AUTOMATIC CONSTRAINT BASED TEST GENERATION USING BEHAVIORAL HDL MODELS

A Project Report

Submitted in partial fullfilment of the
requirements for the award of the degree
of

Bachelor of Technology

by

Siva Kumar Sastry Hari
CS03B022

Under the guidance of

Dr. V. Kamakoti

Department of Computer Science and Engineering
Indian Institute of Technology Madras

May 2007

CERTIFICATE

This is to certify that the thesis titled **Automatic Constraint Based Test Generation Using Behavioral HDL Models** submitted by **Mr. Siva Kumar Sastry Hari** to the Department of Computer Science and Engineering at Indian Institute of Technology Madras for the award of the degree of Bachelor of Technology is a bona-fide record of project work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. V. Kamakoti
Associate Professor
Dept. of Computer
Science and Engg.
IIT Madras
Chennai, India

Date: 14th May 2007
Place: Chennai-36

Acknowledgements

Many people have contributed to my research and the marvelous experience at IIT Madras. First of all, I would like to thank my advisor, Dr. V. Kamakoti for his guidance, support and encouragement that helped me achieve my goals. I convey my gratitude to my project guide for being much more than just a guide. He was the source of inspiration and enthusiasm to me.

"I would thank you from the bottom of my heart, but for you my heart has no bottom"

I thank the Head of the Department, Prof. Timothy A. Gonsalves, for providing excellent facilities with which we were able to work effectively.

I am thankful to Dr. Shankar Balachandran who has always supported me by providing inputs whenever needed. Most of our interactions (technical/non-technical) have turned in to interesting discussions. I learnt a lot from his course, CAD for VLSI and even utilized the code developed for the assignment as a test bench for my project work.

I thank Dr. N. S. Narayanaswamy for monitoring my progress throughout my undergraduate studies. I would like to thank IIT Madras for providing me with the incredible facilities that not only made me achieve my career goals but also made me learn a lot about the world outside academics. The campus provided me with the cherishable memories that are never forgettable. My stay at IIT Madras couldn't have been better without the support of my friends and the faculty members. It gives me immense pleasure to acknowledge their support.

I thank Aniket Ray for being my group partner for various course assignments during my undergraduate stay and making all our meetings fruitful, enjoyable and memorable. I thank Noor Mahammad for making my stay in RISE Lab exciting by his encouraging discussions. I would like to thank Mr. Nagalingam for sharing his knowledge and extending support on various CAD tools.

I would especially like to thank Vishnu Vardhan Konda with whom I worked closely during my final year project. Working with him was fun and I enjoyed most of our night-outs in the lab. I thank K. Najeeb with whom I worked in the later part of my project. His collaboration made the project more enjoyable and exciting.

I would like to thank Intel Corporation for collaborating with us and providing a lot of inputs form time to time. I thank Maneparambil Kailasnath and Dr. Vivekananda M Vedula from Intel, without whose support the project wouldn't have been more meaningful and successful.

Last but not the least, I would like to thank my parents and especially my elder brother, Sreepati Hari, for making me the person I am today.

Siva Kumar Sastry Hari

Abstract

A major challenge in the success of the semiconductor industry is the ever-growing complexity of the chips. The increasing demand for more functionality per chip is leading to complex circuit designs. With the emergence of huge number of digital chip designers, time to market has become the key to success. A major portion of the design process, approximately 70% is consumed in the Verification and Testing process. With the emergence of the complex high-performance microprocessors, Functional Test Generation (FTG) has become a crucial step in the design process. The major obstacle in the growth of the semiconductor industry is to reduce the time taken in verification. This project focuses on an efficient and scalable approach for Functional Test Generation especially in the processor environment.

The rapid growth in the count of transistors on the chips has made the gate-level approaches inefficient and infeasible. To arrive at more scalable solutions, the FTG is attempted at the highest level of abstraction, typically using behavioral Hardware Description Language (HDL) models. A major advantage of handling the design at the high level is that the tool can use this behavioral information to efficiently generate input patterns, and in far less time. Working at behavioral level ensures that the complexity of the FTG is proportional to the code size, rather than being proportional to the number of gates.

