An example of such a traversal is shown in Figure 2.3. The first graph shows the STG while the second one shows the breadth-first search process.



Iteration	Reachable states
0	$\{S_0\}$
1	$\{S_0, \boldsymbol{S_1}, \boldsymbol{S_5},\}$
2	$\{S_0, S_1, \boldsymbol{S_2}, \boldsymbol{S_4}, S_5\}$
3	$\{S_0, S_1, S_2, \boldsymbol{S_3}, S_4, S_5\}$
4	$\{S_0, S_1, S_2, S_3, S_4, S_5\}$

Figure 2.3 Example for FSM Traversal

The explicit method of traversal is simple but impractical for large digital designs since the STG will quickly exceed memory capacity. Thus, a symbolic, implicit state enumeration process is desired.

2.1.2 Image Computation Using the Transition Relation

Given an FSM and its characteristic function represented as a Boolean function $TR(S, X, S')$, the following formulation is possible. Variable sets $S = s_1, ..., s_n$, $S' = s'_1, ..., s'_n$, and $X = x_1, ..., x_n$ are the current state, next state, and input variables respectively. In sequential circuit designs, characteristic functions can be represented as a transition relation, this transition relations will be used in the remainder of this document. For a deterministic circuit, each binary memory element of the circuit under consideration gives rise to yet another term of the transition relation. When the circuit is synchronous, the partitioning is conjunctive and it can be written as the product of bit relations. In this work, it is assumed that the transition relation is given as a product of the bit relations TR_i s.

$$TR(S,X,S') = \prod_{i=1}^{n} TR_i(S_i,X,S'_i) = \prod_{i=1}^{n} (S'_i \equiv \delta(X,S_i))$$

The transition relation uniquely represents the sequence of states the machine will sequence through in response to a set of present states and input assignments. In the process of state space traversal, it is only interested in knowing *if there exists a transition that brings the machine from state p to state q*, while the specific input vector required to exercise a particular transition is not of interest. The smoothed transition relation is computed by *smoothing* (existentially quantifying out) every primary input variable from a transition relation. This operation is defined as follows:

Let $f(x_0,x_1,...,x_n)$ be a Boolean function. Then the functions f_{x_i} and $f_{\overline{x}_i}$ are referred to as the positive and negative cofactors, respectively, of function f with respect to x_i.

$$f_{x_i} = f(x_0,x_1,...,x_i = 1,...,x_n)$$

$$f_{\overline{x}_i} = f(x_0,x_1,...,x_i = 0,...,x_n)$$

The existential quantification of f with respect to the variable x_i is defined as

$$(\exists x_i)f = f_{x_i} + f_{\overline{x}_i}$$

The existential quantification of f with respect to a set of variables, e.g., $X = \{x_1, x_2, ...x_m\}$ is defined as a sequence of single variable smoothing operations.

$$(\exists X)f = \exists x_1 (\exists x_2 ... (\exists x_m f))$$

The transition relation defines a many-to-many projection from the present state space to the next state space as shown Figure 2.4.

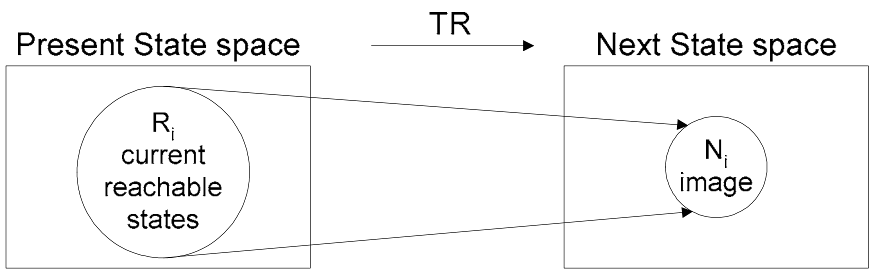


Figure 2.4 TR Projection

Based on the projection, the next reachable states of R_i can be computed and denoted as N_{i+1}, which is also referred to as the *image* of R_i. The procedure of the computation is referred to as an *image computation* and represented as

$$N_{i+1} = \exists S . \exists X . \left(R_i \wedge TR_i(S_i, X, S_i') \right)$$

Similarly, given a set of next states, N_i, the *pre-image* of N_i is the set of its predecessor states (denoted as R_{i-1}) and is computed by

$$R_{i-1} = \exists S' \exists X . \left(N_i \wedge TR_i(S_i, X, S_i') \right)$$

Image (pre-image) computation is a core technique for many equivalence and model checking algorithms.

2.1.3 Symbolic FSM State Space Traversal

A *fixed point* of a function τ is any p such that $\tau(p) = p$. A function τ is *monotonic* when $p \subseteq q$ implies $\tau(p) \subseteq \tau(q)$. Tarski [Tar55] showed that a monotonic function has a least fixed point, which is the intersection of all the fixed points. It also has a greatest fixed point, which is the union of all the fixed points. Figure 2.5 shows the procedure to compute the least (greatest) fixed point of τ:

```
Least (or Greatest) fixed point()
{
  //initialization
  let $Y$ = False; (or $Y$=True;)
  // continue loop until fixed point
  do
      let $Y'$ = $Y$; $Y$ = $\cup(Y)$;
  until $Y'$ = $Y$;
  return $Y$;
}
```

Figure 2.5 Least (greatest) Fixed Point Computation

Based on the transition relation and least fixed point computation, an implicit state enumeration (often referred to as *symbolic FSM traversal*) can be described as given in Figure 2.6.

```
FSM_traversal()
{
 // continue loop until fixed point
 $R_i = R_0$;
 $R_{i+1} = \cup$ ;
 //least fixed point computation
 while($R_{i+1}$!=$R_i$){
    // image computation
         $\bigcup$                              ;
    $R_{i+1} = N_{i+1} \cup R_i$ ;
}
```

Figure 2.6 Symbolic FSM Traversal

The procedure in Figure 2.6 provides the detailed procedure for an image computation of a FSM. Representing transition relations and reachable states will be addressed in the next section.

2.1.4 Binary Decision Diagrams

Binary Decision Diagram (BDD) is data structures used to represent Boolean functions. The concept of BDDs was first proposed by Lee [Lee59] in 1959. The idea was then developed into a useful data structure for Boolean function representation by Akers [Ake78] and subsequently refined by Bryant [Bry86], who introduced the concept of *Reduced, Ordered BDDs* (ROBDDs) along with a set of efficient operators for their manipulation and proved the canonicity property of ROBDDs.

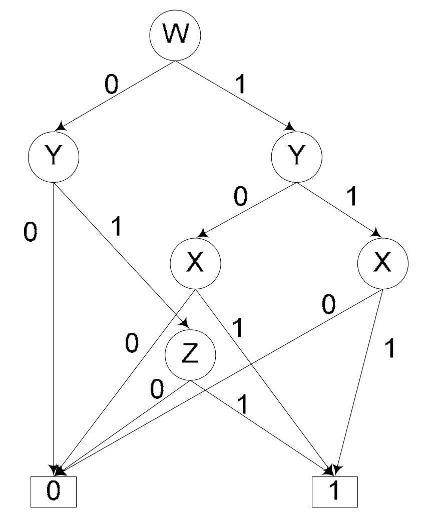
A BDD is a rooted, directed, acyclic graph. There are two types of nodes in the graph: terminal and non-terminal nodes. The terminal node is labeled with either a constant 0 or constant 1 and has no outgoing edges. Each non-terminal node is labeled with one binary variable (for example x_i) and has two outgoing edges, T (Then) and E (Else). Here T and E edges are connected to the positive (then) and negative (else) cofactors, respectively, of function f with respect to the binary variable (x_i). Thus, BDD nodes represent the Boolean function f according to the Shannon expansion theorem:

$$f(x_0, x_1, ..., x_n) = (x_i \wedge f_{x_i}) \vee (\overline{x}_i \wedge f_{\overline{x}_i})$$

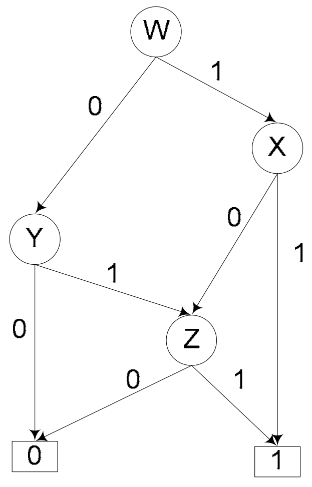
In a ROBDD, no sub-graph is isomorphic to another. Also, all variables appear in the same order in every path. This allows for a canonical representation of Boolean functions. The order of the variables can have a big impact on the size of the BDDs. Some functions exist, e.g. adder, whose sizes vary from linear to exponential for different variable orders. There are also some functions, e.g. multiplier, whose sizes are exponential for any variable orders. The complexity of finding an optimal order is *NP-hard* [BW96].

The size of a BDD can be further reduced by introducing complement edges, [Ake78], [BRB90]. Basically, a complement edge (c-edge), points to the complementary form of the function (BDD node). To maintain canonicity, it is assumed that a complement edge can only be assigned to the 0-edge. In the rest of the paper, BDD refers to a ROBDD.

A graphical example of a BDD with different orders for the Boolean function $f = wx + \overline{w}yz + w\overline{x}z$ is shown in Figure 2.7, while *a* with the order $\pi_0(w, y, x, z)$ and *b* with the order $\pi_1(w, x, y, z)$. From Figure 2.7, you can see the importance of variable ordering for a BDD.



a



b

Figure 2.7 BDD Representation

It is impractical to build a monolithic characteristic function BDD for an entire Boolean network for designs that the number of state bits exceeds a few hundred or designs contains some functions, i.e. multiplier, whose sizes are exponential with any given order. Functional decomposition is an important strategy to reduce the size of BDDs.

2.1.5 The Boolean Satisfiability Problem

The *Satisfiability* (SAT) problem, deciding whether a given Boolean formula is satisfiable, is one of the well-known *NP*-complete problems. Recently, modern SAT solvers, like zChaff [Mal$^+$web], Grasp [Mar$^+$web] and Berkmin [GN02], have demonstrated tremendous success. The key elements in modern SAT solvers are non-chronological backtracking, efficient conflict driven learning of conflict clauses, and improved decision heuristics.

A Boolean formula for a SAT solver is typically specified in *Conjunctive Normal Form* (CNF) which consists of a set of clauses. Each clause is a set of literals, and each literal is an instance of a variable or its complement. A clause represents the disjunction of a set of literals.

The basic framework for modern SAT solvers is based on the *Davis-Putnam-Longeman-Loveland* (DPLL) backtracking search [DLL62], shown in Figure 2.8. The function **decide_next_branch()** chooses the branching variable at the current decision level. The function **deduce()** accomplishes *Boolean Constraint Propagation* (BCP) to deduce further assignments. BCP checks if a partial assignment leaves any clause with only one unassigned literal and all other literals with value 0, then for that clause to be true, the last literal must be assigned the value 1. This is also referred to as implication. In the process, it might infer that partial assignments to variables do not lead to any satisfying solutions. This is referred to as a conflict. In the case of a conflict, new clauses are created by **analyze_conflict()** that is used to prevent entering the same unsuccessful search space in the future. After a conflict is detected, the SAT solver backtracks to the variable that causes the conflict. This variable may not be the most recent variable decided, referred to as a non-chronological backtrack. If all variables have been decided, then a satisfying assignment is found and the procedure returns. The strength of various SAT solvers lies in their implementation of BCP, non-chronological backtracking, decision heuristics, and learning.

```
while(true) {
  if (decide_next_branch()) { // Branching
    while(deduce()==conflict){//Propagate implications
      blevel = analyse_conflict(); // Learning
      if (blevel == 0)
         return UNSAT;
      else
        backtrack(blevel);//Non-chronological backtrack
            }
   }
  else // no branch means all vars have been assigned
     return SAT;
}
```

Figure 2.8 Basic SAT Procedure

2.2 Existing Techniques for Verification

This section describes existing techniques for equivalence and model checking of digital circuits.

2.2.1 Equivalence Checking

Two designs are functionally equivalent if they produce identical output sequences for all valid input sequences. Combinational circuit equivalence checking is more mature than sequential circuits equivalence checking.

There are three basic approaches to combinational equivalence checking. *Structural methods* search for a counter-example that produces a "1" at the output of MITER and are usually implemented using SAT solvers. MITER is a circuit that is comprised of two circuits being compared as shown in Figure 2.9. All common inputs are tied together and all common outputs combined through XOR gates. The output of MITER is one when two circuits are not equivalent. Similarly, *random simulation* is used to find a counter-example by random search. *Functional methods* [AK95] are based on a canonical function representation for which structural equivalence implies functional equivalence. BDDs are widely used for functional methods. The advantage of functional methods is their independence with respect to the circuit structure.

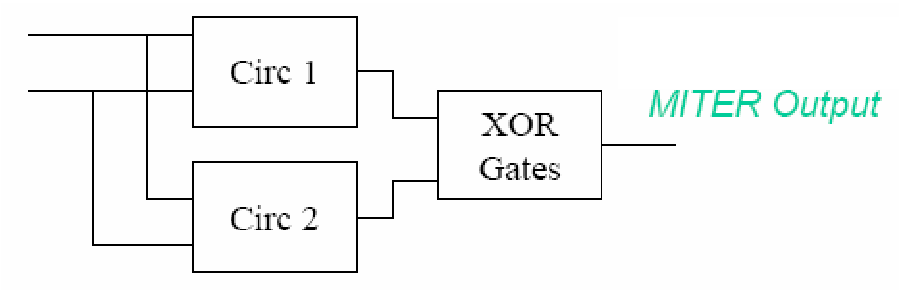


Figure 2.9 MITER

Structural SAT solvers can be used directly for equivalence checking [Rot77]. However, their direct application for a MITER would require an exponential number of backtracks and is therefore impractical. A more practical approach is to exploit structural similarities that are based on internal equivalence points, or cutpoints, which are used to decompose the equivalence checking problem into small pieces [DO76][Ber81][BT89]. These methods are based on the observation that in many equivalence checking cases, one of the machines under comparison contains a large number of internal nets that have a functionally equivalent counterpart in the other machine.

General methods for sequential equivalence require the reachable states of both designs, modeled as FSMs, are computed simultaneously to demonstrate the absence of state pairs with different outputs. A product machine for comparing two FSMs can be built for such a purpose as shown in Figure 2.10, which is also referred to as a MITER in [HC98]. M_1 and M_2 are functionally equivalent, if, and only if, the output of the product machine produces a "0" for all of the reachable states and input vectors. Thus, it is required to systematically explore the state space of the product machine, also referred to as state space traversal. However, performing such a traversal is computationally expensive and becomes intractable if the number of state bits exceeds a few hundred. This is known as the state space explosion problem. There are two basic techniques for symbolic state space traversals. The transition function method [CBM89] is based on a successive traversal of all next-state functions to determine the resulting states for each input. The transition relation method [BCL$^+$90] uses a characteristic function to represent all valid state transitions of the product machine.

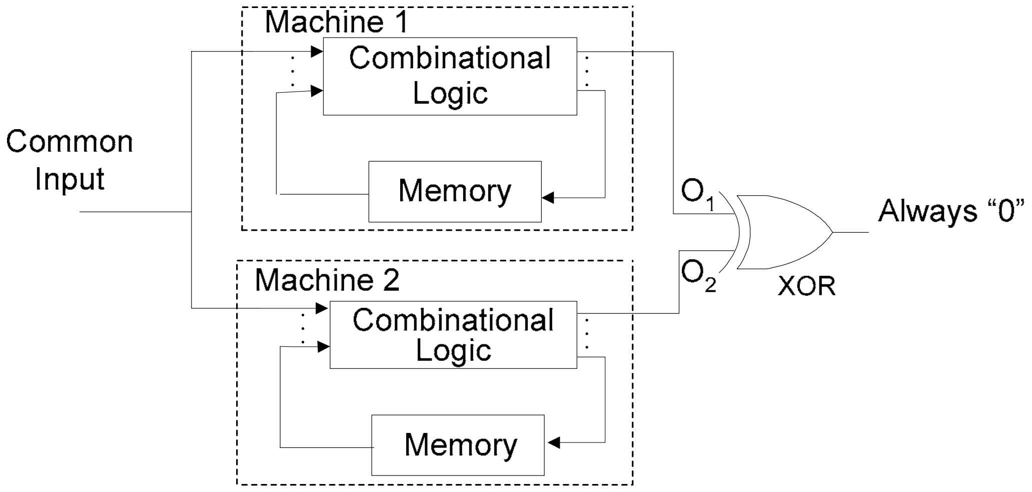


Figure 2.10 Product Machine for Comparing Two FSMs

2.2.2 Model Checking

Model checking verifies whether the implementation of a design satisfies properties that are specified in temporal logic. The properties are classified in general as safety and liveness properties. A safety property asserts that nothing bad will happen in the design. Examples of safety properties are mutual exclusion (no two processes are in the critical section simultaneously) and deadlock free (no deadlock state is reached). A liveness property asserts that eventually something ‘good’ happens in the design. An example of a liveness property is that a design is starvation free (eventually service will be granted to a waiting process).

The requirements of model checking are a model of the system, a temporal logic framework, and a model checking procedure. FSMs are appropriate models for representing sequential designs and are widely used to model a system. Temporal logic systems are mainly classified as linear-time logics or branching-time logics. In *Linear-time Temporal Logic* (referred to as LTL), events are described along a single computation path. In branching-time temporal logic, the temporal operators are quantified over the paths that are possible from a given set of states. Each type of logic has its advantages and disadvantages and different expressive powers. C*omputational Tree Logic* (CTL) is the most commonly used temporal logic in model checking. The process of model checking consists of computing the set of states that satisfies a given property in the design and comparing the satisfying states to the initial states of the design.

CTL was first proposed by Clark and Emerson as a branching-time temporal logic [CE81]. CTL formulae are composed of path quantifiers and temporal operators. The path quantifiers are used to describe the branching structure in the computation tree. There are two path quantifiers:

- **A** – for all paths,
- **E** – there exists a path or for some paths.

There are four basic temporal operators in CTL:

- **X** – next time,
- **F** – eventually or in the future,
- **G** – Always or Globally,
- **U** – until.

In CTL, every quantifier is followed by a temporal operator. Therefore, there are eight basic CTL operators:

- **AX** and **EX**
- **AF** and **EF**
- **AG** and **EG**
- **AU** and **EU**

The path quantifier and the temporal operators have the following relations:

- $\mathbf{F}\ \phi \equiv \text{True}\ \mathbf{U}\ \phi$
- $\mathbf{G}\ \phi \equiv \neg \mathbf{F}\ \neg\phi$
- $\mathbf{A}\ \phi \equiv \neg \mathbf{E}\ \neg\phi$

Then, using these relations, each of the eight CTL operators can be expressed in terms of only three operators that are EX, EG, EU. The satisfying set of states for the three operators can be computed by the following fixpoint computation.

- $\mathbf{EX}\ \phi = \text{Pre\_Image}(\mathbf{T}, \phi)$
- $\mathbf{EG}\ \phi = \nu \mathbf{Z}. \phi \wedge \mathbf{EX}(Z)$
- $\mathbf{E}[\phi\ \mathbf{U}\ \psi] = \mu \mathbf{Z}.\ \psi \vee (\phi \wedge \mathbf{EX}(Z))$

Where Pre_Image $(\mathbf{T}, \phi)$ is a pre-image computation finding all predecessors of the states ϕ in one step; μ and ν are least and greatest fixpoint operators respectively. Notice that the pre-image computation is the basic and key operation in model checking, and model checking is performed by a series of pre-image computations. Thus, the main challenge of model checking is the state space explosion problem, the same as equivalence checking.

There have been many approaches to alleviate the state space explosion problem in model checking, such as abstraction and refinement [LPJ+96][JMH00], Bounded Model Checking [BCCZ99], Symmetry reduction [CEJS98][MHB98], and partial-order reduction[GW94][ABH+97].

In next chapter, the major approaches to leverage state space explosion in image computation are reviewed. Then, two new approaches are presented: one is based on genetic algorithms while the other one combines SAT and BDD methods.

CHAPTER 3

NEW APPROACHES FOR IMAGE COMPUTATION

3.1 Related work

Though equivalence and model checking are quite successful, we still have the state space explosion problem to contend with. Most of the approaches mentioned before try to improve algorithms for model checking and equivalence checking. However, image or pre-image computation is the core operation for both equivalence and model checking. Therefore, efficient algorithms for image and pre-image computation are going to benefit most approaches. Image or pre-image computation work can be classified into three categories based on the data structure they use: BDD-based, SAT-based, and hybrid-based approaches. These three approaches are discussed in the following sections.