Commonly, the random FTG tools are used. Usually many subtle conditions escape the random testing and hence there is a need for directed FTG. Constraint-based test generation is a well studied Directed Behavioral level Functional Test Generation (DBFTG) paradigm. The paradigm involves conversion of a given circuit model

into a set of constraints and employing constraint solvers to generate tests for it. However, automatic extraction of constraints from a given behavioral HDL model remained a challenging open problem. This project proposes an approach for automatic extraction of word-level model constraints from the behavioral Verilog description. The scenarios to be tested are also expressed as constraints and are simple to specify as well, unlike gate-level approaches. The model and the scenario constraints are solved together using an integer solver to arrive at the necessary functional test.

The effectiveness of the approach is demonstrated by automatically generating the constraint model for the 16-bit DLX-architecture from its Verilog based behavioral model. Experimental results that automatically generate test vectors, in this case assembly code of the DLX architecture, for high level scenarios spanning over multiple time frames show that the approach is far better than the existing techniques.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction and Motivation

1.1 Introduction

The complexity of the chips are increasing rapidly and the number of transistors on an integrated circuit is doubling every 18 months (Moore's Law). The emergence of Computer Aided Design (CAD) Tools for VLSI and the Hardware Description Languages (HDL) have worsen the scenario. The million-gate designs have grown in to billion-gate deigns. In the recent years, a large number of companies has emerged in field of VLSI leading to a very stiff competition. A chip today becomes outdated within 1 or 2 years of its release, sometimes, even months. The competition and the speed at which the industry is growing has made time-to-market a crucial factor for success. Which in turn demands to reduce the time spent at each stage of the design process.

A major proportion of the total development time is spent in the testing and verification of the design. As the design time increases for a company, it tend to loose revenue as other chip makers overtake. More over the process of FTG is mostly manual which not only makes the whole design process slower but also consumes a lot of revenue. Even after spending a lot of revenue and time on testing the process is usually incomplete which leaves behind a risk of having some bug in the chip. Ones the chip is in the market and the bug is caught then it causes a huge amount of loss to the company not only in terms revenue but also in reputation. The most famous real life example is the *Pentium FDIV bug*.

Hence, the Functional Test Generation (DFTG) becomes a very crucial step in the design process.

The key focus of the present day FTG approach is to generate test vectors that can verify the complex functionality and importantly the interaction between multiple design units. The current practice is to generate millions of random test vector sets. The random test generation does not guarantee the coverage of all the functionalities, especially in the case of complex designs. This necessitates directed tests, that shall cover the corner cases not covered by the random tests. The more important problem that should be addressed at this point is that the corner cases are no more bit values on specified wires or nets. These corner cases are much more abstract scenarios spanning across multiple clock cycles. For example, in a complex microprocessor a typical corner case can be to generate an instruction which accesses a memory and a register such that the register access results in a data hazard and the memory access results in a page miss. Specifying such higher-level corner cases to the test generator becomes extremely cumbersome at lower levels of abstraction. In addition, the crucial bottleneck with existing test generation tools is their scalability with larger designs. It is well-studied and reported in the literature that for a tool to be scalable with larger designs, it is important to handle the design at higher levels of abstraction, typically at the behavioral level. This explains the need for a Directed Behavioral level Functional Test Generation (DBFTG) tool.

Unlike many design tasks like logic synthesis or place and route that have been automated with sophisticated tools, functional verification has remained largely as a manual process. Languages like Verilog [1] enable the designer to specify the model at behavioral level. The Hardware Verification Languages [2] (HVLs) are capable of working in synchrony with the behavioral level models. This can automate generation of test benches. However, configuring the HVL environment as a DBFTG requires significant amount of manual effort. In order to reduce the manual intervention, there is a need for a tool that is capable of generating test vectors given a behavioral level description of the Design Under Test (DUT) and a higher-level test specification. We propose a DBFTG technique that reduces the human interaction to a large extent.