3.1.1 BDD-based Approach

As indicated before, a transition relation defines a many-to-many projection from the present-state space to the next-state space. Based on this projection, the image of the current reachable states R can be computed. If the transition relation is given in a normal conjunctive decomposed form, the following equation can be used for image computation.

$$N = Img(R) = \exists S.\exists X.\left(R \wedge \prod_{i=1}^{l} TR'_i(S, X, S') \right)$$

Another benefit of image computation on a decomposed transition relation is progressive existential quantification. Let Q denote the variables to be quantified, which is $Q = X \cup S$ and Q_i denotes the set of variables that do not appear in $TR'_1, ..., TR'_{i-1}$. The image computation can be performed as follows:

$$Img(R) = \exists Q_1.(TR'_1 \wedge \exists Q_2.(TR'_2 \cdots \exists Q_l.(TR'_l \wedge R)))$$

The size of intermediate BDDs and the effectiveness of early quantification depend heavily on the order in which BDDs are conjoined in the above equation. For example, consider a 3-bit counter with present state variables s_1, s_2, s_3 and next state variables s'_1, s'_2, s'_3, where s_3 and s'_3 are the most significant bits. Figure 3.1 shows the state diagram of a 3-bit counter.

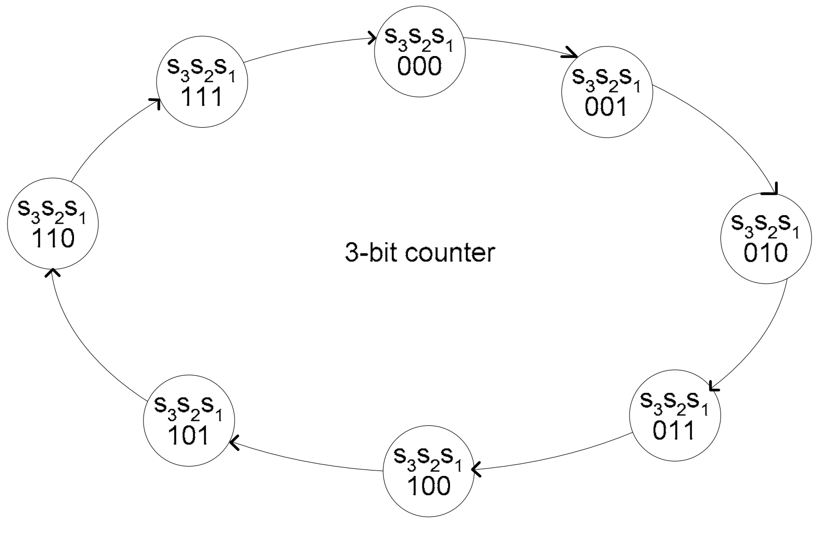


Figure 3.1 State Diagram of 3-bit Counter

The transition relation of the counter can be expressed as:

$$TR(S,S') = TR_1 \wedge TR_2 \wedge TR_3$$

where: $TR_1 = (s_1' \equiv \overline{s_1})$, $TR_2 = (s_2' \equiv s_1 \oplus s_2)$ and $TR_3 = (s_3' \equiv (s_1 \wedge s_2) \oplus s_3)$.

For the order as given in $\pi_1(TR_1, TR_2, TR_3)$, the image computation can be carried out as

$$Img(R) = \exists s_1.\{TR_1 \wedge \exists s_2.[TR_2 \wedge \exists s_3.(TR_3 \wedge R)]\}$$

Alternatively, for the order $\pi_2(TR_3, TR_2, TR_1)$, the image computation is

$$Img(R) = \exists s_1 s_2 s_3.\{TR_3 \wedge [TR_2 \wedge (TR_1 \wedge R)]\}$$

It is seen that for order π_1, present state variables can be quantified out in the order of s_3, s_2, s_1. While for order π_2, no variables can be smoothed out in the intermediate computation. Therefore, order π_1 is better than order π_2 for image computation. So the image computation relies on a good decomposition of the characteristic function BDDs and a good order for the clustered terms. Finding such an order is referred to as the "quantification schedule problem".

The importance of the quantification schedule was first recognized by Burch et al. [BCL91] and Touati et al. [TSL+90]. Geist et al. [GB94] proposed a simple circuit independent heuristic algorithm, in which they ordered conjuncts by minimizing the maximal number of state variables of the intermediate BDDs in the process of performing the image computation. Ranjan et al. [RAP+95] proposed a successful heuristic procedure (known as IWLS95). The algorithm begins by first ordering the bit relations and then clustering them, and finally ordering the clusters again using the same heuristics. The order of relations is chosen using four normalized factors; the number of variables that will be quantified, the number of present state and primary input variables, the number of next state variables that would be introduced, and the maximum BDD index of a variable that can be quantified. After the ordering phase, the clusters are derived by repeatedly conjoining the bit relations until the size of the clustered BDD exceeds a given threshold, at which point a new cluster is started.

Bwolen Yang improved the IWLS95 heuristic in his thesis [Yan99] by introducing a pre-merging phase where bit relations are initially merged pair-wise based on the sharing of support variables and the maximum BDD size constraint. Moon et al. [MS00] presented an ordering algorithm (known as FMCAD00) based on computing the Bordered Block Triangular form of the dependence matrix. Their ordering algorithm minimizes the active lifetime of variables, α. Instead of clustering ordered bit relations in a sequential order, the bit relations are clustered according to the affinity between them. Affinity measures the sharing of the support variables.

Chauhan et al. [CCJ+01a] extended FMCAD2000 and used combinatorial algorithms to improve the performance (i.e. simulated annealing). They also argue in favor of using α. All these techniques are static techniques. Subsequently, the same clusters and ordering are used for all the image computations during symbolic analysis.

Chauhan et al. [CCJ+01b] also proposed a non-linear dynamic quantification scheduling method by viewing the image computation as a problem of constructing an optimal parse tree for the image set. Their "Basic" algorithm is as follows: a heuristic score is computed for each variable in a set of variables Q to be quantified. The variable with the lowest score, say q, is chosen and the two smallest BDDs in whose support set q appears are conjoined. The overall approach is a two-phase approach combining static and dynamic schemes. Before image computation, only as many primary input variables as possible are quantified out using the Basic algorithm. Then, for each image computation step, the remaining input and all present state variables are quantified out using the Basic algorithm.

H. Jin, et al. [JKS02] proposed a fine-grain conjunction scheduling algorithm in terms of a minimum max-cut linear arrangement. The cut whose width is minimized is related to the number of variables active during image computation.

3.1.2 SAT based Methods

SAT is less vulnerable to memory explosion than BDDs. Recent improvements in SAT solvers have attracted a lot of attention in the use of SAT for image or pre-image computation.

McMillan proposed a pure SAT-based unbounded symbolic model checking algorithm in [Mcm02]. The reachable states are represented as CNF. The transition relation is represented as a CNF formula. He used a slightly modified SAT procedure to perform SAT-all. Whenever a satisfied assignment is found, a blocking clause is generated by redrawing the implication graph. The efficiency comes from the fact that a smaller blocking clause will be generated from redrawing the implication graph. The blocking clause is added to the CNF formula and a new search is started by backtracking until all solutions are found.

In [KP03], Kang *et al*. also proposed a SAT based image computation algorithm for unbounded model checking. They use the *Disjunctive Normal Form* (DNF) to represent reachable states and a CNF formula for the transition relation. The blocking clause is added to get SAT-all. Unlike the method for generating the blocking clause from redrawing the implication graph in [Mcm02], the blocking clause is just the complement of the current satisfying assignment. At the end of each iteration, all frontier reachable states are minimized using ESPRESSO [Bra$^+$web]. They report their results based on safety properties for some benchmarks. It seems that no other improvement is incorporated other than using ESPRESSO which may also consume more time (the time of running ESPRESSO).

A problem with using SAT is that only one reachable state (minterm) is returned with each successful search. Chauhan [CCK03] tried to solve the problem by enlarging the satisfying assignment. After a satisfying assignment is found, it may contain input variables, intermediate variables, present-state variables and next-state variables. Since we only care about the present-state variables, all input, intermediate variables and some present-state variables can be seen as free-variables. The bit transition functions are analyzed to see which next-state variable can be set to a free-variable, based on the current set of free-variables. However, since the constraints for free next-state variables are quite strict, this method is not that efficient as seen by the experimental results [CCK03]. In this method, a transition relation is represented in CNF format and reachable states are represented in a DNF format.

3.1.3 Hybrid Approaches

Some approaches combine BDD, SAT and *Automatic Test Pattern Generation* (ATPG) techniques for image computations.

In [GYA01], Gupta et al., proposed a hybrid method for image computation. BDDs are used to represent current states and reachable states. The transition relation is represented in CNF. A SAT solver is deployed to perform a high-level decomposition of the search space and BDDs are used to compute all solutions below the intermediate points in the SAT decision tree. This approach is similar to partitioned BDDs where the SAT-solver is used to compute a disjunctive decomposition of the problem and the decomposed sub-problems are handled by BDDs. Thus, this method still suffers the same kind of memory problems associated with other BDD-based methods. Also, it is hard to predict the depth of a SAT decision to make sure that the resulting BDD sub-problem will not blow up in memory usage.

Sheng et al., [SH03], described another hybrid method that combines ATPG and SAT for one-step pre-image computation based on equivalence cut-sets. They found that many searches will lead to the same sub-space. By identifying the cutting set and a jump between identical cutting sets, they will never revisit the same sub-space that has been searched before. They named this learning technique as “success-driven learning”. Li extended the work by combining success-driven learning with traditional conflict learning in SAT in the same framework in [LHS04]. However, their procedure is not complete since it only provides a one-step pre-image computation and “an efficient procedure for multiple cycles is needed", as the authors pointed out in the conclusion of their work [SH03].

Parthasarathy, *et. al*. proposed an algorithm for image computation using sequential SAT [PIWC04]. The sequential SAT problem is to find a sequence of input vectors to the circuit, such that the value assignments are satisfied or, to prove that no such sequence exists. This approach actually tried to combine the key advantages of ATPG and SAT and it uses circuit structure information to derive a minimum cube of reachable states. They also implement a two-level minimization tool to reduce the number of cubes representing frontier-reached states. Another technique they use is similar to the blocking clauses used in [Mcm02], called "state-space bounding"; however, their improvement is limited.

3.1.4 Summary of Past Approaches for Image Computation

As described in the previous section, BDDs have been used extensively for image and pre-image computations. The key issues of BDD-based implementations of image (pre-image) computations include modeling a FSM as a characteristic function and representing it as a BDD. However, the size of the BDD is very sensitive to the order of the variable which often leads to memory explosion for some functions. Construction of a monolithic characteristic function BDD is typically impractical for circuits that the number of state bits exceeds a few hundred or circuits contains some functions, e.g. multiplier, whose sizes are exponential with any given order. Different variable ordering and reordering algorithms have been proposed to address this problem. Also, instead of building one single BDD, partitioned BDDs and conjunction scheduling may be deployed to alleviate the memory problem. Even with these approaches, the BDD-based approach still has memory explosion problems.

Another known method, which is less vulnerable to memory explosion, is the use of SAT solver. In recent years, several efficient SAT solvers have been developed, such as Chaff [Mal$^+$web], Grasp [Mar$^+$web] and Berkmin [GN02]. These SAT solvers employ conflict learning [MS96b] and non-chronological backtracking [MS96a] to speed up the search procedure. However, these SAT solvers are targeted to find a single solution (minterm). A minterm is a cube that contains every variable in its support set. Image computation requires capturing all satisfiable solutions. A naïve way of finding all satisfiable solutions is repeatedly calling the SAT solver after finding a solution. The solutions found previously are added as blocking clauses to prevent the SAT solver from finding the same solution again; however, the above method is very inefficient. There are two aspects to improve efficiency. One aspect is to narrow down the search space and the other is to find a solution that covers more than one minterm.

3.2 A Genetic Algorithm Approach for the BDD-based Method

Genetic Algorithms (GA) have been successfully used in the BDD reordering [DBG95] and approximate reachability analysis [TD01]. Genetic algorithms generally generate better results as compared to other methods but require longer runtimes. For the conjunctive scheduling problem, the order will be computed only once and better ordering can reduce the image computation time dramatically. Based on the above factors, a conjunctive scheduling approach based on genetic algorithms is developed.

A genetic algorithm emulates the metaphor of natural biological evolution to solve optimization problems. Genetic algorithms generally utilize the following steps. a) Initialize population: find a collection of potential solutions to the problem, also called current population. b) Create offspring: produce a new population through the application of genetic operations on selected members of the current generation. c) Evaluate fitness: evaluate the quality of the solution in the new generation. d) Apply selection: select solutions that will survive to become parents of the next generation based on their quality of solution to the problem. In this way, it is more likely that desirable characteristics are inherited by the offspring solutions. e) This cycle repeats until some threshold or stopping criterion is met.

The detailed description for the GA is given in the following sections.

3.2.1 GA Based Ordering Algorithm

3.2.1.1 Problem Representation and Initial Population

The GA starts with mapping a problem into a set of chromosome representations used within GA. Since we are interested in the order of functions and their support set, a preprocessing step converts the information into a chromosome. Considering the above 3-bit counter example, it is encoded as shown in Figure 3.2:

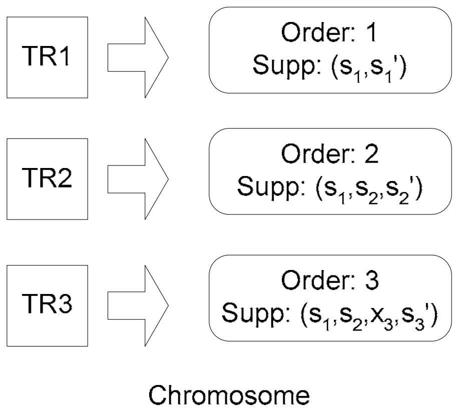


Figure 3.2 A Chromosome for 3-bit Counter

Any ordered set of functions could be a solution, so an initial population is generated by randomly mutating the order of the genes in the chromosome.

3.2.1.2 Fitness function

The fitness function discussed here is based on the dependency matrix of the chromosome. The *dependence matrix* defined in [MS00] is used for an ordered set of functions. The *dependence matrix* of a set of *m* single-output functions ($f_1,...,f_m$) depending on *n* variables $x_1,...,x_n$ is a matrix *D* with *m* rows (corresponding to *m* functions) and *n* columns (corresponding to *n* variables) such that $d_{i,j}=1$ if function f_i depends on variable x_j, and $d_{i,j}=0$ otherwise. The dependency matrix of a chromosome is defined in the same way. The dependency matrix of above chromosome is shown in Figure 3.3.

$$\begin{array}{c|ccc}
 & s_1 & s_2 & s_3 \\
\hline
TR_1 & 1 & & \\
TR_2 & 1 & 1 & \\
TR_3 & 1 & 1 & 1
\end{array}$$

Figure 3.3 Dependency Matrix for a Chromosome

The size of a BDD depends on the number of variables and the functions it represents. Smaller BDDs usually can be produced by conjoining two product terms that have a similar support set because few new variables are introduced. Based on the above observation, the *normalized active lifetime* [MS00] of the variables in matrix D is given by

$$\alpha = \frac{\sum_{i=1}^{n}(h_j - l_j + 1)}{n \cdot m}$$

where $l_j(h_j)$ is the smallest (largest) index i in column j such that $d_{i,j} = 1$ respectively.

$h_j - l_j$ gives a quantity measure on sharing the variable in column j stays. The normalized active lifetime measures how closely the product terms stay based on their support variables. The objective of ordering becomes to lower the normalized average active lifetime for a given matrix by manipulating the order of columns.

Because the objective of ordering is to minimize α, α is used as the fitness function.

3.2.1.3 Selection

The selection is performed by linear ranking selection (i.e., the probability that one element chosen is proportional to its fitness). The size of the population is constant after each generation. Additionally, some of the best elements of the old population are inherited in the new generation. This strategy guarantees that the best element never gets lost and a fast convergence is obtained. Genetic algorithm practice has shown that this method is usually advantageous [Dr98].

3.2.1.4 Genetic Operators

Two genetic operators are used in the algorithm: *Partially Matched Crossover* (PMX) as first described in [GL85] and a random *Mutation* (MUT).

PMX generates two children from two parents. The parents are selected by the method described above. The operator chooses two cut positions at random. Note that a simple exchange of the parts between the cut positions would often produce invalid solutions. A validation procedure has to be executed after exchange. The detailed procedure for PMX follows.

The children are constructed by choosing the part between the cut positions from one parent and preserving the position and order of as many variables as possible from the second parent. For example, $p_1 = \pi(1,2,3,4,5)$ and $p_2 = \pi(3,2,4,1,5)$ are the parents while $i_1 = 2$ and $i_2 = 4$ are the two cut positions. The resulting children before the application of the validation procedure are $c_1' = \pi(1,2,4,1,5)$ and $c_2' = \pi(3,2,3,4,5)$. The validation procedure goes through the elements between the cut positions and restores the ordering. This results in the two valid children $c_1 = \pi(1,2,4,3,5)$ and $c_2 = \pi(3,2,1,4,5)$. This procedure is shown in Figure 3.4.

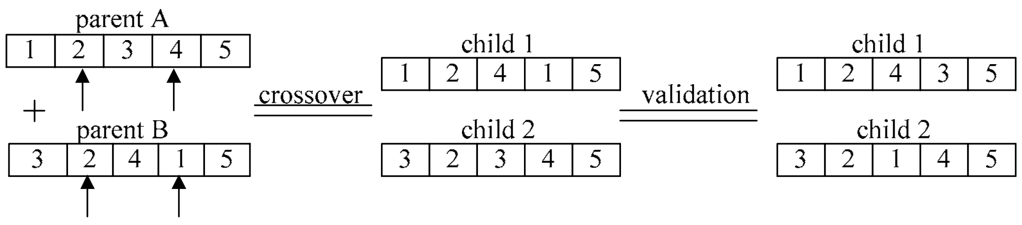


Figure 3.4 PMX

MUT selects a parent by the method described above and randomly chooses two positions. Two genes at these two positions are exchanged, like Figure 3.5 shows.

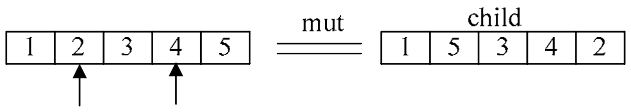


Figure 3.5 MUT

3.2.1.5 Algorithm

Our genetic algorithm is outlined as follows:

1. The initial population is generated using the original order as the first individual and by applying MUT to create more elements.
2. Genetic operators are selected randomly according to a given probability. The selected operator is applied to the selected parent (MUT) or parents (PMX). The better half of the population is inherited in each iteration without modification.
3. The new generation is updated according to their fitness.

4. The algorithm stops if no improvement is obtained for 50 iterations.

The genetic algorithm routine is shown in Figure 3.6.

```
Genetic algorithm(){
    Generate_initial_population;
    Update_population;
    do{
        for( each child $i$ ){
            $j$ =linear_ranking_selection();
            randomly_select_method;
            case MUT: child( $i$ ) = MUT(parent $j$ );
            case PMX: $k$ = linear_ranking_selection();
            child( $i$ , $i+1$ ) = MUT(parent $j$ , $k$ );
        }
    }
}
```

Figure 3.6 Genetic Algorithm Routine

3.2.2 Affinity Based Clustering Algorithm

The ordering algorithm described above rearranges product terms so that product terms sharing more variables stay as closely together as possible. The next step is clustering some of the small product terms into a single big one while the BBD size of the clustered product terms is within a reasonable threshold. The motivation for clustering is to reduce iterations and improve efficiency in computation.

One naive way of clustering is sequential clustering. Starting from an ordered list of product terms obtained from the ordering step, one continuously merges product terms sequentially until a given threshold is reached. The merged product terms are set aside as the first element of a cluster. The process is then repeated on the remainder of the list [BCL91].

The sequential approach may lead to suboptimal results because the sharing of variables is not considered in the conjunction. The dependency matrix defines the similarity of support variables of an ordered set of functions. Affinity defines the similarity of support sets of two functions. Affinity is defined as the following [MS00]:

Let d_i be the *i*-th row of the dependency matrix. Let $|d_i|$ be the length (number of non-zero entries) of row vector d_i. Finally, let $d_i \times d_j$ designate the inner product of d_i and d_j. The *affinity*, β_{ij} of vector d_i and d_j is defined as:

$$\beta_{ij} = \frac{d_i \times d_j}{|d_i| + |d_j|}$$

The affinity based clustering algorithm is now discussed. The affinities for pairs of adjacent product terms are computed as above, and then the pair with the highest affinity is merged. As in the sequential approach, merging is accepted only if the resulting BDD size does not exceed the cluster threshold size. If the threshold is exceeded, a barrier is introduced between the two terms. The process is then recursively applied to the two subsets of the rows above and below the barrier. If the size of the conjunction BDD is below the threshold, the algorithm computes the affinity for the new function and its neighbors and then selects a new pair with the highest affinity. The terminal case of the recursion occurs when only one function is left.

3.2.3 Ordering for Image Computation

As indicated before, a good conjunctive decomposition offers a good starting point for image computation. Early quantification could be employed to reduce the size of the

BDD by quantifying away variables in its support set but not the support set of future image computation steps.

To get a good order for early quantification, we can use the same technique introduced before with a slight change on the fitness function.

3.2.3.1 Fitness function for image computation

A fitness function for image computation to find a good order for early quantification is needed. In [MS00] and [CCJ+01a], both argue in favor of using active lifetime, α. However, it is enough to just consider active lifetime. As an example, consider the 3-bit counter given before and two dependency matrices with orders of $\pi_1(TR_1, TR_2, TR_3)$ and $\pi_2(TR_3, TR_2, TR_1)$ as shown in Figure 3.7.

	s_1	s_2	s_3
TR_1	1		
TR_2	1	1	
TR_3	1	1	1

	s_1	s_2	s_3
TR_1	1	1	1
TR_2	1	1	
TR_3	1		

Figure 3.7 Dependency Matrices for Two Chromosomes

From the example, we can see that both orders have the same active lifetime, α_{π_1} = 2/3, α_{π_2} = 2/3. In the image computation, as we showed before, order π_1 is better than order π_2.

Based on above observations, another measure, named as *normalized total lifetime* [MS00], is defined as

$$\lambda = \frac{\sum_{i=1}^{n}(m - l_j + 1)}{n \cdot m}$$

The normalized total lifetime for the above two chromosomes are λ_{π_1} = 2/3, λ_{π_2} =1 respectively. Order π_1 is better than order π_2 because it has a smaller total lifetime. The total lifetime λ and active lifetime α are not independent. A better λ could also result in a better α.

An advantage of the GA algorithm is that we can minimize total lifetime α and active lifetime λ at the same time. The fitness function we use includes these two parameters, shown as follows:

$$C(\pi_i) = a_0\lambda + a_1\alpha$$

where π_i is a permutation of transition relations and a_0, a_1 are weights attached to two time parameters, $0 \le a_0, a_1 \le 1$ and $a_0 + a_1 = 1$. As indicated before, the total lifetime λ and active lifetime α are not independent, but they do have different impacts in the results. The total lifetime tries to pull all terms close to the bottom of the matrix while the active lifetime tries to pull all terms closely based on the support set. Thus, a tradeoff between the two parameters is needed based on a_0, a_1. Experiment result shows that $a_0 = a_1 = 0.5$ achieves the best results.

3.2.4 Experimental results

In order to evaluate the GA approach, we ran the conjunctive decomposition algorithm and then applied it to FSM traversal. The benchmarks are from the ISCAS'89 and LGSYNTH'91 suites. The algorithm is implemented using the CUDD BDD package [Som$^+$web]. All experiments are carried out on a 733MHz HP PC running cygwin under Windows XP with 192MB of main memory. The following figures show the dependency matrix before (Figure 3.8) and after (Figure 3.9) GA based ordering algorithm for the benchmark *mm9b*. The variables on the right side of Figure 3.8 form two triangles while only one triangle for the corresponding variables in Figure 3.9. With one triangle in Figure 3.9, all the variables at the bottom of triangle can be quantified out as the image computation carried on from bottom up. With two triangles, the variables at the lower triangle can not be quantified out until the upper triangle is reach. Thus, the order in Figure 3.9 will definitely produce a better result than the order given in Figure 3.8 in image computation.

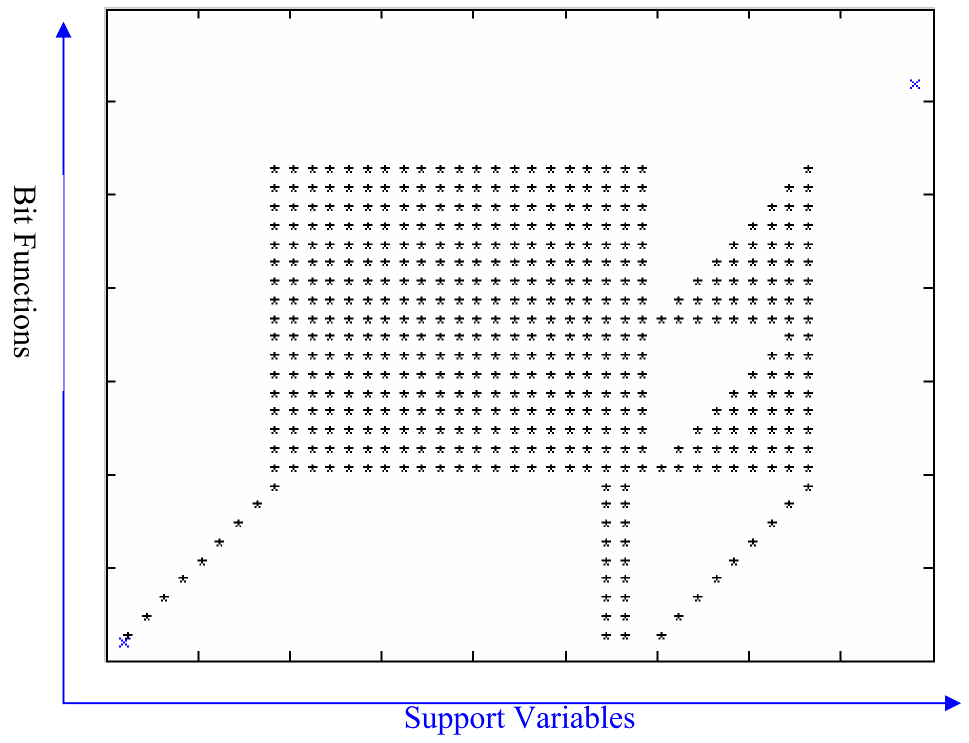


Figure 3.8 Dependency Matrix Before Ordering

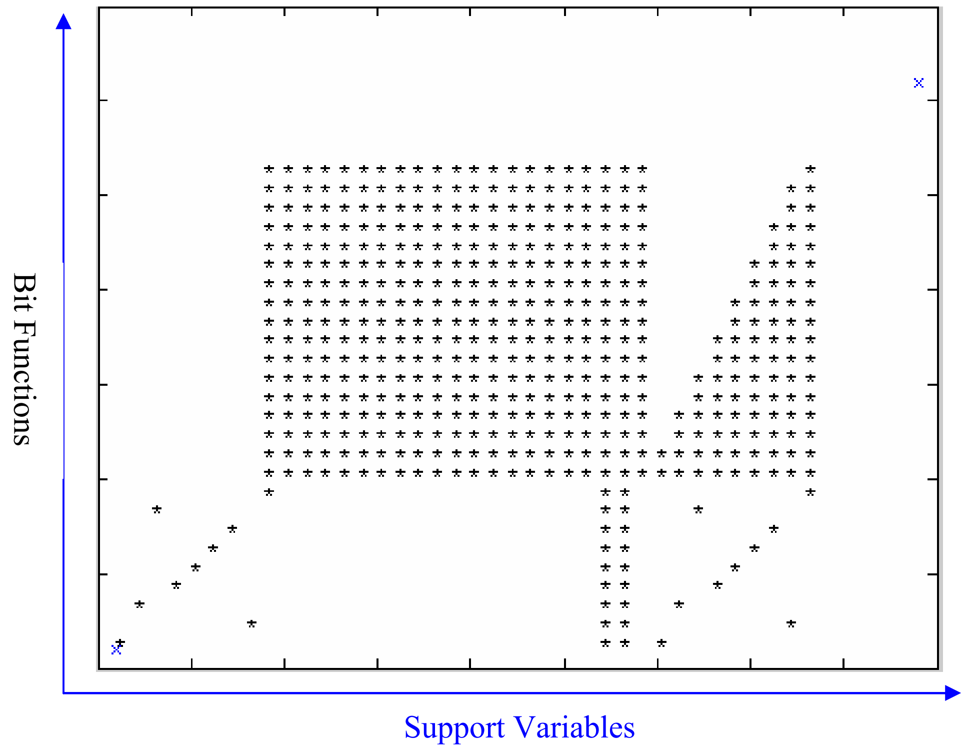


Figure 3.9 Dependency Matrix After Ordering

The GA approach is compared with the best known FMCAD00 approach. Dynamic BDD variable reordering is enabled in both approaches. A time limit of 7200 seconds is used. A threshold, 5000, is set to limit the number of nodes for each partitioned BDD. The two parameters measured are the number of clusters (in the column labeled "clusters") and the total number of BDD nodes (in the column labeled nodes). Shared nodes among various clusters only count once. Experimental results given in Table 3.1 shows that our approach improved the memory performance as compared with FMCAD00 in most test benchmarks.

Table 3.1 Genetic Result on Image Computation

Circuits	FMCAD00		GA		Improvement
	Time(s)	peak nodes(KB)	Time(s)	peak nodes(KB)	on nodes
sbc	4.3	12.9	10.5	16.9	-31%
clma	24.7	142.4	137	32.7	77%
clmb	30.6	141.6	137	32.7	76%
mm9a	2.8	29.4	5.6	10.2	65%
mm9b	55.4	702	5.57	17.1	97%
mm30a	21.8	268.3	113	106.4	60%
bigkey	88.8	34.7	1066	105.6	-204%
s420.1	19.7	0.673	11	0.642	-4%
s1512	641.6	121	371	65.9	45%
s1269	1267	1743	2669	2079	-19%
s4863	242.5	419.5	6996	1014	-141%
s3271	Time out	11256*	2758	686.9	93%
Average	199	1239	1189	347	9%

3.3 A Hybrid Method

Compared to BDD-based methods, SAT techniques do not have memory blow-up problems. However, using current SAT solvers to find all satisfiable solutions is time consuming. Thus, narrowing the search space to make the process faster is critical to pre-image computation. The method presented here to narrow down the search space is to use a BDD-based method to find upper and lower bounds.

Because BDDs can not handle image computation for large digital designs since memory explosion occurs, researchers are motivated to investigate an approximation technique that can be used to estimate reachable states. Past research shows that approximate image computation can be much faster then exact image computation using BDDs [Cho95][TD01].

3.3.1 Narrowing Down the Search Space

Figure 3.10 shows the basic idea of narrowing down the search space. It consists of three parts. One is the lower-bound which is obtained from BDD-based under-approximation pre-image computations. Another part is the upper-bound which is obtained from BDD-based over-approximation pre-image computations. BDDs is used to represent lower and upper bound sets of states. The remaining portion is found by invoking a SAT solver. The upper and lower bound BDDs will be read into the SAT solver and used as boundaries. Here a BDD bounding technique described in [GYA01] is used. It works as follows: whenever a state variable value is set or implied in SAT, the intersection of the partial assignment with the given over (under) BDDs is checked. If the intersection is indeed non-null, the SAT procedure can proceed forward. Otherwise it must backtrack, since no solution consistent with the conjunctions can be found under this sub-tree.

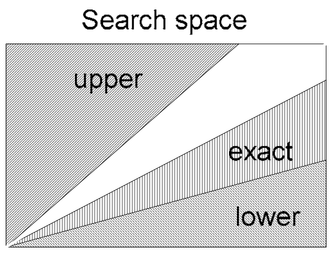


Figure 3.10 Narrowing Search Space

The overall algorithm is shown in Figure 3.11. While the circuit netlist is parsed, three types of transition relations are constructed. One is the exact transition relation in CNF form. One is the over-approximation transition relation (*OverBdd*) in BDD form and another one is the under-approximation transition relation (*underBdd*) in BDD form. These three types of TRs are supplied to the *preImage*() routine. Two BDD boundaries, *Over* and *under*, are calculated. These two boundaries, together with the transition relation in CNF are sent to a SAT solver to get the remaining portion of the reachable states (*remaining*). The overall *frontier* states are the sum of *under* and *remaining*.

Besides the over- and under- BDD bounding, there is also a third type of bounding in the SAT solver. Whenever a satisfiable assignment is found, instead of adding a blocking clause as proposed in [Mcm02][KP03], the assignment is added to a BDD that records all satisfiable solutions found so far (called *remaining*) and continue to search until all satisfiable solutions are found. By implementing BDD bounding with the *remaining* structure, the same solutions found previously can not be searched again. BDD bounding helps early backtrack in the SAT solver and thus speeds up the search process.

```
preImage(S’, CNF_TR, overBdd, underBdd) {
    //get the initial states of formula
    frontier=get_intial_states();
     while(frontier!=NULL){
        //calculate upper bound
         $over = \exists S'.\exists X.(S' \wedge overBdd(S,X,S'))$;
         //calculate lower bound
        $under = \exists S'.\exists X.(S' \wedge underBdd(S,X,S'))$
         //calling SAT for the rest
         remaining = zChaff_ALL(S’, CNF_TR,
over, under);
         //frontier is the combination of under
and exact
         frontier = exact + remaining;
    }
}
```

Figure 3.11 Pre-image Computation Procedure

Two different methods for upper and lower approximation are tried. One method is given by CUDD, which extracts the dense subset of given BDDs. Another way is the algorithm described in [TD01]. Both methods resulted in similar results. To show the effect of approximation on narrowing down the search space, the result of the search space for the pre-image computation of benchmark s1269 is shown in Table 3.2. The property checked is the *liveness property* (*EG*(*p*) where *p* is a conjunction of 8-bit state variables in this example). This type of property specifies that there exists a path that the property holds in every state along the path.

In Table 3.2, column 1 provides the depth of the pre-image computation. It reaches to a fixed point in step 8. In column 2 and 3, the number of over-approximation states and the number of bounding occurrences for the upper-boundary is provided. Columns 4 and 5 are the number of under-approximation states and the number of bounding occurrences for the lower-boundary. Column 6 is the number of exact reachable states. Column 7 provides the number of bounding running to the same state reached before. Detailed results of this work are provided in [LTS06].

Table 3.2 Upper/Under Bound for S1269

Depth	Over (states)	B1	Under (states)	B2	Exact (states)	B3
1	6.872e+10	2	1.762e+09	10	5.369e+10	14
2	5.369e+10	7	8.808e+08	6	4.724e+10	110
3	4.724e+10	14	5.033e+08	7	4.456e+10	154
4	4.456e+10	11	3.460e+08	11	4.349e+10	313
5	4.349e+10	17	2.831e+08	4	4.308e+10	354
6	4.308e+10	21	2.595e+08	13	4.295e+10	460
7	4.295e+10	23	2.517e+08	9	4.292e+10	488
8	4.292e+10	21	2.509e+08	9	4.292e+10	497

3.3.2 Modified SAT Procedure

Modern SAT solvers, like zChaff, Grasp, and Berkmin, are targeted to find a single solution (minterm). They quit after a successful search. zChaff is one of the most popular SAT-solvers. The SAT solver used in this dissertation is a modified form of zChaff. zChaff uses a two-literal watch strategy to speed up the implication and backtrack functions in the search process. Thus, it will assign every free variable until no such variable is available. So a satisfiable assignment found by zChaff is always a minterm. It is very inefficient to find all satisfiable solutions using zChaff. To differentiate zChaff from our modified version, the modified zChaff is referred to as zChaff_ALL. The modifications are targeted to find a satisfiable solution that contains as many don't cares as possible (covers more than one minterm). In addition to the BDD bounding techniques mentioned above, two other modifications are referred to as *early detection* and *expansion*. The overall algorithm for zChaff_ALL is shown in Figure 3.12.

```
zChaff-ALL(){
  while(1) {
    //Check if current partial assignment all ready satisfy
    // every clause
     if(early_detection())
        return SAT;
     //Bounding with over
     if(over_bounding(over))
        backtrack();
      //Bounding with under
     if(under_bounding (under))
         backtrack();
      //expansion
      expansion ();
      if (decide_next_branch()) { // Branching
          //Propagate implications
           while(deduce()==conflict ) {
              blevel = analyse_conflict(); // Learning
           if (blevel == 0)
              return UNSAT;
           else
               //Non-chronological backtrack
                backtrack(blevel);
                    }
         }
        else
           // no branch means all vars have been assigned
            return SAT;
      }
}
```

Figure 3.12 zChaff_ALL Procedure

3.3.2.1 Early Detection

The idea for early detection is quite simple. There are many cases where a partial assignment has already made every clause in the Boolean function satisfied. If every clause is satisfiable, the partial assignment is a satisfiable solution for the Boolean function. All free variables can be seen as don't cares. Early detection will not only terminate the SAT procedure earlier but also avoid much of the unnecessary backtrack steps. The early detection feature works as follows: when a new variable is assigned and all the implication clauses are handled, a check is made to determine if every clause is satisfied by the current assignment. If the result is affirmative, a partial assignment with all other free variables as don't cares is made; if not, the technique continues as before.

Here, the order of the decision is very important since it may be possible to find assignments with more don't cares than with good orders. The principle of giving higher priority to input variables, next-state variables, and intermediate variables are tested respectively. Experiments show that giving higher priority to intermediate variables usually produces more don't cares in a partial assignment on average.

3.3.2.2 Expansion

A further improvement can be accomplished by expanding a current satisfiable assignment to make it contain more don't cares. The condition that the satisfiable assignment can be safely expanded is provided below.

Given a set of states (named *N*) in the next-state variable domain, pre-image computation aims at finding a set of states (named *P*) in the present-state variable domain such that *N* is reachable from *P*. The support set of *N* contains next-state variables, in most cases, not all of them. In other words, the next-state variables that are not in the support set of N can be viewed as free variables.

As was mentioned before, the transition relation is usually produced by a conjunction of bit transition relations. Each bit transition relation (TR_i) consists of one next-state variable (s_i'), some present-state variables, and the input variables. The support of TR_i is represented as *supp*(TR_i).

Suppose *supp*(N)=$\{s_i',...,s_j'\}$ is used to represent the support set of state set N. Each next state variable s_i' corresponds to one bit transition relation (TR_i) and the value of s_i' is specified only by TR_i which is then determined by the support set of TR_i, *supp*(TR_i). In other words, the variables that are not in the support set of the bit transition relation TR_i will not change the value of next state variable s_i' and can be set as don't cares. If we expand this conclusion to all the next state variables in the support set of N. The following observation can be concluded.

Observation: Variables that are not in the support sets of all bit transition relations which correspond to the next-state variables in the support set of state sets N can be safely set to don't cares.

Based on the above observation, our expansion works as follows. The don't care set of the frontier for each iteration will be calculated before calling the SAT solver. The don't care set here only includes present state variables. The don't care set is supplied to a SAT solver. When a satisfiable assignment is found, it may contain some variables in the don't care set but is assigned a specific value by the SAT solver. It is safe to set those variables to don't cares.

3.3.3 Results of Extended Image Computation Approach

In order to evaluate our approach, we ran model checking experiments on benchmarks from the ISCAS'89 benchmark set. The BDD package used is the CUDD BDD package [Som$^+$web]. All experiments are carried out on a 2.6GHz PC running Linux with 1GB of main memory.

Our approach is compared with VIS [Bra$^+$web*] using the **EG** type property checking. The property has the form of **EG**(p), where p is a conjunction or disjunction of a group of state wires. The group size ranges from 1 to 8. A size of 1 normally refers to a single control wire such as reset while a size of 8 corresponds to an 8-bit bus. All parameters in VIS, such as the image computation method, are set as defaults. Dynamic BDD variable reordering is enabled in both approaches. A time limit of 3600 seconds and a memory limit of 850MB are used. The two parameters measured are running time and memory utilization. Table 3.3 shows the results.

The first column denotes the benchmark circuit name. The second, third, and fourth columns provide the number of inputs, outputs, and D flip-flops in each circuit. The fifth and sixth columns represent the execution times and memory required by the BDD-based model checking tool, VIS, and the seventh column represents those by the proposed algorithm. The hyphen '-' means that the algorithm is halted because it exceeds time or memory limits. The table shows that the proposed algorithm can check more circuits than the BDD-based one. In the cases that the BDD-based tools fail to respond, a memory overflow is indicated.

Table 3.3 Comparison of Property Checking

Bench	In	Out	DFF	VIS		zChaff-ALL	
				Time (s)	Mem (MB)	Time(s)	Mem(MB)
S298	3	6	14	0	4.7	0.01	4.7
S526	3	6	21	0.02	4.9	0.69	4.8
S1512	27	21	57	32.7	54.7	0.04	54.0
S1269	18	10	37	3600	-	17.9	25.0
S1423	17	5	74	-	850	10.2	5.7
S5378	35	49	179	-	850	16.5	6.5
S6669	83	55	239	-	850	70.5	27.5
S3384	43	26	104	-	850	45.3	6.2
S9234.1	36	39	211	-	850	18.7	12.2
S9234	19	22	228	-	850	200	12.4

In this section, we present an approach that combines BDD and SAT solvers for pre-image computation. BDD-based over- and under-approximation pre-image computation is used before the SAT solver is invoked to narrow down the search space by computing the upper and lower boundaries for the reachable states. The SAT solver is used to compute the remaining portion of the reachable states. A BDD-based technique is used for upper and lower bounding and to speed-up the search process. Two more techniques are used to compute a satisfiable solution that contains as many don't cares as possible. The experimental results show that our approach performs well.