1.2 Overview

The proposed methodology accepts as input a behavioral level Verilog model and converts it into an Assignment Decision Diagram (ADD) [3] based data structure called the Assign-Always-Module (A^2M) graph preserving the hierarchy present the Verilog model. The A^2M graph is converted into a set of integer constraints. These constraints are called as the *model constraints*. The salient feature of the tool is that the above steps are *fully automated*, reducing the human effort significantly. The *scenario* for which a test has to be generated is also modeled as constraints, namely, the *scenario constraints*. Both the *model* and *scenario* constraints are together solved using a Integer Constraint solver to generate the required test. The design flow of the tool is shown in the Figure 1.1.

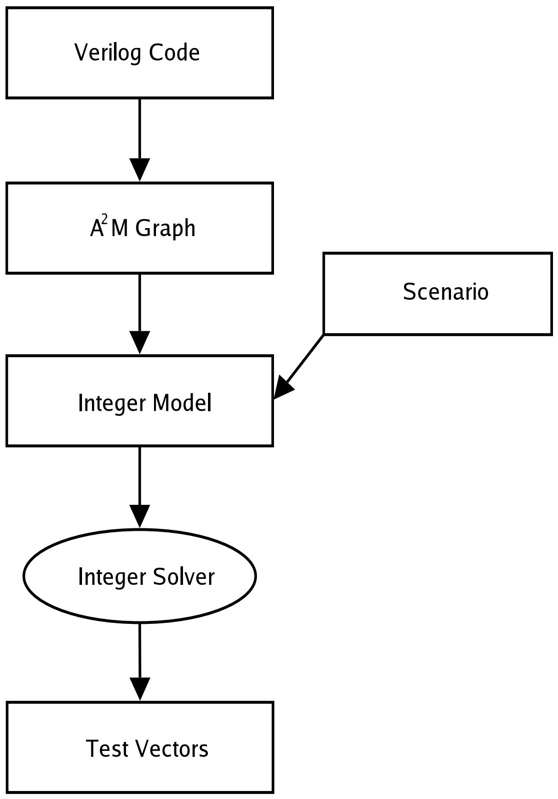

Figure 1.1: Tool Flow

The major advantage that over the conventional methods is that the scenario specification becomes extremely easy at high level. Another advantage is that the *scenario* may span across several clock cycles or time-frames. Important point to note is that the size of the A^2M graph grows linearly with that of the input Verilog code. The number of model constraints generated, in turn, also grows linearly with the size of the A^2M graph. The above observations imply that the proposed methodology is scalable with increasing design size.

1.3 Case Study: DLX Architecture

The proposed methodology is employed on the well-known DLX architecture [4]. The DLX architecture has a 5-stage pipeline. These stages are Instruction Fetch, Instruction Decode/Register Fetch, Execute/Address calculation, Memory access and Write back. The architecture provides a good platform to test the proposed methodology as it offers a variety of interesting and complex micro-architectural features and scenarios such as hazards, ALU operations, etc. The architecture is shown in Figure 1.2.