3.4 Summary of Image Computation

To date, we have extended BDD-based image computation techniques with genetic algorithms [LTS04 LT05]. A "two-step" algorithm for representing a large function as a conjunctive decomposition of BDDs is described where a Genetic Algorithm (GA) approach for ordering individual bit functions is given followed by an affinity-based clustering technique. Applications in image computations were discussed in [LT05]. While developing the image computation algorithm, we noticed the limitations of BDDs and advantages of SAT solvers; we then present a new way to combine BDD- and SAT-based methods for image computation [LTS06]. A BDD-based approximation method is used to calculate the over- and under- estimated boundaries of the reachable states. A SAT solver is used to find the remaining states. The SAT solver is enhanced by techniques we call “early detection” and “expansion” to find a satisfiable assignment containing more don’t cares.

The research presented in this chapter focused on formal techniques for hardware verification. While some formal verification methods are beginning to appear in commercial tools, most formal methods are limited to some types of ICs. Formal verification itself cannot solely accomplish the validation task. Also, simulation is still the dominant tool in industry and the question of combining these two different approaches together to serve the validation purpose has attracted a lot of research attention recently. In the following chapters, we will focus on Integrated Design Validation (IDV) system. All the methods discussed so far are integrated into the IDV flow.

CHAPTER 4

INTEGRATED DESIGN VALIDATION SYSTEM

We are currently involved in a research project that is developing an integrated approach to design validation that takes advantage of current technology in the areas of simulation, and formal verification, resulting in a practical verification engine with reasonable runtime, called the *Integrated Design Validation system* (IDV).

This research utilizes existing simulation, verification techniques, new methods such as those described in Chapter 3, and concentrates on their efficient integration to provide a comprehensive tool for design specification compliance. Recent results in all areas of verification [GDP99, MTS04, LTS04], and simulation [Szy90, KS03] are being used to provide a design compliance tool that will be extremely effective and has the potential to "out-perform" the current "state-of-the-art" methods focused upon a single methodology. The focus in the IDV system is in the development of a circuit complexity analyzer and partitioning tool based upon design hierarchy. Also, the development of coverage analysis methods that compute a degree of design validation and invoke methods for intelligently updating the partitioning tool for further validation iterations is crucial. There have been recent attempts to tightly combine two different verification tools [BS98, HKWF02], most notably SAT solvers and BDD approaches for equivalence checking; however, no overall verification/simulation engine with significant analysis before design validation occurs has been produced.

4.1 System Description

The overall structure of the prototype IDV system is shown in the block diagram of Figure 4.1. A primary focus for this project is the complexity analyzer, partitioning, and coverage analyzer blocks, to determine the most effective use of formal verification.

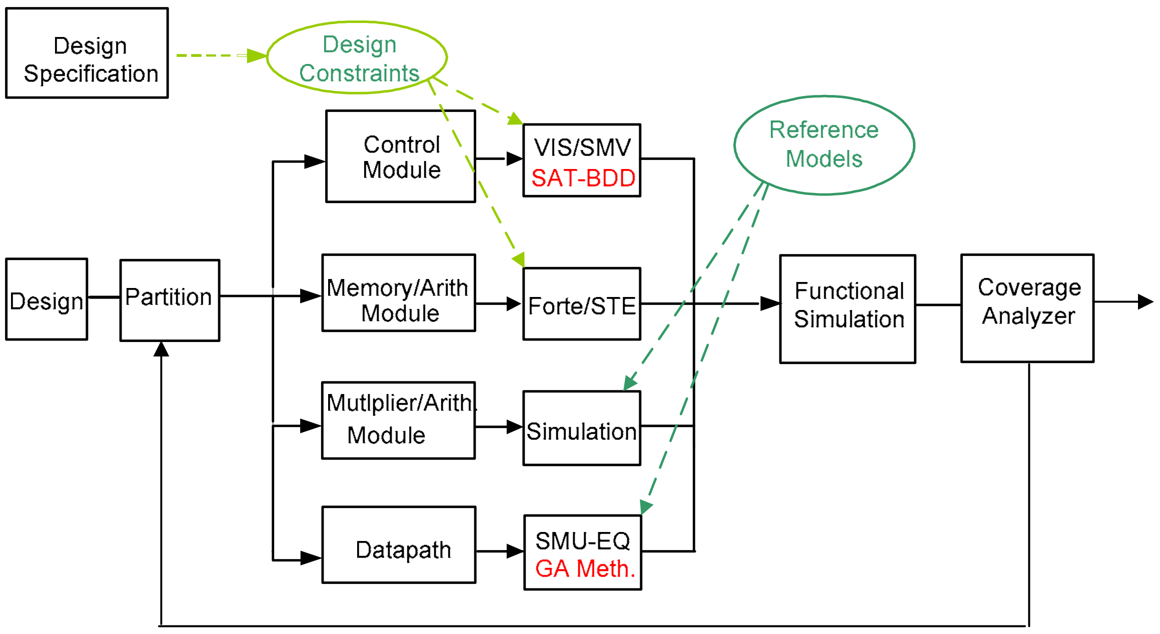


Figure 4.1 Architecture of the Integrated Design Validation System

4.1.1 Complexity Analyzer

The complexity analyzer estimates the complexity of an RTL or netlist design based on existing methods for controller/datapath extraction. Integration with the partitioner is crucial for this function. The extracted control and datapath portions of the circuitry are being analyzed for the applicability of various techniques based upon known existing strengths of verification and simulation tools. As an example, a portion of a datapath may be supplied as input to simulator or an equivalence checking tool.

Given different constraints, different tools can be applied to verify the constraints. The complex analyzer is also utilized for such purpose. For example, constraints expressed in simple trajectory formula can be verified via Symbolic Trajectory Evaluation (STE) or property checking tools such as Verification Interacting with Synthesis (VIS). STE has high capability in term of number of flip-flops that a circuit contains.

4.1.2 Design Partitioning

One of the biggest hurdles in applying formal techniques is to correctly identify target circuits. Although a lot of work has been accomplished with respect to partitioning for logic and physical level synthesis, there is not as much for design validation and simulation. Currently designer-defined hierarchy is utilized for partitioning. Designers typically design the system with multiple RTL blocks (these blocks usually mirror the floor-planned design) in order to apply modern design tools. Methods that exploit such an inherent design hierarchy have been used in the past such as the jMocha tool [AA+01].

Based on the design hierarchy, a *process-module* (PM) graph which describes the hierarchy of the design is built. Each node in the graph represents a component/module/process and edge corresponding to the interconnections of these components. The PM graph is utilized to partition the design.

4.1.3 Coverage Analysis

Some work has been done in terms of coverage analysis particularly with respect to evaluating the effectiveness of simulation-based validation. An overview of design validation coverage methods is given in [TK01] that classify existing metrics in terms of code coverage, metrics based on circuit structure, metrics defined on finite state machines, functional coverage, error models, observability, and metrics applied to specifications. These existing metrics will be used as a starting point for the development of the coverage analyzer. Not much work has been accomplished in terms of combining formal verification with simulation and computing the overall coverage.

4.1.4 Verification and Simulation Tools Comprising IDV

Our goal is to integrate the tools and make them complement each other. Various tools have been developed for formal verification and simulation. Choosing the right tools will set up the baseline for the success of the IDV system. In the following we describe the tools that are selected, developed, or still under development.

4.1.4.1 Symbolic Trajectory Evaluation

Symbolic Trajectory Evaluation (STE) [HS97] is a model checking approach designed to verify circuits with very large state spaces. STE is more sensitive to the property being checked instead of the size of the circuit. The STE package selected is from the Intel Strategic Research Lab, *Forte*. It also supports a simple yet effective compositional theory. Two important properties of STE are:

a. It is suitable for verifying designs of circuits at the gate or switch level
b. STE provides accurate models of timing, which is reflected in the types of properties checked for.

STE originated from the idea of using multi-level simulation and ternary-valued symbolic simulation. It is a formal verification method that is close to traditional simulation. One of the distinguishing features of STE is that the state space is represented as a lattice. The partial order of the lattice represents an information ordering or abstraction relation between states. The higher up we go in the information ordering, the more information we have. The computational advantage of this is that, given the appropriate logical framework, if a property is proved to hold in a state in the lattice, it holds for all states above it in the lattice. Another important fact is that circuits have natural representations as lattices, and the use of the information ordering allows us to easily abstract out the necessary information for property checking.

The properties to be checked are represented as *Temporal Logic* (TL). TL is usually propositional or first-order logic augmented with temporal modal operators that allow reasoning about how the truth values of assertions change over time. TL can express safety and liveness properties, such as “property *p* holds at all times” or “if *p* holds at some instant in time, *q* must eventually hold at some later time.” Properties of this sort can be employed to specify desired properties of systems, i.e. in a traffic signal control system, “the signals at both directions should never be green at the same time” and “the signal at one direction will eventually be green”.

The properties that STE focuses on are a restricted TL that offers only the next-time operator [SB95], which is called a trajectory formula. A trajectory assertion has the form $A \rightarrow C$, where A and C are trajectory formulas, referred to as *antecedents* and *consequences* respectively. Informally, a trajectory assertion holds for a circuit M iff each sequence of states of M that satisfy the antecedent A also satisfies the consequent C. Typically, A specifies constraints on how the inputs of a circuit are driven, while C asserts the expected results on the output nodes [KG99]. For example, the formula $((read\_enable=1 \wedge addr) \rightarrow (out = \mathbf{Next}(M[addr]))$ asserts that if signal *read_enable* is asserted and *address* is specified, the output of memory is the value stored at *address* in the next cycle.

4.1.4.2 Verification Interacting with Synthesis (VIS)

VIS is a verification package developed jointly at the University of California at Berkeley, the University of Colorado at Boulder, and more recently, at the University of Texas, Austin [Bra+web*]. VIS is able to synthesize finite state systems and/or verify properties of such systems, which have been specified hierarchically as a collection of interacting finite state machines. VIS is built upon the BDD package developed by the University of Colorado at Boulder, referred to as CUDD [Som+web]. VIS and CUDD have been used extensively in academia for model checking.

STE and VIS are both capable of model checking. They differ in the following aspects.

Properties: VIS can verify more properties since it uses CTL while the trajectory formula supported by STE is less expressive.

Capacity: STE can handle bigger circuits in terms of latches and bit cells (over 1000 latches). VIS usually exceeds memory capacity when there are more than 200 latches. STE trades expression power for capacity.

BDD Memory: The underlying engine for VIS is compact symbolic representation of the circuit model in terms of BDDs. The underlying engine for STE is symbolic simulation where the size of BDDs is related more to the properties instead of circuit model.

Application: Based on above differences, we can conclude that VIS is better in control dominated designs while STE is more suitable for memory dominated circuits. Actually, STE has been used extensively in property checking for memory.

4.1.4.3 Speed5

Speed5 is a Tegas-like, 5-value multi-modal, assignable-delay, five-valued simulator [Szy90] [KS03]. It performs gate-level and functional-level simulation. Nominal and critical timing (min/max) delays are used in simulation. Speed5 has fault simulation ability by fault generation and insertion into the simulated circuit. Fault models that are provided are: stuck-at, shorts, transient fault models, and multiple faults. Performance is improved by parallel simulation of faults where a specified number of faults are simulated in one pass. The number of faults per simulation is determined from indistinguishable fault classes, fault blocking characteristics and the desired diagnostic resolution.

4.1.4.4 SMU Equivalence Checker

The equivalence checker developed in our group (SMU-EQ) [LTS04] performs quite well on large designs. The core part of the equivalence checking tools is image computation where conjunctive scheduling is very important to reduce the BDD size of intermediate computations. In our approach, a genetic-based approach is developed to minimize total lifetime and active lifetime at the same time. Experimental results show that SMU-EQ is very effective. We also incorporated a SAT engine into our equivalence checker to make it more robust and to handle more designs that uses the ideas of "early detection" and "expansion" described in the previous chapter.

4.1.4.5 SMU Functional Simulator

A functional simulator is also under the developing stage in our group which is a critical part of the system-level simulation portion of IDV. At the system level, we are interested in the interconnection of modules versus the internal function of separate modules. The functionalities of these modules are fully verified by VIS or STE or simulated by Speed5 before they are integrated into the functional simulator.

A more detailed introduction of IDV can be found in [LTS05].

4.2 Validation Flow with IDV

The IDV system is a constraint-based system. Constraints specify the system's operation such as what validation method will be used for each design module, the properties to be verified, and so on.

Figure 4.2 shows the validation flow chart for the IDV system. The circuit is parsed as a netlist either in blif or structural RTL format. The next block is the partitioning portion. The design hierarchy information are utilized for partitioning. This is reasonable since most current designs are created in a hierarchical format. After partitioning, bases on the result of complexity analysis, all modules and corresponding constraints are supplied as input to appropriate validation engines for verification and/or simulation. The general rules are listed as follows:

a. VIS deals with complex properties presented in CTL and control logic.

b. STE deals with simple TL and control logic, and also all properties related to memory

c. SMU-EQ deals with datapaths that do not cause memory explosion.

d. Speed5 simulator deals with multipliers or other complex components specified by the designers

e. Functional simulation/system-level simulation is used as the last step for the interconnections of the components

After the sub-modules are validated separately, a functional simulator will be applied to simulate the system but the main focus will be on system interconnections. The coverage analysis will provide a degree of confidence of the design validation. When the coverage value is low, the component that is simulated in previously stage is further partitioned to smaller modules. The coverage for further partitioned component can be improved with formal verification method or more simulation. The increased coverage on component also improves the coverage for the design. The tool continuously decreases the granularity of the partitions until the desired coverage goal for validation is reached.

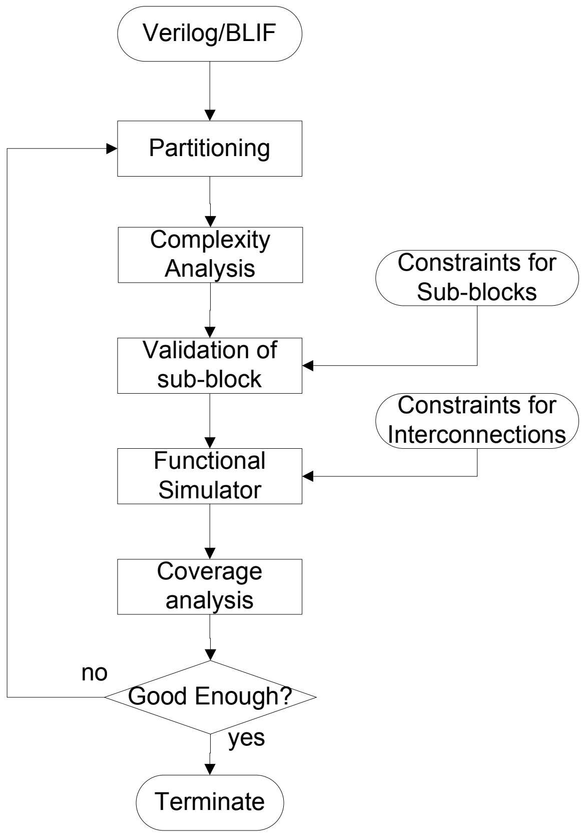


Figure 4.2 Validation Flow of IDV System

4.3 IDV Implementation Architecture

The software architecture of IDV is shown in Figure 4.3. It consists of three blocks: a shell-like command line input module, a coordinator, and several core engines. The shell-like command line input module accepts commands and sends them to the coordinator. An example of the shell-like command line input is shown in Figure 4.4. The coordinator accepts commands from the command input module, translates them to a format that the core engines can recognize, sends them to the core engines, and finally collects the results and displays them.

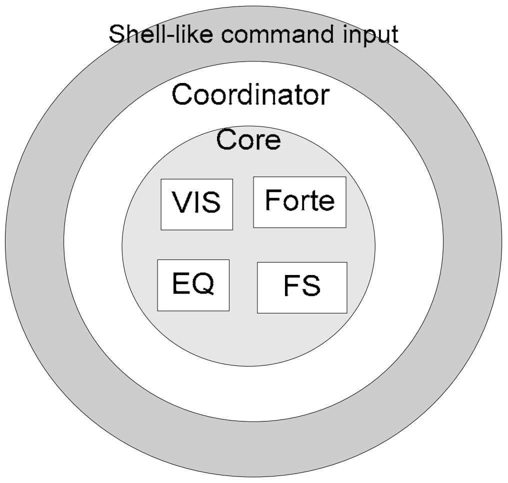


Figure 4.3 Architecture of IDV System

```
%Unix _prompt%  IDV
IDV>read_blif moduleA.blif moduleB.blif moduleC.blif
Info: three modules (A,B,C) has been defined, top level is A;
IDV>set_method –module A VIS(parameter for VIS)
IDV>set_method –module B Forte(parameter for Forte)
IDV>set_method –module C Sim(parameter for Sim)
IDV>validate_submodule –module A
IDV>validate_submodule –module B
IDV>validate_submodule –module C
IDV>read_constraint –Interconnection
IDV>validate_interconnection
IDV>quit
%Unix _prompt%
```

Figure 4.4 Command Line Input Example

CHAPTER 5

VERIFICATION RESULTS

5.1 Types of Digital Circuits Suitable for IDV

As described in previous chapters formal verification methods work well for some “types” of circuits. Also, different tools are suitable for different types of circuits. There is no single tool that can handle designs containing different subcircuits. The purpose of IDV is to develop an integrated environment that applies appropriate tools for suitable subcircuits. Thus, the whole design can be validated in one framework.

The circuit types that are especially suitable for IDV are those that contain datapaths, controllers, memory units, and interfaces. One of our target designs is *Systems-on-a-chip* (SoC), which integrates all components on a single chip. SoCs usually contain IP cores such as embedded CPUs, a memory subsystem, controller, standard interfaces (e.g., USB, PCI, Ethernet), and software components. IP cores are typically available through purchase from third parties or developed in-house previously.

To demonstrate the ability of IDV, two circuits are developed that contain most of the required components of a SoC: a datapath, memory units, and a controller. The two designs used as example here are partially sponsored by the *Semiconductor Research Corporation* (SRC). Since the designs are used to benchmark the algorithms being developed for integer operations, it is important to verify these designs. This type of design cannot be totally verified formally due to the memory explosion problem in formal methods and the coverage problem with simulation. No single tool environment has been presented to validate such types of designs previously.

5.2 Integer Powering Circuits

In this section, we describe the design and theory of the integer powering circuits developed under the SRC sponsored project that will be the subject of the verification effort.

Algorithms for computing the powering operation $z = x^y$ where x, y, and z are positive integers have been the subject of considerable research. A “fast” method, which can be traced back to al-Kashi in the 15th century has been described in many popular texts [CLR01][Par00][Knu81]. Figure 5.1 shows the basic algorithm for the “fast” method. This binary squaring method first determines $x, x^2, x^4, x^8, \ldots$ and processes the bits of y right-to-left to multiply by the appropriate binary powers of x to determine x^y. Since this method involves multiplication and squaring operations, it is also referred to as the *Multiplier Method* (MM).

```
Input: $k, x = x_{k-1}x_{k-2}..x_2x_1 1$, $y = y_{k-1}y_{k-2}..y_2y_1y_0$
Output: $z = |x^y|_{2^k}$ .
$L1$: $z := 1$; $q := x$;
$L2$: for $i := 0$ to $k-1$ do
$L3$:     if bit$(i, y) = 1$ then
$L4$             $z := |z \times q|_{2^k}$
$L5$      end
$L6$:     $q := |q \times q|_{2^k}$
$L7$:end
```

Figure 5.1 Squaring and Multiply Based Powering Operation Algorithm

Given that x, y, and the result z are all non-negative *k*-bit integers, such as *k* = 8, 16, 32, 64, 128,…, this integer-valued powering operation based on *MM* requires *O*(*k*) squaring and *O*(*k*) multiplication operations in the worst case which is expensive for hardware implementation. A simpler algorithm that avoids the use of a large multiplier will greatly benefit efficient hardware implementation. There is a further need for a right-to-left digit serial algorithm that requires less time for lower precision operations when a family of precision levels is implemented in hardware.

The inheritance principle for integer operations was introduced in [MFT05]. It can be summarized as "the *k*-low order bits of the result depend only on the *k*-low order bits of the operands for all $k \geq 1$". This principle provides the basis for right-to-left digit-serial integer operations. Specifically, assume the low order (*k*-1)-bits of the result obtained from the (*k*-1) low order operand input bits have been determined. Simply by incorporating the *k*-th bits of the operands, the *k*-th result bit can then be determined with the (k-1) lower order result bits “inherited” from the preceding serial computation. The formal statement for this inheritance principle can be found in [MFT05].

Modular notation $\left|\bullet\right|_{2^k}$ defined in [ST67] is employed in this chapter. For a binary integer $x = b_{n-1}b_{n-2}..b_2b_1b_0$, the modular notation $\left|x\right|_{2^k} = b_{k-1}b_{k-2}..b_2b_1b_0$ denotes the value of the standard low order k- bit string for all $1 \le k \le n$.

Note that every k-bit integer x is uniquely represented by the triple (s, p, e) such that $x = \left|(-1)^s 2^p 3^e\right|_{2^k}$. x has a unique factorization into odd and even terms $x = 2^p n$ with n odd. All the odd values can be represented as $\left|(-1)^s 3^e\right|_{2^k}$ [FM04] [Ben99]. The triple (s, p, e) and its manipulation are referred to as the *Discrete Logarithmic System* (DLS). Based on DLS representation, the powering operation can be rewritten as the following:

$$z = \left|x^y\right|_{2^k} = \left|((-1)^s 2^p 3^e)^y\right|_{2^k} = \left|(-1)^{sy} 2^{py} 3^{ey}\right|_{2^k}$$

Integer x is first converted to DLS such that $x = \left|(-1)^s 2^p 3^e\right|_{2^k}$. Then the power y is distributed to the three exponents. Three multiplications are required to obtain $s \times y$, $p \times y$, and $e \times y$. $(-1)^{sy}$ determines the sign of the result. 2^{py} determines the number of least significant zeros in the result. Finally $\left|(-1)^{sy} 2^{py} 3^{ey}\right|_{2^k}$ is deconverted to obtain z. Without loss of generality, we focus on odd numbers in the following discussion. For odd numbers, item $p \times y$ is ignored due to $p = 0$.

The block diagram for this approach is shown in Figure 5.2. Binary-to-DLS conversion refers to determining the triple (s, p, e) given the $k-$bit integer n, and deconversion refers to determining n given the triple (s, p, e), where n, s, p, and e satisfy $n = \left|(-1)^s 2^p 3^e\right|_{2^k}$.

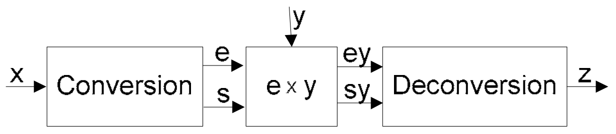


Figure 5.2 Serial Odd Integer Powering Algorithm Based on DLS

Two types of conversion/deconversion algorithms are introduced: one is iterative computation based and the other is a table look-up based technique.

Efficient iterative computation algorithms for integer-to-DLS conversion and deconversion were presented at the algorithmic level in [FMT05a] [FMT05b] [Fi05]. Since both the conversion and deconversion algorithms employ k sequential steps of a table lookup operation interleaved with a shift-and-add modulo 2^k operation. This approach is also referred to as the *DLS iterative computation based method* (DLSiter) in this document.

We also present a table look-up conversion/deconversion approach based on an encoding scheme that provides a one-to-one mapping between k-bit integers and k-bit DLS values. This encoding scheme is scalable for all practical word sizes. This k-bit DLS encoding also satisfies fundamental properties allowing table lookup based conversion and deconversion to be completed with tables whose size are less than 8K

Bytes in each direction for $k \leq 16$. This approach is referred to as *DLS table lookup based method* (DLStable).

Hardware implementations for the integer powering operation with the DLS conversion/deconversion methods are described in next section.

5.2.1 DLSiter Conversion/Deconversion Circuit

DLSiter conversion can be accomplished separately with deconversion as Figure 5.2 shows. The detailed algorithm for *DLSiter* conversion/deconversion can be found in [FMT05a] [FMT05b] [Fi05]. The conversion algorithm is also referred to as *Discrete Log* (DLG) while the deconversion algorithm is referred to as *Exponentiate* (EXP). Here a faster algorithm is presented where the conversion/deconversions are processed in parallel. For every available bit of e, a bit of the intermediate product is generated and followed by a bit of z being produced. This method is referred to as the parallel algorithm and is described in Figure 5.3.

```
Stimulus: $k, x = x_{k-1}x_{k-2}..x_2x_1 1$, $y = y_{k-1}y_{k-2}..y_2y_1y_0$
Response: $z = | x^y |_{2^k}$ .
Method
L1: if $|x|_8 \in \{1,3\}$ then $s := 0$ ;
L2: else        $s := 1$;  $x := 2^k - x$
L3: end
L4: $p := 1$; $e := 0$; $z := 1$; $q := 0$; $t = e$;
L5. if bit($1, x$)=bit($1, p$) then
L6:     $p := |p + p << 1|_{2^k}$ ; $e := e + \mathrm{dlg}(3)$
L7:     if bit($0, y$)=1 then
L8:             $z := |z + z << 1|_{2^k}$ ;
L9:     end
L10: end
L11: for $i := 3$ to $k-1$ do
L12:    if bit($i, x$)=bit($i, p$) then   //update for DLG
L13:            $p := |p + p << i|_{2^k}$ ; $e := e + \mathrm{dlg}(2^i + 1)$
L14:    end
L15:    $t = t << 1$;
L16:    if(bit($i-2, e$)=1)
L17:            $m = m + t$   //accumulator.
L18     end
L19:    if bit($i-2, m$)=1 then
L20:            $q = q + 2 << (i-2)$
L21:    if bit($i, q$)=1 then   //update for EXP
L22:            $z := |z + z << i|_{2^k}$ ;
L23:            $q := q - \mathrm{dlg}(2^i + 1)$
L24:    end
L25:end
```

Figure 5.3 Parallel Integer Powering Algorithm

The initialization stage is performed in lines $L1 - L4$ where all the required initialization steps are accomplished. The second stage ($L5 - L10$) performs the computation for $i=1$ where i is the index for iteration and $1 \le i \le k-1$. The third stage contains the main iteration step and is represented by lines $L11 - L25$. The third stage can be separated into 3 sub-stages. Both p and e are updated (i.e. $L12 - L14$) which generates one bit of e based on the conversion algorithm defined in [FMT05a]. The second sub-stage (i.e. $L15 - L18$) corresponds to the accumulator used to compute $e \times y$. The third stage (i.e. $L19 - L24$) updates z according to the deconversion algorithm defined in [FMT05b]. The final result is obtained at line $L22$. As can be seen by inspection of the algorithm, the time complexity is essentially *k* dependent shift-and-add modulo 2^k operations.

5.2.1.1 Hardware Implementation

Hardware implementation for *DLSiter* conversion/deconversion is described here. The state diagram of the hardware implementation is given in Figure 5.4. There are 6 states available, **Load**, **Init**, **Loop_DLG, Loop_ACC, Loop_EXP** and **Ready**. The **Load** state is also a reset state. It accepts input when the *load* signal is asserted and also performs all the initialization operations in lines $L1 - L4$. The **Init** state accomplishes the operations in the second stage ($L5 - L10$) in Figure 5.3. The **Loop_DLG, Loop_ACC, Loop_EXP** states correspond to the 3 sub-stages in the algorithm. The loop count goes from 3 to *k*, with a maximum of *k*-3 iterations. The **Ready** state is the state that outputs the result. The circuit automatically transitions into the **Load** state after **Ready** state.

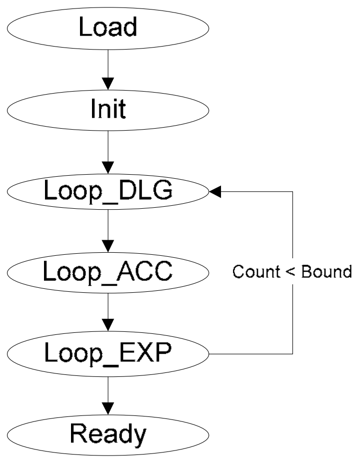


Figure 5.4 State Diagram for DLSiter Implementation

There are three major components in the *DLSiter* circuit, a controller, a ROM lookup table, and a computation datapath.

The major components in the datapath are adders, shifters, and units called bit-checkers that are used to check if a certain bit is asserted. The output of the bit-checker controls the operation of the adders and shifters. No operation is performed if the output is false; otherwise, registers holding p, e, z, q are updated by the shifter and adder. The controller consists of a counter and state controller block. The state controller starts and stops the counting procedure. The output of the counter, *count*, is used for purposes such as address generation for the ROM, index generation for the bit checker, and feedback to the state controller for state transition. The ROM is used as a lookup table for the DLS values for $2^i + 1$ where $3 \le i \le k$. The modular operation given in the algorithm is handled by limiting the size of p, e, z, q. The sizes of p, e, z, q are set to *k*. Thus, while updating p, e, z, q, the result values may be longer than the specified size (or overflow). Overflow bits are ignored since this computation is performed modulo 2^k.

In order to evaluate the effectiveness of *DLSiter* as compared to the *MM*, both methods are implemented in Verilog RTL and synthesized into circuits using the Synopsys tool set based on a standard cell library from Synopsys [Syn03]. Table 5.1 shows the cell delay calculated based on the output net total capacitance (cap.) for three types of D-flip-flops in the library. From Table 5.1, it is apparent that the library is not a fast technology library, and our circuit performance results are relative to the characteristics of this cell library.

Table 5.1 Technology Library Parameters

Cell name		denrq1	denrq2	denrq4
Cell delay(ns)	rise	0.379	0.266	0.297
	fall	0.458	0.319	0.357
Total output cap.(pf)		0.051	0.038	0.0386
Size(μm)	x	11.89	12.30	13.12
	y	3.69	3.69	3.69

More detailed information on this implementation can be found in [LFTM05, LTM06]. Table 5.2 compares the results of *DLSiter* and *MM* for *k*=8, 16, 32, 64, 128 respectively. The speed and area trends of the two circuits in terms of word size k are plotted in Figure 5.6 and Figure 5.7 respectively. Figure 5.6 shows that *DLSiter* is faster than *MM* for all *k* values. Regarding area, *DLSiter* requires more space for small word sizes but increases slowly compared with *MM*. Thus when $k \geq 64$, *DLSiter* requires less area. It should be noted that the area values reported here are only the net area required by the total cell area since we do not route the resulting circuits, thus additional area required by routing is not included.

Table 5.2 Comparison of Layout Result

k (bits)	speed(ns)		core area(μm^2)	
	DLSiter	MM	DLSiter	MM
8	2.05	2.4	23386.4	8207.48
16	2.41	3.45	40306.7	26076.3
32	2.75	4.55	109135	79409.1
64	3.52	5.55	184725	302942
128	3.8	6.8	371366	1.26E+06

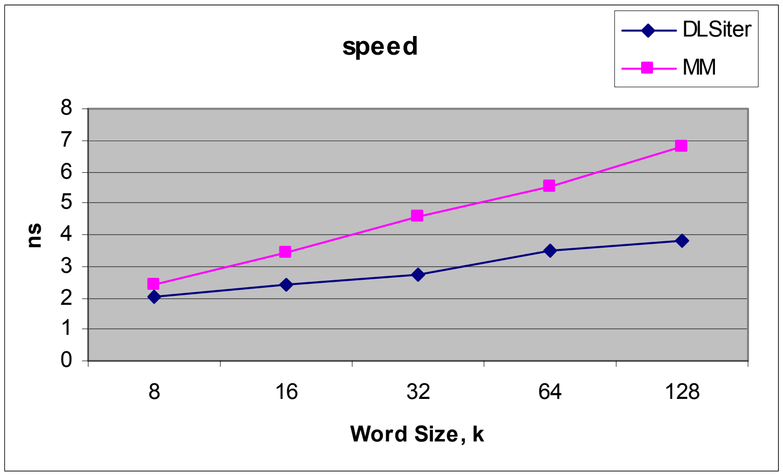


Figure 5.5 Speed Trend of the Two Circuits

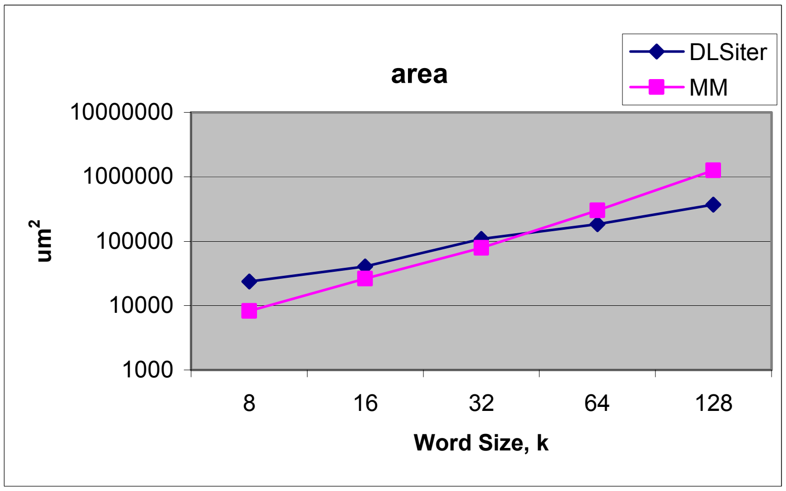


Figure 5.6 Area Trend of the Two Circuits

5.2.2 DLStable Conversion/Deconversion Circuit

A compact encoding scheme for the DLS triple (s, p, e) is introduced here. This encoding employs variable length fields for p and e, and provides a one-to-one mapping between *k*-bit DLS values and *k*-bit unsigned binary integers. Example tables and figures are used to illustrate the encoding and some of its significant properties.

The one-to-one mapping between 5-bit DLS values and 5-bit integers is given in Table 5.3. The DLS bit string is partitioned as follows to determine the three exponents *s*, *p*, and *e*. Consider the line in the table for DLS string 10110_2 which yields binary $01110_2=14_{10}$.

Table 5.3 Conversion Table from the 5-bit DLS Number to the 5-bit Integers [0,31]

Discrete Log Number System (DLS) Encoding	Paritioned DLS Bit Strings			Integer Value	Standard Binary	Integer Parity
	e	$e_0 \oplus s$	2^p	$\left\|(-1)^s 2^p 3^e\right\|_{32}$		
00001	000	**0**	1	1	00001	Odd
00011	000	**1**	1	31	11111	
00101	001	**0**	1	29	11101	
00111	001	**1**	1	3	00011	
01001	010	**0**	1	9	01001	
01011	010	**1**	1	23	10111	
01101	011	**0**	1	5	00101	
01111	011	**1**	1	27	11011	
10001	100	**0**	1	17	10001	
10011	100	**1**	1	15	01111	
10101	101	**0**	1	13	01101	
10111	101	**1**	1	19	10011	
11001	110	**0**	1	25	11001	
11011	110	**1**	1	7	00111	
11101	111	**0**	1	21	10101	
11111	111	**1**	1	11	01011	
00010	00	**0**	10	2	00010	Singly Even
00110	00	**1**	10	30	11110	
01010	01	**0**	10	26	11010	
01110	01	**1**	10	6	00110	
10010	10	**0**	10	18	10010	
10110	10	**1**	10	14	01110	
11010	11	**0**	10	10	01010	
11110	11	**1**	10	22	10110	
00100	0	**0**	100	4	00100	Doubly Even
01100	0	**1**	100	28	11100	
10100	1	**0**	100	20	10100	
11100	1	**1**	100	12	01100	
01000		**0**	1000	8	01000	Triply Even
11000		**1**	1000	24	11000	
10000			10000	16	10000	Quadruply Even
00000			00000	0	00000	Zero

The variable length fields are interpreted by first determining the value p from its unary encoding, then finding e in binary form, and finally the sign bit. Specifically, the parsing of DLS string 10110_2 begins from the right-hand side determining the variable length field identifying $2^p = 10_2$ by reading until the first unit bit is encountered. The 2-bit field "unary" encoding of p determines $p = 1$. The next bit is a separation bit providing the logical value $s \oplus e_0$ used to determine s after e is determined. The remaining leading bits are the $5-(p+2)$ bits of the exponent $0 \le e \le 2^{5-(p+2)} - 1$. In this example, e=10_2=2_{10}, and then s=1 is determined from $e_0 = 0$ and $s \oplus e_0 = 1$. Finally $\left|(-1)^1 2^1 3^2\right|_{32} = \left|-18\right|_{32} = 14$ is obtained. Note that the low-order bit field determining the even factor 2^p is an identical field in both DLS and binary integer encodings.

This above encoding scheme also satisfies the inheritance property [MFT05] providing that the $(k-1)$-bit DLS encoding for an integer in the range $0 \le x \le 2^{k-1} - 1$ can be determined by simply truncating the leading bit of the k-bit DLS encoding. This allows the encoding of the even integers to be simply determined from a table for the odd k-bit integers by shifting out the leading bits of the odd factor. For example, the DLS string for 14 can be obtained by shifting left one place the DLS string for 7. The specification of the separation bit as equal to $s \oplus e_0$ is an important condition that provides inheritance property.

Based on the inheritance property, the conversion/deconversion in Table 5.3 can be visualized by the lookup trees illustrated in Figures 5.7, 5.8 and 5.9. Figure 5.7 and Figure 5.8 show the lookup trees of integer-to-DLS conversion for odd and even integer respectively. Navigation in Figures 5.7 and 5.8 occurs by reading down with edge direction determined by the 5-bit integer string $a_4a_3a_2a_1a_0$ right-to-left. The DLS output (right-to-left) is obtained from the bits extracted from the vertices along the path. The deconversion from DLS-to-binary is similarly illustrated for the odd values in Figure 5.9. The even number deconversions can be accomplished by employing shifting using the inheritance property or from extending the table to include an even part portion.

Besides the inheritance property, sign symmetry, one-to-one mapping, and normalization are 3 other properties that can reduce the tree size dramatically. Sign symmetry dictates that the result of the operation on the 2's complement of the input is the 2's complement of the output. One-to-one mapping holds when distinct *n*-bit inputs have appropriately determined distinct *n*-bit outputs. Normalization means that the result for an even input can be derived based on the result of corresponding odd input. More descriptions of these properties can be found in [Fi05]. These properties in the lookup trees allow the conversion/deconversion for 16-bit integers to be accomplished with 2-8KBytes table for DLS-to-binary and 2-8KBytes table for binary-to-DLS. The presentation here is given to illustrate the representation which pertains to general *k*. However, when $k > 16$, the table size is quite large and impractical for hardware implementation, the conversions can be better handled with *DLSiter*.