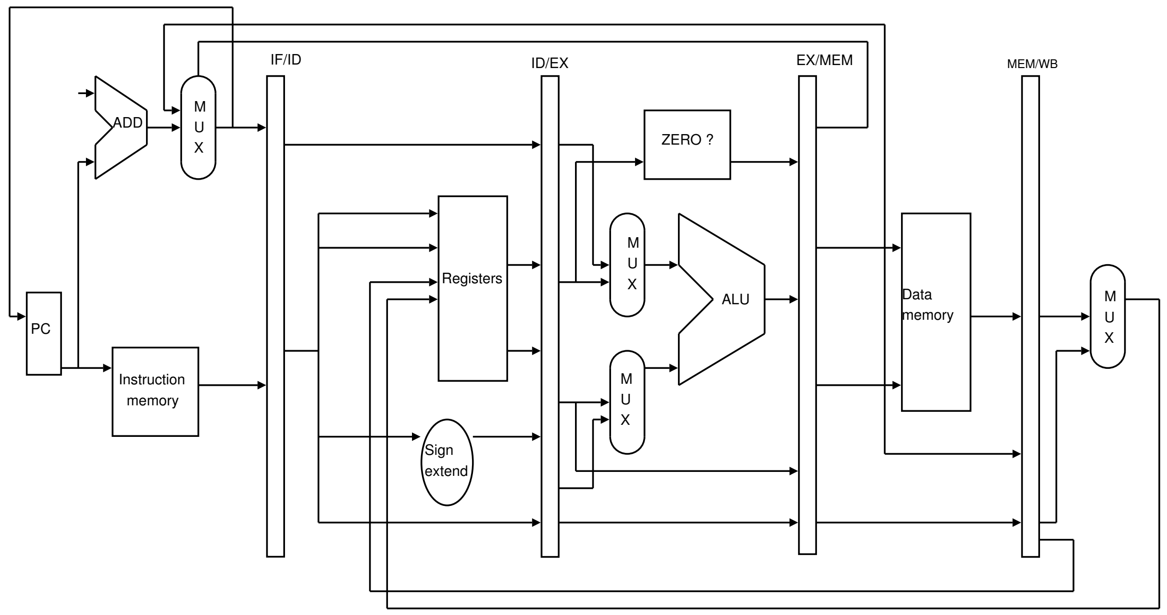

Figure 1.2: DLX Architecture

The code snaps that are presented throughout the following chapters for explanation of different concepts are taken from the Verilog description of the DLX architecture.

1.4 Organization of the thesis

In **Chapter 2** a survey of the work done in this area is presented and the advantages of the proposed technique is highlighted. Our contribution to the field of testing is also explained. **Chapter 3** discusses the approach of constraint model generation for FTG in details. **Chapter 4** presents the experimentation results and the observations based on the them. **Chapter 5** concludes by presenting the salient features of the approach and discusses the directions of future research.

Chapter 2

Previous Work and Contribution

In this chapter we present an overview of the previous work and highlight the contributions of the proposed technique.

2.1 Previous Work

Previous work in this area can be broadly classified into two, namely, the conventional Automatic Test Pattern Generation (ATPG) [5] based approach and the Constraint based approach.

2.1.1 ATPG Approaches

Classical ATPG methods [5] work at gate-level representations and hence exhibit less scalability with increasing design size. This results in a large requirement of both computing time and memory resources to generate tests even for moderately sized circuits. Some of the recent ATPG-based techniques consider behavioral level design models as inputs. DBFTG techniques using ATPG are reported in [3, 6]. These papers also present a comprehensive survey of the techniques reported prior to them. The frame-work presented in [6] converts a given HDL representation to an Assignment Decision Diagram (ADD) representation and uses a modified ATPG algorithm, called the RTL ATPG to create functional tests. However, no results were reported for processor level circuits. The other issue is to express the test scenario to the tool. It is not clear from [6] of how to specify a scenario that spans across multiple time-frames which becomes a prominent issue when one wants to test complex high-level functionalities.

2.1.2 Constraint-based FTG

Constraint-based FTG approaches model the given circuit description as constraints [7, 8, 9, 10]. The constraints are solved using constraint solvers to generate the required functional tests. A program slicing based technique that extract constraints from a given behavioral description to generate tests for a given submodule of the circuit is presented in [11]. The constraint based approach is also widely used for generating software based self tests for microprocessors [12, 13, 14, 15].

There are two approaches to constraint modeling, namely, (1) to model using Boolean constraints (that use a Boolean Satisfiability (SAT) solver) and (2) to model using Integer constraints (that uses an Integer solver).