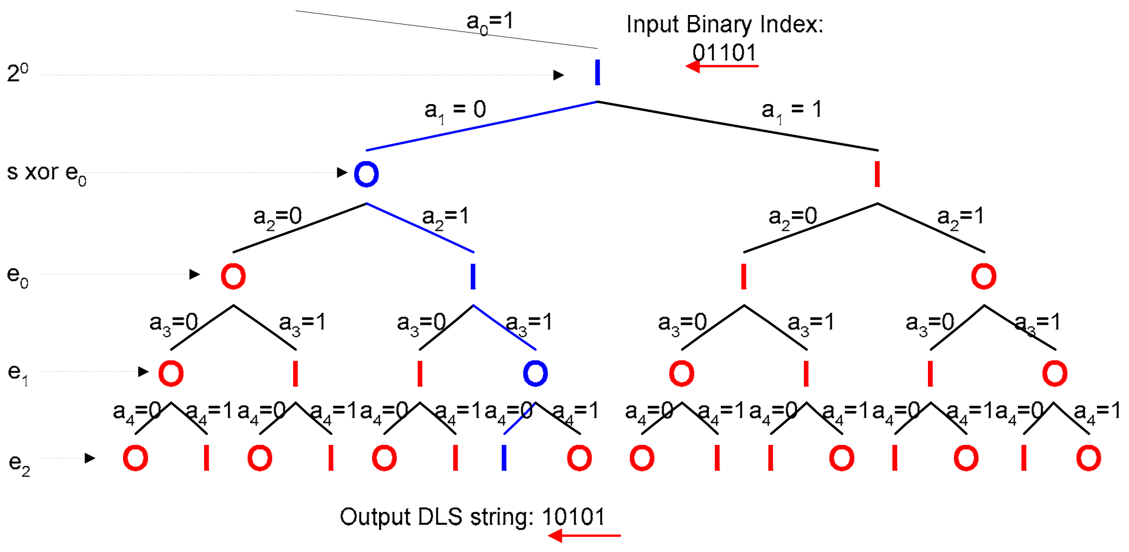


Figure 5.7 Lookup Tree for Odd Integer Binary to DLS Conversion

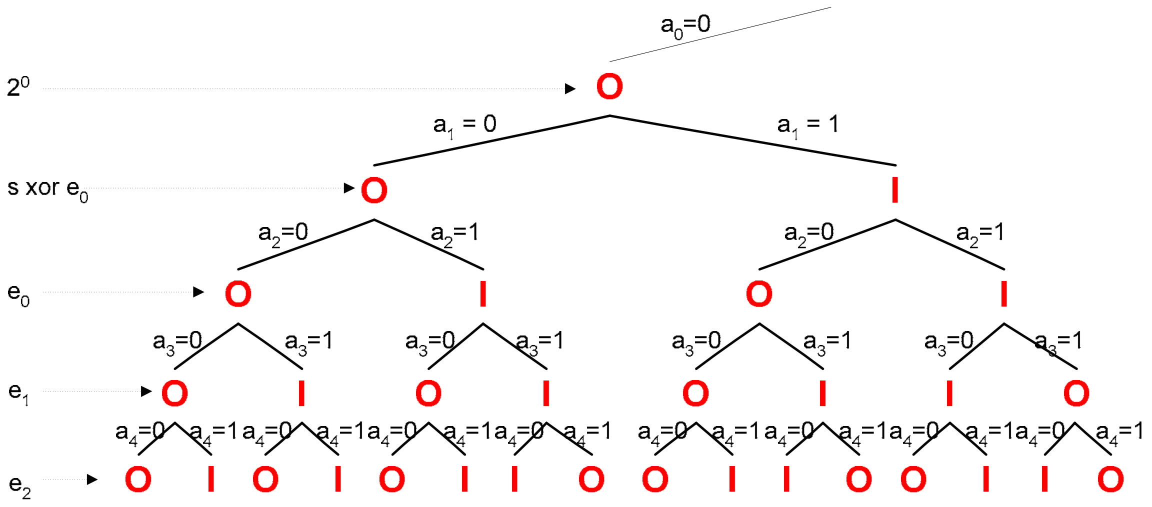


Figure 5.8 Lookup Tree for Even Integer Binary to DLS Conversion

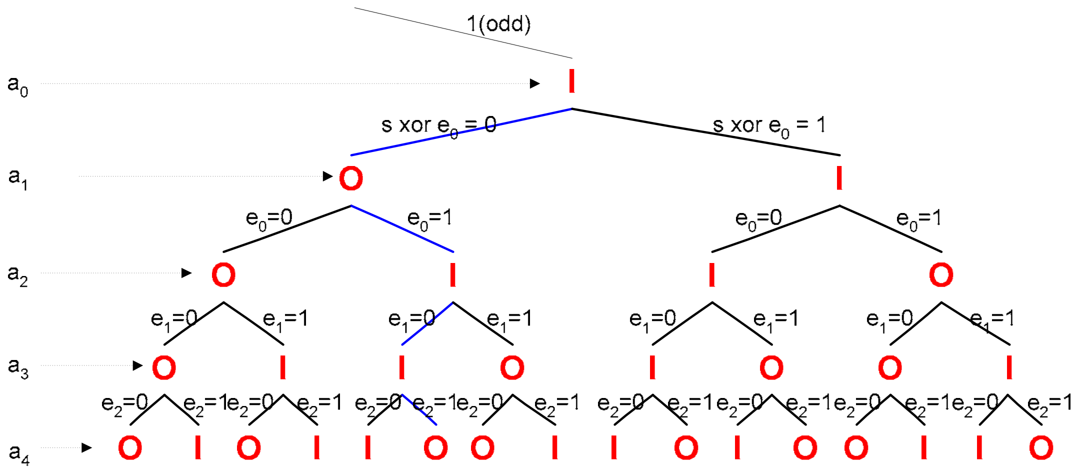


Figure 5.9 Lookup Tree for DLS Odd number to Integer Binary Conversion

DLStable allows for direct conversions between binary and DLS resulting in fast performance. Figure 5.10 shows the table lookup architecture. It consists of three major components: a pre-processing block, a post-processing block and a ROM. The pre-processing block produces the ROM address based on the input operand. After the data in the ROM is read, the post-processing block selects the correct bit fields, and performs some additional processing, such as complementation. Two schemes for *DLStable* are compared here. One scheme uses a larger table with less pre- and post-processing logic while the other uses a smaller table with more pre- and post-processing logic.

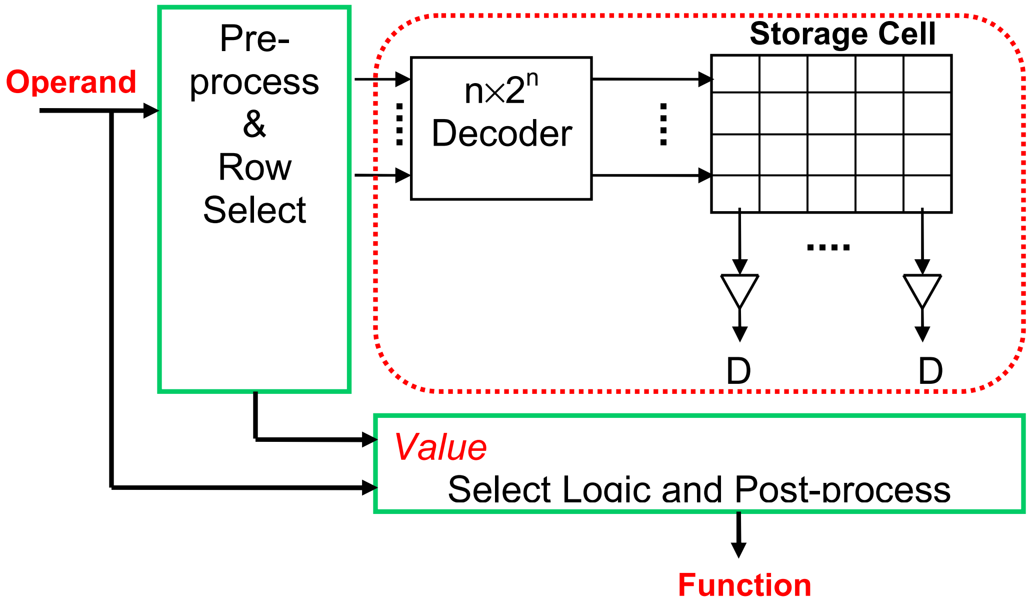


Figure 5.10 Table Lookup Architecture

5.2.2.1 DLStable with Larger Table

For *DLStable* with larger-sized table implementation, the inheritance and one-to-one mapping properties are exploited. Due to the one-to-one property, only the left children of the lookup tree are stored. No normalization and sign symmetry are utilized; therefore, no pre-processing is required before table lookup occurs. For post-processing, conditional complementation is required on the table output value with the input value since only the left children values are stored in the table. In the following, the circuit structure and the hardware implementation are discussed.

The ROM structure and select logic are shown in Figure 5.11. The ROM represents a 3-level tree. The first level forms 256 rows where the low 8-bits ([$a7$:$a0$]) are used as the address bits. Each row has 264 bits. 8 leftmost bits are selected directly as output in the row. The other 256 bits represent the second and third level tree. In the second level, four sub-trees between levels 8 and 9 are formed as four bytes. [$a9$:$a8$] are used to select one of the four bytes. After the byte is selected, [$a10$] and [$a11$:$a10$] are used to select one bit from the selected byte respectively, while the other two bits are extracted directly without selection. Therefore, a total of 4 bits are extracted from the selected byte. In the third level, there are 32 sub-trees between level 8 and level 12 formed as 32 7-bit fields. [$a12$:$a8$] are used to select one of the 32 7-bit fields. [$a13$] and [$a14$:$a13$] are used to select one bit from the selected field respectively, while the single rightmost bit is extracted directly without selection. Therefore, a total of 3 bits are extracted from the selected 7-bit field. Finally, a 15-bit output is produced from the select logic.

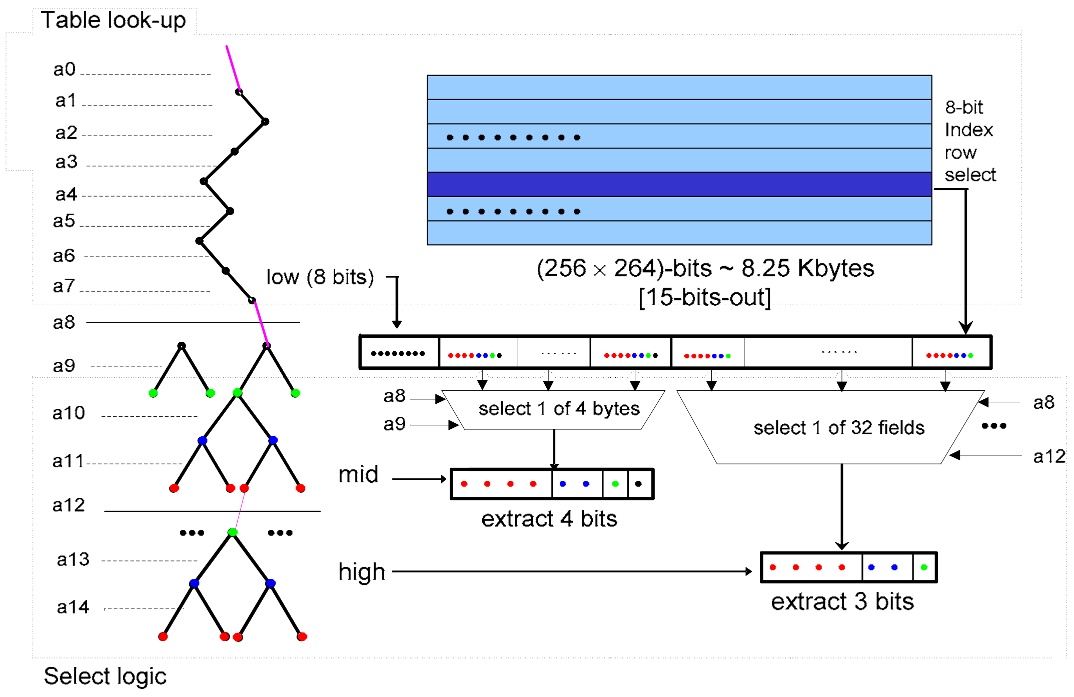


Figure 5.11 15-bit Table Lookup Architecture

The post-processing logic for a larger table lookup scheme is very simple. Since only the left children are stored and 15 bits are extracted from ROM, a one is padded to the *Least Significant bit* (LSb) to produce a 16-bit output. Also it is necessary to conditionally complement the result produced from the padding with the input value to produce the correct final result. Sixteen 2-bit-input XOR gates serve as conditional complement logic for this purpose where the corresponding bit from the result of the padding and the input are connected to the inputs of the XOR gates.

5.2.2.2 DLStable with smaller table

The ROM table size may be reduced by utilizing more properties of the DLS encoding. The inheritance property, one-to-one mapping property, sign symmetry, and normalization are all utilized for the smaller table design. For this approach, some pre-processing is required before table lookup and more complicated post-processing is also necessary. Pre-processing logic, the ROM structure, the post-processing logic and the hardware implementation are described in detail in the following.

Pre-processing consists of normalization and sign-bit extraction. Normalization is used to produce the p field of the DLS triple. It is accomplished by shifting right and counting the number of trailing zeros. In the worst case, 16 shifts are required. A divide and conquer approach is adopted in the implementation. At the beginning, shift the operand right 8 bits and the result determines whether to check the lower 8 bits or the higher 8 bits. Next, the selected 8-bit field from the previous step is shifted right 4 bits and result determines whether to check the lower 4 bits or the higher 4 bits. This procedure continues until the binary exponent p of the operand is obtained. The second operation is sign-bit extraction. The sign-bit is the third bit ($[a2]$) of the normalized operand. If the sign bit is asserted, it is required to conditionally complement the normalized operand. Since normalization (determined by $[a0]$) and sign-symmetry (determined by $[a2]$) are utilized in this step, the index for the address and select logic in next step are formed as $[a'14:a'3a'1]$ after conditional complementation.

The ROM structure and select logic are shown in Figure 5.12. The ROM represents a 3-level tree. The first level forms 128 rows where the lower 7-bits ($[a'8:a'3a'1]$) are used as address bits. Each row has 141 bits. 7 leftmost bits are selected directly as output in the row. The other 134 bits represent the second and third level tree. In the second level, sub-trees between level 7 and 8 are represented as a 6-bit field. $[a'9]$ and $[a'10:a'9]$ are applied to select one bit from the selected field respectively. Therefore, a total of 2 bits are extracted from the 6-bit field. In the third level, 16 sub-trees between level 7 and level 10 are formed as 16 bytes. $[a'12:a'9]$ are used to select one of 16 bytes. $[a'13]$ and $[a'14:a'13]$ are used to select two bits from the selected byte respectively, while the other two bits are extracted directly without selection. Therefore, a total of 4 bits are extracted from the selected byte. Finally, a 13-bit output is formed from the select logic.

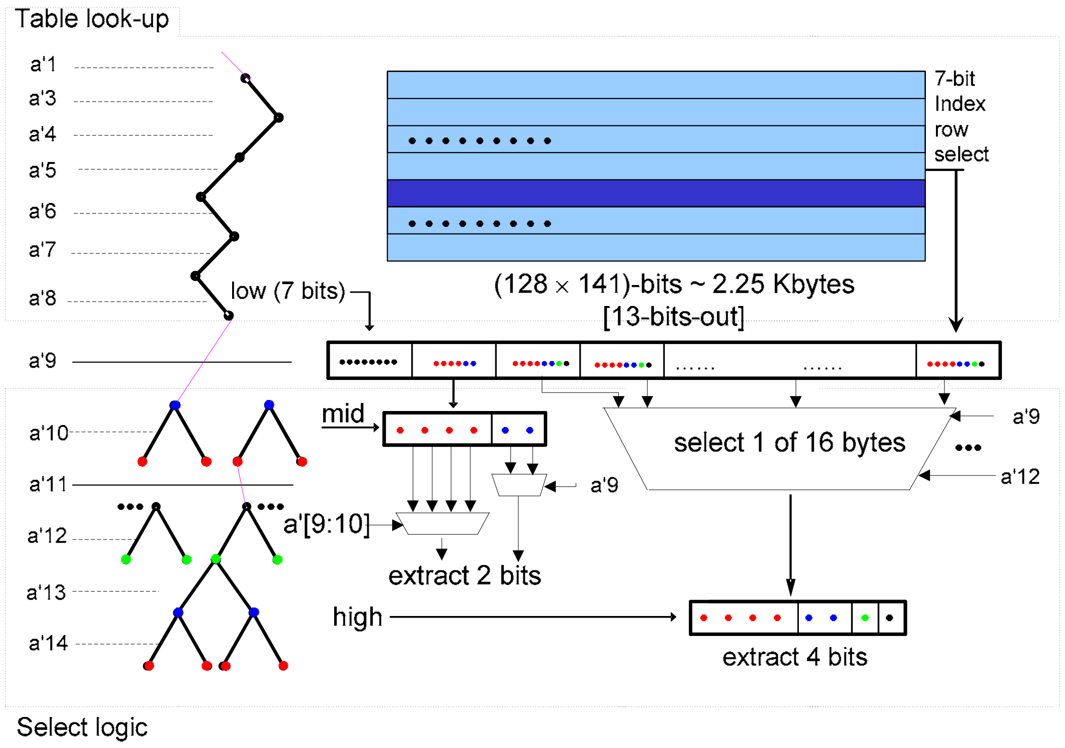


Figure 5.12 13-bit Table Lookup Architecture

Post-processing for the smaller table lookup scheme is more complex as compared to the larger table approach. Since normalization is performed in the pre-processing circuitry, de-normalization is necessary. All bits whose index is less than p are padded with zeros, while all bits whose index is larger than p are filled with lookup values. Since only the left sub-tree are stored, conditional complementation of the result from the output of de-normalization logic with the input word is necessary to obtain the final result. Sixteen 2-bit-input XOR gates are used for conditional complementation as described previously.

We implemented the integer powering operation circuits with the *DLStable* conversion/deconversion shown in Figure 5.11 and Figure 5.12 by describing them in a *Verilog* module and using the Synopsys tool set (Design Compiler and Physical

Compiler) based on a standard cell library obtained from the Synopsys tutorial files [Syn03]. More detailed information can be found in [LFTM06].

Table 5.4 shows the comparison between the two schemes for directed lookup table conversion for *k*=16. The ROM size is given in KB. The core area is the area of standard cell implementation for all other logic except the ROM. Both circuits have the same minimal clock period of 1.7ns but the larger table implementation requires one less cycle for post-processing. We do not compare the result of *DLStable* with *MM* and *DLSiter* since both *MM* and *DLSiter* are iterative computation based approaches where the latencies depend on the word size k.

Table 5.4 Comparison of Results for Two DLStable conversions

k=16	ROM size (KB)	Core area(μm^2)	Clock period (ns)	Latency (clock cycles)
DLStable with Larger Table	8.25	21011.2	1.70	2
DLStable with Smaller Table	2.25	19003.5	1.70	3

5.3 Verification Procedure and Results

In this section, the *DLSiter* and *DLStable* circuits described in the previous section are validated using the IDV system. The above designs include a memory unit (ROM), a datapath, control logic (counter and state controller), and some small components, such as *mux* and *xor*.

5.3.1 DLSiter Circuit Verification

In the *DLSiter* circuit, a small memory unit, a datapath, and a control logic block are present. For the *DLSiter* circuit, the properties that need to be verified are listed as follows:

a. Liveness property: *load*=1 → **AX**:2(**AF**(*busy*=0)): Along all the state trajectory paths in the future, there will be a state that *busy*=0 and it will last for at least the next two states. The signal *load* is asserted means two integers are loaded for calculating the powering operation. The signal *busy* is one while in the process of calculating the powering and zero when it is idle or when the calculation is complete. This property indicates that if integers are loaded for powering, the circuit must finish the calculation sometime in the future and will not get into an endless loop (busy will never be zero). This is a liveness property since it indicates that the circuits will eventually finish a powering operation.

b. Safety property: *load*=1 → **Next**(*busy*=1 && *current_state*=**Init**): If the signal *load* is asserted, the signal *busy* has to be one in the next cycle and the *current_state* has to be in the **Init** state. As indicated before, busy is asserted when the circuit is initiating the calculation process. This property ensures that if integers are loaded, the calculation is started in next cycle.

State Transition properties:

c. (*current_state* = **Init)** → **Next**(*current_state* = **Loop_DLG**): if the *current_state* is in state **Init,** then *current_state* has to in state **Loop_DLG** in next cycle. The same as the following several state transition properties.

d. (*current_state* = **Loop_DLG)** → **Next**(*current_state* = **Loop_ACC**)

e. (*current_state*=**Loop_ACC)** → **Next**(*current_state*=**Loop_EXP**)

f. (*current_state*=**Loop_EXP)** → **AF**(*current_state*=**Ready**)

g. *current_state*=**Loop_EXP** && Count<Bound**)** → **Next** (*current_state* = **Loop_ACC**)

h. Properties related to Memory: *RE*=1 $\wedge$ *addr* → *ROM_out* = **Next**(*M*[*addr*]): The signal RE indicates read enable for ROM. The property can be interpreted as: if read enable for ROM is on and a valid address is given, the output of ROM in next cycle should be the value stored in that address.

5.3.1.1 Partitioning

Design hierarchy is explored in this stage. A process-module graph which describes the hierarchy of the design is built. Each node in the graph represents a component/module/process and edge corresponding to the interconnections of these components. The process-module graph for *DLSiter* powering circuit is shown in Figure 5.13. This information is used for partitioning and system level functional simulation.

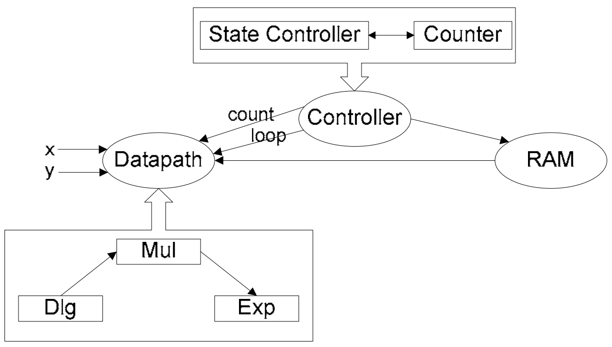


Figure 5.13 Graph Representation of Design Hierarchy

Initially a coarse-grain partition is explored and the tool continuously decreases the granularity of the partitions until the desired coverage goal for validation is reached. Given the above example, the top-level consists of three parts, a controller, a ROM, and a datapath. These three parts are extracted and supplied to complexity analyzer.

5.3.1.2 Complexity Analyzer

The complexity analyzer is responsible for:

a. Analyzing the properties that need to be checked and assigning them to appropriate verification tools. Complex properties specified in CTL are supplied as input to VIS while properties given in the trajectory formula format are supplied as input to STE. Given the above example, the liveness property (*a*) is quite complicated and unsuitable for STE and thus is supplied to VIS while properties *b*-*g* can be verified via either VIS or STE. In such a case, STE is preferred since STE has a larger capacity in terms of number of states variables in a design. Also, the property related to memory, *h*, is supplied to STE for formal verification since STE performs better for such components and the related properties usually can be expressed as trajectory formulas.

b. Selecting equivalence checking or simulation for datapaths. Simulation is more time consuming but can handle any design while an equivalence checking is faster for the "types" of designs that do not cause the memory explosion problem to occur. Currently, a rule based approach is used to select the appropriate method: 1) if the number of variables exceeds a threshold, or 2) if the circuit contains certain elements (i.e. multiplier), simulation is selected. Otherwise equivalence checking is utilized for datapaths. Even if an equivalence checking tool were selected, a timing threshold is set and simulation will be started if the equivalence checking tool cannot generate the result in given timing threshold.

Complexity analysis concludes that the controller is verified by property checking, the datapath is validated by simulation since the multiplier presented in the datapath causes memory problem due to the size of BDD. The read operation for the ROM can be verified with STE while the content of the ROM can be validated with equivalence checker.

5.3.1.3 Verification or Simulation Processing

After complexity analysis and partitioning are completed, the subcircuits and corresponding constraints are supplied to appropriate tools for verification and/or simulation. The results are shown in Table 5.5 for the *DLSiter* circuit. The results demonstrate runtime for the different tools. These results were obtained using a Pentium 4 PC with 512MB of Memory.

Table 5.5 Verification/Simulation Result

Component	Properties	Tools	Result	Time
Controller	a	VIS	T	1.2s
	b-g	STE	T	15s
Memory	h	STE	T	1.5s
	content	SMU-EQ	100%	18s
Data Path		Speed	10%	170s
Functional		Func. Sim	1%	230s

5.3.1.4 Coverage Analysis

Different coverage metrics have been proposed in different tool sets. We are currently focusing on vector coverage and plan to expand to other coverage metrics. The vector coverage of an output is based on:

a. the coverage rate of the components related to the output when they are simulated or verified at the block level

b. the contributions or importance of each component related to the output

c. the vector coverage of the system-level simulation and the interconnection error

Detailed descriptions for the above three points are demonstrated as follows. The first point is easy to understand. For each output of the design, not all components contribute such as the output signal *busy* which is only related to the component controller. If an output is related to the *n* components $C_1 \ldots C_n$, each component has a corresponding normalized coverage value $R_1 \ldots R_n$ when they are verified or simulated at the block level. R_i is one if the component has been fully verified or a value $R_i \in (0,1)$ that corresponds to the percentage of vectors simulated. The coverage for an output will increase as the coverage for each related component increases. However, even if all related components are fully verified ($R_i = 1$), the coverage for the output may not reach a perfect level of 100% since the interconnection may cause an error such as the case where two interconnections are reversely connected.

Also, not all components related to an output have the same contribution. For example, the output *z* is related to all three components and it is the direct output of the component datapath which in turn relates to the other two components. The contribution of the component C_i to the output is denoted as w_i. Currently, there are two ways to determine the value w_i: (1) designers assign the value, (2) automatic method based on the input distribution of the directly related component. An example is used to demonstrate the automated method. For the *DLSiter* circuit, the directly related component datapath has five inputs. Two of five inputs are from system inputs, two of them are from the component controller, and one of them from the component ROM. The contribution of each component is proportional to the input distribution. The contribution of the component datapath is $2/5$, the contribution of the component controller is $2/5$, and the contribution of the component controller is $1/5$.

The third point is related to the system level simulation and interconnection errors. Even if all related components are fully verified, the coverage for an output may not reach a perfect level of 100% since the interconnections may cause errors. System level simulation can be used to detect the presence of possible interconnection errors. The number of possible interconnection errors is related to the number of interconnections. The more interconnections, the more errors are possible. Also, the more system level simulation that is accomplished, the less possible interconnection errors present in a design. Based on above description, a graph of relationship between possible interconnection errors and the coverage of the system level simulation is shown in Figure 5.14.

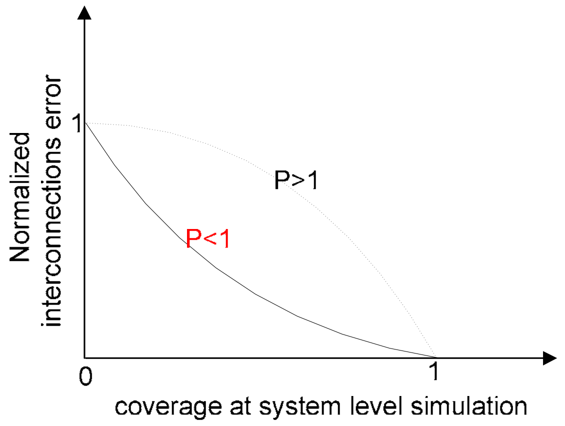


Figure 5.14 Possible Interconnection Errors vs. Coverage of System Level Simulation

Figure 5.14 shows the changes on coverage of the system level simulation and presence of possible interconnection errors. When no system level simulation is performed, all possible interconnection errors are normalized to one. The possible interconnection errors presented in the system decrease as the coverage of system level simulation increases. Various slopes in Figure 5.14 show the different decreasing rate of the possible interconnection errors which is referred to as dropping rate p. No interconnection error presents in a design once 100% coverage is reached at the system level simulation. Dropping rate p is related to the number of interconnections and the interconnection architecture. The smaller is p, the faster possible interconnection errors are detected via the system level simulation. The function in Figure 5.14 is referred to as the dropping function and denoted as $d(s,p)$ where the parameter s is the vector coverage at the system level simulation. Currently $d(s,p)$ is calculated as

$$d = (1 - s^{p})^{1/p} \qquad \text{Eq. 1}$$

where p is determined by $\frac{number\ of\ interconnections}{total\ inputs}$. Here only the number of interconnections is considered and the interconnection architecture is ignored.

In other words, $d(s,p)$ defines the distance between the perfect situation (no interconnection error) and the imperfect situation due to interconnection errors. $1-d(s,p)$ represents the confidence an output gains for interconnections with s system level simulation. If $d(s,p)$ is given, the coverage of an output can be calculated as $R\times(1-d(s,p))$ considering interconnection errors. In an optimistic analysis, interconnection error is ignored. Thus, the coverage for components is adjusted with and without considering interconnection errors. The adjusted coverage rate is referred to as R' and is calculated as an interval value

$$R' = [P_{low}, P_{high}] \qquad \text{Eq. 2}$$

where $P_{low} = R\times(1-d(s,p))$, $P_{high} = R$.

Based on the previous analysis, the total coverage rate for the output can be written as:

$$P_o = w_1 R_1 + w_2 R_2 + ... + w_n R'_n \qquad \text{Eq. 3}$$

Although this is a simple approach, it is our first attempt at automating coverage analysis. Detailed and in-depth research into automating coverage calculations is the research project being investigated by another Ph.D student in our group.

We now show how to use Eq. 3 to calculate the coverage of the given example, there are two outputs, *busy* and *z*. Among the three components, controller and ROM are fully verified and their coverage values are $R_c = 1$ and $R_m = 1$ respectively. While the component, datapath, is simulated with 10% coverage at the block level yielding $R_d = 0.1$.

The output *Busy*, is only related to the component *controller* and also the input *load*. The component controller totally controls the functionally of *Busy*, thus yielding $w_c = 1$. Applying Eq. 3, the coverage for the output *Busy* is

$$P_{busy} = w_c R'_c = w_c R_c = 1 \times 1 = 1 = 100\%$$

which indicates that the output *Busy* has been fully verified.

The output z is related to all three components. z is the direct output of the component datapath which accepts inputs from the system level input x and y, the output of the component ROM and the outputs of the component controller. With automatic methods based on input distribution, the contribution of the component controller is $w_c = 2/5 = 0.4$, the contribution of the component datapath is $w_d = 2/5 = 0.4$, and the contribution of the component ROM is $w_m = 1/5 = 0.2$. Next, the coverage for the component datapath is adjusted.

In the system-level simulation, 1% of the vectors for *x* and *y* have been simulated ($I_{xy} = 0.01$). Among five inputs for the component *datapath*, three of them are from interconnected component yielding $p = 3/5$. According to Eq. 1, the dropping function $d(s, p)$ is written as $d = (1 - s^{3/5})^{5/3}$. Substituting I_{xy} with s in the formula yields $d = 0.89$. This shows that with 1% of system simulation, 11% of confidence is obtained with the system interconnections.

The simulated vectors of the component *datapath* at the block level are 10%. Thus the adjusted coverage for the *datapath* is $R'_d = [0.011, 0.1]$ according to Eq. 2.

Then the coverage for the output z is calculated as

$$\begin{aligned} P_z &= w_c R_c + w_m R_m + w_n R'_d \\ &= 0.4 \times 1 + 0.2 \times 1 \times + 0.4 \times [0.011, 0.1] \\ &= [0.604, 0.640] \end{aligned}$$

The coverage of the output z with the initial partition is [0.604 0.640]. This coverage can be further improved by refinement.

5.3.1.5 Loop Back

When coverage is too low, IDV loops back to perform more partitioning over previously simulated components. In the above example, the datapath is the only component simulated, all others are completely verified. When IDV loops back and examines the component datapath, it contains three small modules, *conversion*, *multipler*, and *deconversion*. The component datapath can be further partitioned into three smaller modules and IDV can verify some of the smaller modules. The tools progress into a finer-grained partition of the design. Since all of the three modules cannot be formally verified, one way to improve the coverage is to increase the coverage rate of each module in the component *datapath* by simulation. After component-level simulation, the coverage of the datapath is recalculated. Using the automatic contribution methods described previously, equal contribution is assigned to the three components,

$w_{\text{mult}} = w_{\text{dlg}} = w_{\text{exp}} = 1/3$. All three components have been simulated at the same rate, $R_{\text{dlg}} = R_{\text{mul}} = R_{\text{exp}} = 0.3$. Thus applying these results, the new coverage value becomes:

$$\begin{aligned} R_d^2 &= w_{\text{mult}} R_{\text{mult}} + w_{\text{dlg}} R_{\text{dlg}} + w_{\text{exp}} R_{\text{exp}} \\ &= 1/3 \times (0.3 + 0.3 + 0.3) = 0.3 \end{aligned}$$

The coverage of the component, datapath, has been improved from 10% to 30% by refined partitioning. With the increased coverage for the component datapath, the adjusted coverage rate of the component datapath is calculated as [0.033, 0.30] according to Eq. 3. Given that all other parameters stay the same, the entire system coverage of the output z is calculated in the following

$$\begin{aligned} P_z &= w_c R_c + w_m R_m + w_n R_d' \\ &= 0.4 \times 1 + 0.2 \times 1 \times + 0.4 \times [0.033, 0.30] = [0.613, 0.720] \end{aligned}$$

5.3.2 DLStable Circuit Verification

The table-based design consists of a memory unit, a datapath for pre- and post-processing and multiplier, and a very simple control logic subcircuit. The following three properties need to be verified for the *DLStable* circuit:

a. Liveness property: *load*=1 → **AX**:2(**AF**(*busy*=0)): Along all state trajectory paths in the future, there will be a state such that busy=0 and it will be asserted for at least the next two states. The signal *load* is one means two integers are loaded for calculating the powering operation. The signal *busy* is asserted while in the process of calculating the result and zero when it is idle or the calculation is complete. This property indicates that if integers are loaded for powering, the circuit must finish the calculation sometime in the future and will not enter an endless loop (busy will never be zero). This is a liveness property since it checks that the circuit will not be deadlocked.

b. Safety properties: *load*=1 → **Next**(*busy*=1): If the signal *load* =1, the signal *busy* has to be asserted in the next cycle. As indicated before, busy is one when the circuit is initializing the calculation process. This property checks that if integers are loaded, the calculation should be started in next cycle.

c. Properties related to Memory: *RE*=1 $\wedge$ *addr* → *ROM_out* = **Next**(*M*[*addr*]): The signal RE indicates read enable for ROM. The property can be interpreted as: if read enable for ROM is on and a valid address is given, the output of ROM in next cycle should be the value stored in that address.

5.3.2.1 Partitioning

A PM graph is built which describes the hierarchy of the design. The PM graph for the *DLStable* powering circuit is shown in Figure 5.15.

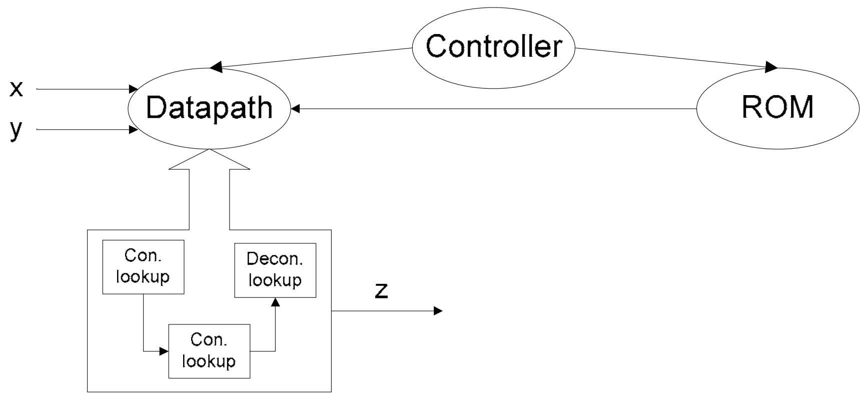


Figure 5.15 Graph Representation of Design Hierarchy for DLStable

Given the above example, the top-level consists of three parts, a controller, a ROM, and a datapath. These three parts are extracted and feed into complexity analyzer.

5.3.2.2 Complexity Analyzer

The liveness property (*a*) is quite complicated and unsuitable for STE, thus VIS is applied. Property *b* can be verified via STE.

The read operation for the ROM can be verified with STE while the content of the ROM can be validated with equivalence checker. The component datapath in *DLStable* circuit also cannot be formally verified since it contains multiplier. The controller is verified by property checking.

5.3.2.3 Verification or Simulation process

After the complexity analysis and partitioning are completed, the subcircuits and corresponding constraints are supplied to appropriate tools for verification and/or simulation. The results obtained using a Pentium 4 PC with 512MB of Memory are shown in Table 5.6 for *DLStable* circuit.

Table 5.6 Verification/Simulation result

Component	**Properties**	**Tools**	**Result**	**Time**
Controller	a-b	VIS	T	1.2s
Memory	c	STE	T	18s
	content	SMU-EQ	100%	200s
Data Path		Speed5	10%	78s
Functional		Func. Sim	1%	95s

5.3.2.4 Coverage Analysis

Based on the equations in the previous section, the coverage for outputs *busy* and *z* are calculated.

The components controller and ROM are fully verified and their coverage rates are $R_c = 1$ and $R_m = 1$ respectively. The component datapath is simulated with rate $R_d = 0.1$. In the system-level simulation, the input *load* is simulated in both phases ($I_{load} = 1$) and 1% of vectors for *x* and *y* have been simulated ($I_{xy} = 0.01$).

The output *Busy*, is only related to the component controller and also the input *load*. The component controller totally controls the functionally of *Busy* yielding $w_c = 1$. Applying Eq. 3, the coverage rate for the component *Busy* is

$$P_{busy} = w_c R'_c = w_c R_c = 1 \times 1 = 1 = 100\%$$

which indicates that the output *Busy* has been fully verified.

z is related to all three components and is directly the output of the component datapath which accepts four inputs; two of the inputs come directly from the system input, one input is the output of controller, and the other one is produced by the component ROM. Based on the automatic method, the contribution of the component datapath is $w_d = 2/4 = 0.5$. The other two components have equal contribution with $w_m = 0.25$ and $w_c = 0.25$ each. Next, the adjusted coverage rate for the component datapath is determined. Among the four inputs for the component datapath, two of them are from interconnected components yielding $p = 1/2$. Based on Eq. 1, the dropping function $d(s, p)$ is written as $d = (1 - s^{1/2})^2$. Substituting I_{xy} with s in the formula yields

$d = 0.81$. This shows that with a 1% of system coverage, 19% of confidence is obtained with the interconnections. The component datapath is simulated with 10% coverage. The adjusted coverage for the component datapath is interval [0.019, 0.10].

We now calculate the coverage for the output z.

$$\begin{aligned} P_z &= w_c R_c + w_m R_m + w_n R'_d \\ &= 0.25\times 1 + 0.25\times 1 \times + 0.5\times\left[0.019, 0.10\right] \\ &= [0.51, 0.55] \end{aligned}$$

5.3.2.5 Loop Back

IDV loops back to perform more partitioning over the previously simulated component datapath. In the component datapath, three small modules, *conversion table lookup*, *multipler*, and *deconversion table lookup* exist. The component datapath is further partitioned into three smaller modules and we verify some of the smaller modules. Two table lookup modules can be formally verified with equivalence checker SMU-EQ yielding $R_{\mathrm{dlg}} = R_{\mathrm{exp}} = 1$. After the component-level simulation, the coverage of the datapath is calculated using Eq. 3. Using the automatic contribution methods described previously, equal contribution is assigned to the three components, $w_{\mathrm{mult}} = w_{\mathrm{dlg}} = w_{\mathrm{exp}} = 1/3$. The component *multiplier* is simulated with 10% coverage yielding $R_{\mathrm{mul}} = 0.1$. Thus applying these results, the coverage value for the component datapath at the system level is:

$$\begin{aligned} R^2{}_d &= w_{\mathrm{mult}} R_{\mathrm{mult}} + w_{\mathrm{dlg}} R_{\mathrm{dlg}} + w_{\mathrm{exp}} R_{\mathrm{exp}} \\ &= 1/3 \times (1 + 0.1 + 1) = 0.70 \end{aligned}$$

The coverage of the component datapath has been improved from 10% to 70% by refined partitioning. With the increased coverage for the component datapath, the adjusted coverage for the component datapath is calculated as [0.133, 0.7] according to Eq. 2. All other parameters stay the same. Thus, the coverage of the output z is calculated in the following

$$\begin{aligned} P_z &= w_c R_c + w_m R_m + w_n R'_d \\ &= 0.25\times 1 + 0.25\times 1\times +0.5\times[0.0133, 0.70] = [0.567, 0.85] \end{aligned}$$

5.4 Summary

The purpose of the IDV is to provide one tool so that a designer can validate designs that can not be validated by any single tool. Through the above example, the IDV system is shown to have compatibility with different tools. Currently, the IDV system has not been fully automated and requires some amount of human interaction. The coverage calculation can be further improved by introducing more accurate models. Other members in the CAD methods research group are focusing their research on improvement of the coverage calculations and further automation of IDV.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

Formal verification plays an increasingly important role in design validation since evolving markets demand short design cycles while the increasing complexity of a modern design makes simulation coverage less and less complete.

Image/Pre-Image computation is a core algorithm in formal verification. BDD-based methods are usually faster but can exceed memory capacity for larger designs, limiting scalability. SAT solvers are less vulnerable to memory explosion but can be slow. We extended BDD-based image computation with a genetic algorithm and also presented a way of combining BDD and SAT approaches under one framework. A BDD-based approximation method was used to calculate the over- and under- approximation boundaries of reachable states. A SAT solver was used to find the remaining states. The SAT solver was enhanced by techniques referred to as “early detection” and “expansion” to find a satisfiable assignment containing more don’t cares. The experimental results showed that our approach can check more circuits than purely BDD-based symbolic model checking tools.

Formal verification itself cannot solely accomplish the validation task. Thus, we are motivated to combine different approaches to serve the purpose of validation of diverse digital designs. The IDV project resulted in the development of an integrated approach for design validation and takes advantage of current technology in the areas of simulation, and formal verification resulting in a practical validation engine with reasonable runtime. The focus in this approach is the circuit complexity analyzer and partitioning tool based upon design hierarchy. The IDV system also incorporates coverage analysis methods that compute the degree of design validation, a method for intelligently updating the complexity analyzer for further validation iterations, and integration of these techniques with existing simulation and formal verification techniques.

To demonstrate the capability of the IDV, two practical application circuits designed in our lab that are "hard cases" for validation were considered. The circuits were designed to implement a powering operation with two approaches: *DLSiter* and *DLStable*. The algorithms as well as hardware implementations were also outlined. IDV was then applied to validate the circuits. Since the designs are used to benchmark the algorithms being developed for integer operations, it is important to verify these designs.

6.2 Future Work

Currently, the IDV system has not been fully automated and requires some amount of human interaction. Ongoing work is in progress to make the IDV system automatic. Coverage analysis is also a major issue in IDV and in the verification research community in general. How one handles functional coverage and other coverage metrics should be explored further. Automatic feedback to improve the coverage is a very hot topic now. Several commercial EDA companies are working on this topic. Opening the IDV system to incorporate more tools is also one of our objectives.

At this stage, only formal verification and functional simulation are integrated in the IDV system. We also plan to extend IDV to include other types of simulations such as critical timing and fault simulation. ATPG is another important functionality to consider.