The Boolean Satisfiability (SAT) problem is a well-known constraint satisfaction problem with many applications in the field of VLSI Computer-Aided Design. A Boolean circuit can be encoded as a satisfiability equivalent Conjunctive Normal Form (CNF) formula using the method of [16]. SAT has been used in the context of formal verification by several researchers to verify complex hardware designs with significant improvement over ATPG-based approaches. However, practical gate-level circuits can be quite large, dealing with substantially large CNF formulas results in unacceptable runtime. Another approach is to solve the Boolean satisfiability problems based on Binary Decision Diagrams (BDDs) [17, 18]. The major limitation is that BDD-based methods like BSAT requires excessive time/memory to create BDDs for the test circuits. A detailed survey of recent advances in SAT-based formal verification is presented in [19]. Regular expression algebra based techniques for identifying relevant paths that can be sensitized by test vectors in a precomputed test set to test modules in a circuit model in which the control and the data paths are separated are presented in [20, 21]. The above technique is efficient only for circuits where the data and control paths are separate and which have Design For Testability (DFT) [5] support. A SAT-based ATPG technique for non-separable Control-Datapath circuits is presented in [22].

Techniques that employ word-level reasoning for FTG are also reported in the literature. The technique presented in [23] proposes a functional vector generation method for RTL models using word-level constraint logic programming based on assertions. Constraint propagation techniques across different domains, that is, (both arithmetic and boolean domains) have been explored to generate functional tests and high level ATPG vectors on HDL descriptions [24, 25, 26]. A hybrid ATPG based modular arithmetic constraint solving technique for assertion checking is proposed in [27]. A Hybrid Satisfiability approach, HSAT, to generate functional test vectors for RTL design is proposed in [28]. This approach was unified in [29] that yields a single Linear Constraint Problem instance. The approach presented in [29] is not scalable for SAT instances with large portions of sequential logic. The ideas discussed in [30, 31, 32, 33] use constraint solvers to generate tests for functional verification of higher-level architectural features.

One of the recent results reported in literature dealing with Micro-architecture verification [34] concentrates on using a Generic-Test-Plan (GTP) approach to generate tests. However, the paper does not explain how to generate directed tests that verify specific micro-architectural features. Employing genetic algorithms to evolve efficient test benches for behavioral level circuit models is studied in [35, 36]. A unified framework for functional verification of processors using constraint solvers was proposed in [37]. This framework did not address automatic constraint generation from circuit models.

2.2 Contribution

We proposed a fully-automated framework to generate directed tests for functional verification of any digital system, specifically microprocessors. The proposed methodology accepts a behavioral level Verilog model as input and converts it into an Assignment Decision Diagram (ADD) [3, 38] based data structure called the Assign-Always-Module (A^2M) graph. The (A^2M) was then passed through two stages of optimization. The optimized A^2M graph is then converted into an Integer model. We call the constraints in the model as the *model constraints*. The salient feature of the methodology is that the above steps are fully automated.

The scenario for which a test has to be generated is also modeled as constraints, namely, the *scenario constraints*. Both the model and scenario constraints are solved together as one model using a integer constraint solver, ILOG [39], to generate the required test. Importantly, the scenario may span across several clock cycles or time-frames. The salient features of the proposed technique are outlined below:

1. *The input to the proposed technique is a behavioral level HDL model*

   The behavioral level models are not only lesser in size but also expresses the high-level system functionality more explicitly unlike their gate level representation. This is crucial for functional level test generation as the specification of complex scenarios becomes easier.

2. *Automatic generation of the constraint model from the behavioral model*

   Extraction of constraints from a behavioral or RTL description is a challenging problem [22]. The proposed methodology does solve the problem comprehensively and the conversion from Verilog design to A^2M graph to Integer model is fully automated.

3. *Word-level constraints in contrast to bit-level constraints*

   The proposed methodology deals with word-level constraints and employs an Integer solver. This has the following advantages

   (a) RTL models contain variables of varying width, some of which are used in arithmetic operations that have a regular structure and meaning in the integer domain. Test generation approaches that use CNF-SAT and ATPG operate at the Boolean level and require that the RTL model to be flattened. This results in loss of the regularity in the arithmetic operations that could have been leveraged while reasoning about these operations for test generation. Thus, the word-level reasoning is best suited for the DBFTG.