Besides the partitioning based on the design hierarchy, automatic partitioning a design is also desired. A methodology for automatically extracting controllers from an RTL-HDL specification is described in [LJ00]. This work introduces an algorithm for automatically separating the datapath and controller described at the RTL level by locating general patterns of FSMs in a PM graph representation of the design. In such a representation, the hierarchy is preserved and each module contains its own PM graph. Because a FSM's next-states always functionally depend on their current state, signals stemming from state-registers will loop back after some combinational paths have been traversed. Finding the FSMs in the HDL is based on finding such loops in the PM graph. Some loops that are found, however, may have a valid pattern topologically but not be part of the FSM. To deal with such instances, the checking of functional dependency follows the loop search to determine if the loop is a valid part of the FSM. The extraction process is divided into four phases, most of which are traversal procedures resembling a depth-first search of the PM graph. All steps are of linear complexity. This research demonstrates feasibility by focusing on the separation of a design into separate controller and datapath circuits. This can be used as a starting point.

One direction to expand IDV is to consider software verification, especially device driver verification. Some specific details that can be investigated include (1) automatic generation of Boolean programs along with a set of predicates from high-level languages such as C/C++ programs, (2) method of modeling a Boolean program in the form of a microprogramming controller, (3) applying suitable formal verification tools for memory to validate the microprogrammed-based model of a device driver.

We would also like to apply IDV for quantum logic and reversible logic. Quantum computing has many powerful applications that a classical computer cannot accomplish efficiently. Quantum hardware design remains an emerging field, but the work done thus far suggests that it will only be a matter of time before quantum devices are available to test Shor's and other's quantum algorithms. If this prediction comes to pass, quantum computers will emerge as computational devices that have profound implications in computing. Developing CAD tools for quantum hardware design will greatly benefit the quantum computing community.

System specification of a design is usually given in a natural language which is hard to translate into properties or assertions that can be supplied to formal verification tools directly. It is necessary to describe the system specification in a more rigorous and well-formed language in order to bridge the gap between a system specification and properties or assertions that formally verified. *Unified Modeling Language* (UML) has commonly applied to model software project and we adopt UML as tool for system specification.

Some preliminary work has been accomplished and the results are very promising [LOT06].

The SystemVerilog language evolved from the Verilog hardware description language as an industrial standard language to describe hardware design as well as to write assertions supporting the enormous task of verifying the correctness of a design. Incorporating SystemVerilog into the IDV system will greatly benefit the validation procedure.

REFERENCES

[AA+01]R. Alur, L. de Alfaro, T. A. Henzinger, M. Kang, C. M. Kirsch, R. Majumdar, F. Mang, and B. Y. Wang, "*jMocha: A Model Checking Tool that Exploits Design Structure*," in Proceedings of the IEEE International Conference on Software Engineering, 2001.

[ABH$^+$97] R. Alur, R. Brayton, T. Henzinger, S. Qadeer, and S. Rajamani, "*Partial order reduction in symbolic state space exploration*,", in Proceedings of 9th Conference on Computer Aided Verification, pp. 340-351, 1997.

[Ake78] S. B. Akers, "*Functional testing with binary decision diagrams*," in Eighth Annual Conf. Fault-Tolerant Computing, 1978, pp. 75–82.

[AK95] D. Appenzeller, and A. Kuehlmann, "F*ormal Verification of a PowerPc microprocessor*," in Proceedings of the IEEE International Conference on Computer Design, 99. 79-84, Oct. 1995.

[ARM99] ARM Limited, AMBA™ Specification (Rev 2.0), 1999, available at www.arm.com

[Ber81] C. Berman, "*On logic comparison*," in Proceedings of the 18th ACM/IEEE Design Automation Conference, pp. 854-861, Jun. 1981

[BCCZ99]A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "*Symbolic Model Checking using SAT procedures instead of BDDs*, " in Proceeding of the Design Automation Conference, Jun. 1999.

[BCL$^+$90] J. Burth, E. Clarke, D. Long, K. MaMillian, D. Dill, and L. Hwang, "*Symbolic model checking: 10^{20} states and beyond*," in IEEE Symposium on Logic in Computer Science, pp. 428-439, Jun. 1990.

[BCL91]J. R. Burch, E. M. Clarke, and D. E. Long, "S*ymbolic Model Checking with partitioned transition relations*", in Proceedings of the International Conference on Very Large Scale Integration, Edinburgh, Scotland, August 1991.

[Ben99] N. F. Benschop, "*Multiplier for the multiplication of at least two figures in an original format*" US Patent Nr. 5,923,888, July 13, 1999.

[Ber02] Reinaldo A. Bergamaschi, "*The A to Z of SoCs*," in Proceedings of the IEEE/ACM international conference on Computer-aided design, pp790-798, 2002

[BR00] T. Ball, and S. K. Rajamani, "*Boolean Programs:A Model and Process for Software Analysis*," Microsoft Research Publications, 2000

[BRB90] K. Brace, R. Rudell, and R. Bryant, "*Efficient implementation of a BDD package*," in Proc. Design Automation Conf., 1990, pp. 40–45.

[Bry86]R. Bryant, "*Graph-based algorithms for boolean function manipulation*," IEEE Trans. Computers, vol. 35, pp. 677–691, Aug. 1986.

[Bra$^+$web] R. Brayton et al.http://www-cad.eecs.berkeley.edu/Software/software.html

[Bra$^+$web*] R. Brayton et al. VIS: A system for verification and synthesis. http://vlsi.colorado.edu/vis/.

[BS98]J. R. Burch and V. Singhal, "*Tight Integration of Combinational Verification Methods*," in Proceedings of the International Conference on Computer Aided Design, pp. 570-576, 1998.

[BT89] C. Berman, and L. Trevillyan, "*Functional comparison of logic designs for VLSI circuits*," in Digest of Technical Papers of the IEEE International Conference on Computed-Aided Design, pp. 456-459, Nov. 1989.

[BW96] B. Bollig, and I. Wegener, "*Improving the variable ordering of OBDDs is NP-complete*," IEEE Transactions on Computers, vol. 45, no. 9, 1996, pp. 993-1002.

[CBM89] O. Coudert, C. Berthet, and J. Madre, "*Verification of sequential machines using Boolean functional vectors*," in IMEC-IFIP International Workshop in Applied Formal Methods for Correct VLSI design, pp. 111-128, Nov. 1989.

[CCJ$^+$01a] P. Chauhan, E. Clarke, S. Jha, J. Kukula, H. Veith, and D. Wang, "*Using combinatorial optimization methods for quantification scheduling*", In Proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME), September 2001.

[CCJ$^+$01b]P. Chauhan, E. Clarke, S. Jha, J. Kukula, T. Shiple, H. Veith, and D. Wang," *Nonlinear Quantification Scheduling in Image Computation*", In Proceedings of International Conference on Computer Aided Design (ICCAD), 2001.

[CCK03] P. Chauhan, E. M. Clarke, D. Kroening, "*Using SAT Based Image Computation for Reachability Analysis*," *Technical Report CMU-CS-03-151*, Carnegie Mellon University, School of Computer Science, July, 2003

[CE81] E. M. Clarke and E. A. Emerson, "*Design and synthesis of synchronization skeletons using branching time temporal logic*," In the Proceedings Workshop on Logics of programs, pp:52-71, Berlin, 1981, Springer-Verlag, LNCS 131.

[CEJS98] E. Clarke, E. Emerson, S. Jha, and A. Sistla, "*Symmetry reduction in model checking*, " in Proceeding of 10th Conference on Computer Aided Verification, pp. 147-158, 1999.

[Cho95] H.Cho, et. al. "Approximate Finite State Machine Traversal: Extensions and New Results," International Workshop on Logic Synthesis (IWLS'95).

[CI+95] A. Chandra et. al., "*AVPGEN – A Test Generator for Architecture Validation*," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol 3, No 2, June 1995

[CLR01] T. Cormen, C. Leiserson, R. Rivest, C. Stein, "Introduction to Algorithm", 2nd edition, The MIT Press, 2001, pp. 879-880.

[DLL62] M. Davis, G. Logemann and D. Loveland, "*A Machine Program for Theorem-Proving*", Communications of ACM, Vol. 5, No. 7, pp. 394-397, 1962.

[DO76] W. Donath, and H. Ofek, "*Automatic identification of equivalence points for Boolean logic verification*," IBM Technical Disclosure Bulletin, Vol. 18, No. 8, pp. 2700-2703, 1976

[DBG95] R. Drechsler, B. Becker, and N. Göckel, "*A genetic algorithm for variable ordering of OBDDs*", ", In Proceedings of the International Workshop on Logic Synthesis, Granlibakken, CA, May 1995.

[Dr98] R. Drechsler, *Evolutionary Algorithm for VLSI CAD*, Kluwer Academic Publication, 1998.

[Fi05] A. Fit-Florea, "Extending Hardware Support for Arithmetic Modulo 2^k," Dissertation, Dept. of Computer Science and Engineering, Southern Methodist University, 2005

[FM04] A. Fit-Florea, D. W. Matula, "*A Digit-Serial Algorithm for the Discrete Logarithm Modulo* 2^k ", Proc. ASAP, IEEE, 2004, pp. 236-246.

[FMT05a] A. Fit-Florea, D. W. Matula, M. A. Thornton, "Additive Bit-serial Algorithm for the Discrete Logarithm Modulo 2^k ", IEE Electronics Letters Jan. 2005, Vol. 41, No. 2, pp: 57-59.

[FMT05b] A. Fit-Florea, D. W. Matula, M. A. Thornton, "*Addition-Based Exponentiation Modulo* 2^k ", IEE Electronics Letters, Jan. 2005, Vol. 41, No. 2, pp: 56-57.

[GB94]D. Geist and I. Beer, "*Efficient Model Checking by automated ordering of transition relation partitions*", in Proceedings of Sixth Conference on Computer Aided Verification (CAV), vol. 818 of LNCS, Stanford, USA, 1994, pp. 299–310.

[GDP99] S. G. Govindaraju, D. L. Dill, and J. P. Bergmann, "*Improved Approximate Reachability using Auxiliary State Variables*," in Proceedings of the Design Automation Conference, June 1999, pp. 312-316.

[GL85] D.E. Goldberg, and R. Lingle. "*Alleles, loci, and the traveling salesman problem*," In Int'l Conference on Genetic Algorithms, 1985, pp. 154-159.

[GN02] E.Goldberg, Y.Novikov, "*BerkMin: a Fast and Robust SAT-Solver*," In Proceeding of DATE, 2002, pp. 142-149.

[GW94] P. Godefroid, and P. Wolper "*A partial approach to model checking*," Information and Control, pp. 305-326, 1994.

[GYA01] A. Gupta, Z. Yang, P. Ashar, L. Zhang, S. Malik. "*Partition-based decision heuristics for image computation using SAT and BDDs*". in Proceedings of the Intl. Conf. on Computer-Aided Design (ICCAD), 2001.

[HC98] Shi-Yu Huang, Kwang-Ting Chang, *Formal equivalence checking and design debugging*, Kluwer Academic Publishers, Boston, 1998

[HKWF02] S. Hazelhurst, G. Kamhi, O. Weissenberg and L. Fix. "*A Hybrid Verification Approach : Getting Deep into the Design*," in Proceedings of the Design Automation Conference. 2002.

[Hoa69] C. Hoare, "*An Axiomatic Basis for Computer Programming*," Communications of the ACM, 12:576-580, 1969

[HS01] S. Hassoun, and T. Sasao, *Logic Synthesis and Verification*, Kluwer Academic Publishers, 2001

[HS97] S. Hazelhurst and C-J Seger, "*Symbolic Trajectory Evaluation*." in T. Kropf, editor, Formal Hardware Verification, ch. 1, pp 3-78, Springer Verlag; New York, 1997.

[JKS02] H. Jin, A. Kuehlmann and F. Somenzi, "*Fine-Grain Conjunction Scheduling for Symbolic Reachability Analysis*," Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), 2002, pp 312-326.

[Jas04] Jason Andrews "*Co-Verification of Hardware and Software for ARM SoC Design,*" Newnes, 2004

[Jas:web] Jason Andrews "*Improving HW/SW Co-Verification with SoC Verification Matrix,*" http://www.techonline.com/community/tech_group/soc/tech_paper/36644

[JMH00] J. Jang, In-Ho Moon, G. Hachtel, "*Iterative Abstraction-based CTL Model Checking*, ", in Proceedings of the Conference on Design, Automation & Test In Europe, pp. 520-507, Mar., 2000.

[KG99] Christoph Kern, and Mark Greenstreet "*Formal verification in hardware design: a survey*," ACM Transactions on Design Automation of Electronic Systems, Vol. 4, Iss. 2, pp: 123-193, 1999.

[Knu81]D. Knuth, "*The Art of Computer Programming: Seminumerical Algorithms*" Addison Wesley, Vol. 2, 2nd Edition, 1981, pp: 441-466.

[KP03] Hyeong-Ju Kang, and In-Cheol Park. "*SAT-based unbounded symbolic model checking*," in Proceedings of Design Automation Conference (DAC'03), pp. 840-843, 2003.

[KS03]K. Kang and S.A. Szygenda, "*Accurate Logic Simulation by Overcoming the Unknown Value Propagation Problem*", Simulation Journal, Vol. 79, Issue2, February 2003.

[Lee59] C. Y. Lee, "*Representation of switching circuits by binary decision programs*," Bell System Techn. J., vol. 38, no. 4, pp. 985–999, June 1959.

[LFTM06] L Li, Alex Fit-Florea, M. A. Thornton, D. W. Matula "*Performance Evaluation of a Novel Table Lookup Method and Architecture for Integer Functions*," submitted to ASAP 2006.

[LFTM05] L Li, Alex Fit-Florea, M. A. Thornton, D. W. Matula "*Hardware Implementation of an Additive Bit-Serial Algorithm for the Discrete Logarithm Modulo 2^k*," IEEE Computer Society Annual Symposium on VLSI (ISVLSI), May. 2005, pp. 130-135.

[LHS:04] Bin Li, Michael S. Hsiao, and Shuo Sheng "*A novel SAT all-solutions solver for efficient preimage computation*," in Proceedings of Design Automation and Test in Europe (DATE) Conference, Feb., 2004, pp. 272-277.

[LJ00] C.-N. J. Liu and J.-Y. Jou, "*An Automatic Controller Extractor for HDL Descriptions at the RTL*," IEEE Design & Test of Computers, pp. 72-77, July-September 2000.

[LPJ$^+$96] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi "*Tearing based abstraction for CTL model checking*," in Proceedings of the IEEE International Conference on Computed-Aided Design, pp. 76-81, Nov. 1996

[LOT06] L. Li, P. Ongsakorn, M. Thornton, F. Coyle, S. Syzgenda "*Automatic High Level Assertion Generation and Synthesis for Embedded System Design*," under preparation.

[LT05] L Li, M. A. Thornton, "*BDD-Based Conjunctive Decomposition Using a Genetic Algorithm and Dependent Variable Affinity*," in Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), 2005, pp. 277-280

[LTM06] L Li, M. A. Thornton, D. W. Matula "*A Fast algorithm for the integer powering operation*," to be appear in GLSVLSI 2006

[LTS04] L Li, M. A. Thornton, S. Syzgenda "*A Genetic Approach for Conjunction Scheduling in Symbolic Equivalence Checking*," IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Feb. 2004, pp. 32-36

[LTS05] L Li, M. A. Thornton, S. Syzgenda "*Combining Simulation and Formal Verification for Integrated Circuit Design Validation*," Proc. 9th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI), 2005, pp. 92-97. (**best paper in the session: Simulation and Applications of Modeling**)

[LTS06] L Li, M. A. Thornton, S. Syzgenda "*Combining BDD and SAT for Pre-image Computation*," In preparation.

[Mal$^+$web] S. Malik, et. al http://www.princeton.edu/~chaff/

[Mar$^+$web] João Marques, http://sat.inesc-id.pt/~jpms/grasp/

[MFT05] D. W. Matula, A. Fit-Florea, M. A. Thornton, "*Table Loopup Structures for Multiplicative Inverses Modulo* 2^k ", 17th Symp. Comp.Arith., June 27-29, 2005, pp. 130-135.

[MS96a] João P. Marques-Silva and Karem A. Sakallah, "G*RASP -- A New Search Algorithm for Satisfiability*," in Proceedings of IEEE/ACM International Conference on Computer-Aided Design, November 1996.

[MS96b] João P. Marques-Silva and Karem A. Sakallah, "*Conflict Analysis in Search Algorithms for Propositional Satisfiability*," in Proceedings of the IEEE International Conference on Tools with Artificial Intelligence, November 1996.

[Mcm02]Ken McMillan. "*Applying SAT methods in unbounded symbolic model checking*," in Proceedings of the International Conference on Computer-Aided Veri_cation (CAV'02), vol. 2404, pages 250-264, 2002.

[MHB98] G. Manku, R. Hojati, and R. Brayton, "*Structural Symmetry and in model checking*," in Proceeding of 10th Conference on Computer Aided Verification, pp. 159-171, 1999.

[MS00] I. Moon and F. Somenzi, "*Border-block triangular form and conjunction schedule in image computation*", in Proceedings of the Formal Methods in Computer Aided Design (FMCAD), vol. 1954 of LNCS, November 2000, pp. 73–90.

[MTS04] R. Marcyzynski, M.A. Thornton, and S.A. Szygenda, "*Test Vector Generation and Classification Using Symbolic FSM Traversals*," International Symposium on Circuits and Systems, pp. V-309 – V-312, May 2004.

[Par00]B. Parhami, "*Computer Arithmetic Algorithms and Hardware Designs*", Oxford University Press, 2000, pp. 383-384.

[Phi01]Lars Philipson, "*Survey compares formal verification tools*", EE Design, Nov. 2001, URL: http://www.eedesign.com/article/showArticle.jhtml?articleId=17407560

[PIWC04] G. Parthasarathy, M. Iyer, Li Wang, and K.Cheng, "*Efficient Reachability Analysis Using Sequential SAT*", in Proceeding of Asia and South Pacific Design Automation Conference(ASP-DAC), Jan, 2004.

[PMV98]V. Paruthi, N. Mansouri and R. Vemuri, "*Automatic Data Path Abstraction for Verification of Large Scale Designs*," in Proceedings of the IEEE International Conference on Computer Design, pp. 192-194, 1998.

[PM04] Carl Pixley, Sharad Malik, "*Exploring synergies for design verification*," IEEE design and test of computers, pp: 461-463, Nov.-Dec., 2004

[Rot77] J.Roth, "*Hardware verification*," IEEE Transactions on Computers, Vol. C-26, pp. 1292-1294, Dec. 1977

[RAP$^+$95] R. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. Brayton, "*Efficient BDD algorithms for FSM synthesis and verification*," in Proceedings of International Workshop on Logic Synthesis, Lake Tahoe, 1995.

[RL:web] K. Rustan and M. Leino, "A SAT Characterization of Boolean-program Correctness," Microsoft Research, SLAM Project web page.

[Som$^+$web] F. Somenzi et al. CUDD: University of Colorado Decision Diagram Package. http://vlsi.colorado.edu/~fabio/CUDD/.

[SB95] Carl Seger, and Randy Bryant, "*Formal verification by symbolic evaluation of partially ordered trajectories*," Formal Methods System Design, Vol. 6, Iss. 2, pp: 147-189, 1995.

[SD02] K. Shimizu and D. L. Dill, "*Using Formal Specifications for Functional Validation of Hardware Designs*," IEEE Design & Test of Computers, pp. 96-106, July-August 2002.

[SGC+05] J.E. Stine, J. Grad, I. Castellanos, J. Blank, V. Dave, M. Prakash, N. Illiev, N. Jachimiec, "*A Framework for High-Level Synthesis of System-on-Chip Designs*," In Proceedings of IEEE International Conference on Microelectronic Systems Education, 2005, pages 67-68.

[SH03] Shuo Sheng and Michael Hsiao. "*Efficient preimage computation using a novel success-driven ATPG*," in Proceedings of Design Automation and Test in Europe (DATE'03), 2003.

[ST67] N. S. Szabo, R. I. Tanaka, "*Residue arithmetic and its applications to computer technology*", McGraw-Hill Book Company, 1967.

[Syn03] Synopsys Design/physical Compiler Student Guide. 2003.

[Szy90] S. A. Szygenda, "*The Simulation Automation System, Using Automatic Program generation, for Hierarchical Digital Simulation Systems*," in Proceedings of the European Simulation Conference, 1990

[Tar55] A. Tarski, "*A lattice-theoretical fixpoint theorem and its applications*," *Pacific J. Math.*, 5:285-309, 1955

[TD01] M. A. Thornton, and R. Drechsler, "*Evolutionary Algorithm Approach for Symbolic FSM Traversals*," IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), August 26-28, 2001, pp. 506-509.

[TK01] S. Tasiran and K. Keutzer, "*Coverage Metrics for Functional Validation of Hardware Designs*," IEEE Design & Test of Computers, pp. 36-45, July-August 2001.

[TSL$^+$90] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli, "*Implicit enumeration of finite state machines using BDDs*", in Proceedings of the

International Conference on Computer Aided Design (ICCAD), November 1990, pp. 130–133.

[Yan99]B. Yang, *Optimizing Model Checking Based on BDD Characterization*, PhD thesis, Carnegie Mellon University, May 1999.