   (b) The integer solvers are better suited for word-level reasoning than the SAT solvers. While the control portion of the design lends itself well to

Boolean-level reasoning, word-level reasoning can be used where ever possible to improve the efficiency of the overall test generation. The intuition behind improved performance is that the Boolean SAT and conventional ATPG techniques are NP-complete in the number of bit-level variables, whereas, word-level solving complexity is NP-complete in the number of word-level variables, which grow less dramatically with increasing design functionality.

4. *A single constraint model with unified control and data paths*

   Unlike some of the previous approaches, the proposed technique does not demand a separation between control and data paths.

5. *Handling sequential designs*

   Most integer level constraint solvers are not built to handle sequential behavior of circuits. A set of techniques have been devised that converts sequential RTL description to constraint models.

6. *Scalability with increasing design size*

   The size of the A^2M graph grows linearly with that of the input Verilog code. The number of model constraints generated grows linearly with the size of the underlying (A^2M) graph. The above two observations imply that the proposed technique is more scalable with increasing design size as compared to the bit-level approaches.

7. *Verifying complex scenarios*

   The proposed technique generates test vectors automatically that can verify specific module interaction issues that can go over multiple time-frames. The major advantage we have over bit-level approaches is that the specification of complex scenarios becomes extremely simple. Hence, testing high level functionalities also becomes feasible which was a nightmare for the bit-level approaches.

Chapter 3

Constraint Model Generation

This chapter explains the entire design flow, from the conversion of Verilog file to A^2M graph to the functional test generation. The described flow automatically generates the test vectors that for a given *scenario* and for an input Verilog code. The approach has three major stages as listed below:

1. A^2M graph generation from the input behavioral HDL description;

2. Constraint model generation from the A^2M graph; and

3. Modeling scenario constraints and the functional test generation.

The complete tool flow is shown in Figure 3.1. A detailed explanation of each stage of the approach is presented in the rest of this chapter.

The proposed approach is implemented for Verilog based HDL models. However, the tool can be extended to other Hardware Description Languages like VHDL. The standard ILOG [39] constraint solver was used for experimentation. Hence, the ILOG-type syntax is used to illustrate the examples in this chapter.

3.1 A^2M graph generation

A^2M graph generation is the first step among the several stages of the entire design flow, Figure 3.1. This stage parses the input Verilog code and hierarchically creates the A^2M Graph. A synthesizable Verilog code that describes the behavior of a circuit has three major structures, namely, the `Always block`, the `Assign`

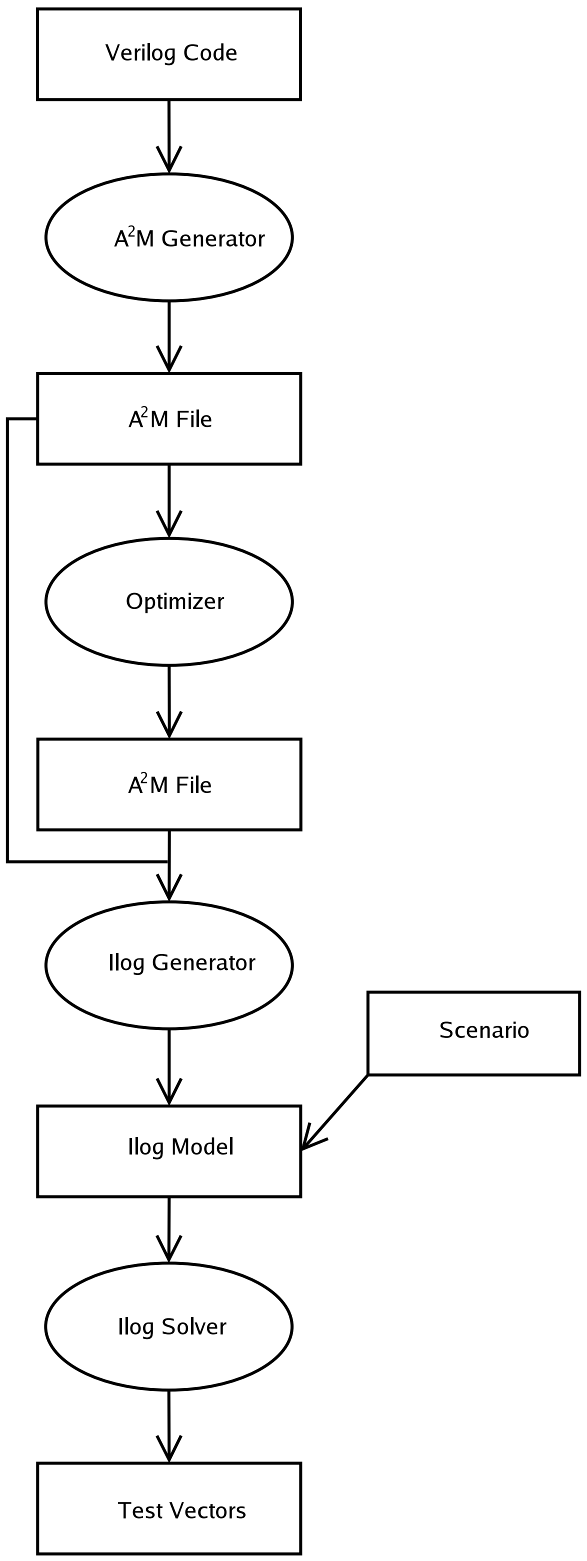

Figure 3.1: Design Flow

`(continuous) statements` and the `Module instantiations` [1]. Hence, the acronym A^2M.

The code of the Verilog parser is written in the `yacc` which captures the grammar of Verilog and uses the syntax directed translation to construct the A^2M graph.

3.1.1 A^2M Graph

The A^2M graph has following four entities that captures the entire information about the underlying digital circuit described in the Verilog code:

1. **Module** :

   Every *module definition* in the Verilog code maps on to a `Module` entity of the A^2M graph. The `Module` entity contains various fields necessary to encapsulate a module definition in Verilog. They include `Name` of the module, an array of all the `Components` (module instantiations, always and assign statements) inside a module, an array of all the `Signals` (all `reg` and `wire` Verilog variables including the `input` and `output` variables) inside a module, an array of all the signal indices inside the `Signal` array that form the `input` to the module, an array of all the signal indices inside the `Signal` array that form the `output` to the module.

2. **Component** :

   Every *always*, *assign* and *module instantiation* statement in the Verilog code maps on to a `Component` entity of the A^2M graph. The `Component` entity contains the various fields required to encapsulate the three types of structures, a module Instantiation, an assign statement and an always block inside a module definition. Two additional `Component` types are defined, namely, the `feedback component` and the `stem component` to handle the loops and stems (signals that drive multiple components) respectively in a Verilog design. Various fields inside a `Component` are `name`, `type`, pointer to the `Parent Module` (the Verilog module inside which the Component is defined), `fan-ins` (input signals to the component), `fan-outs` (output signals of the component), and the `ADDNode` corresponding to the Component.

3. **Signal** :

   The `Signal` entity contains various fields required to encapsulate a signal inside a Verilog module. The signals connect the different `Component` entities in the A^2M graph. The different fields inside a Signal are `name` and `type` of the Signal, the `source component` of the Signal, the `destination component` of the Signal, the `size` of the signal (if it is a vector), and the `constant` value (if the signal is assigned to a constant in the Verilog module).

4. **ADD Node** :

   The `ADDNode` captures the functionality of the `Component` and is defined only for the `Components` that corresponds to an `always` block or an `assign` statement. The ADD Framework is an internal representation of the HDL Description and has been shown to be *complete and efficient*. More details on ADD are available in [3, 6, 38, 40].

3.1.2 A^2M Graph Optimization

The A^2M graph generated above is a very generic representation that captures the complete information present in the Verilog design. We introduced an additional stage to optimized it for integer modeling. The optimization stage refines the A^2M graph by two levels so as to remove the `Component` entities that shall cause redundant constraints in the Integer model to be generated in the next phase.

1. First level of refinement is the elimination of the `stem components` from the graph which is a redundant information for the constraint model, as all the branches driven by the *stem component* can be replaced by a single variable. Thus, the `stem components`, if kept in the A^2M input graph shall result in more number of constraints and in-turn effect the efficiency of the solver. However, if the test scenario assumes that a branch of a particular `stem component` s is faulty, then s may be retained.

2. The second level of the refinement is the removal of unnecessary constants and related operators from the A^2M graph. For example, the `ADDNode`

corresponding to $b \leftarrow c\&1$ can be replaced by the one corresponding to $b \leftarrow c$. Here b is the `target` signal and c is the `original` signal. Such redundant logical operations are eliminated by replacing the `target` by the `original` signal.

These refinements reduce the number of constraints and the variables in the subsequent constraint model, hence improving the performance of the solver.

3.2 Constraint Generation

This section describes how to generate the *model constraints* from the given A^2M graph. We show different HDL constructs, their equivalent ADD representation and the corresponding constraint models. This in turn, shall explain the automatic generation of model constraints from the given behavioral Verilog model.

3.2.1 Reg and Wire Variables

Two types of variables are commonly used in Verilog models, namely, the reg and wire variables [1]. These variables can either be bits or integers (bit-vectors). The integer domain deals more with bit-vectors rather than individual bits. This in turn, increases the modeling complexity.

Any bit-vector of size n in the input behavioral model is mapped on to a `Signal` entity in the A^2M graph with an attribute, $size = n$. In integer domain, this is treated as integer variable (*IloIntVar* in ILOG) whose value ranges from 0 to $2^n - 1$. The single bit variables are also treated as bit-vectors with size $n = 1$. The variables Verilog map on one of the two types of nodes on the A^2M graph, the *read nodes* or *write nodes*. As the names suggest, the variables on the right hand side of a Verilog expression map on to read nodes and the variables on the left hand side map to write nodes. The read and write nodes in the A^2M graph have two attributes attached to it, namely *range* and *index*. The need for these attributes arises from the fact that a vector can be referred in the following three ways in a Verilog code:

1. **Bit Select**: For example, $ir[3]$, that selects the third bit of the bit-vector ir. The respective node in the A^2M graph has $index = 3$ and $range =$

$[-1 : -1]$. Since, the integer solver do not directly deal with bits, the corresponding ILOG representation of $ir[3]$ is

$$IloDiv(ir, 2^3) - 2 * IloDiv(ir, 2^4),$$

where, the function `IloDiv` stands for integer division in ILOG.

2. **Part select**: For example, $ir[16 : 13]$, that selects the four bits starting from the bit 13 through 16 of the bit-vector ir. The respective node in A^2M graph has $range = [16 : 13]$ and $index = -1$. The corresponding ILOG representation is

$$IloDiv(ir, 2^{13}) - 2^4 * IloDiv(ir, 2^{17})$$

3. **The entire vector**: For example ir. The respective node in A^2M graph has $index = -1$ and $range = [-1 : -1]$. The ILOG representation models the variable ir as a normal integer ($IloIntVar$).

The variables defined inside a module M carry different values for different instantiations of M. To encapsulate this, every variable in M is declared as an array of size equal to the number of instantiations of M. For example, given that the variable ir is defined in M and that M is instantiated twice, then $ir[0]$ ($ir[1]$) denotes the value of ir in instantiation 0 (1). On similar lines, the third bit of ir in instantiation 0 is modeled as

$$IloDiv(ir[0], 2^3) - 2 * IloDiv(ir[0], 2^4).$$

The part select of ir comprising of the four bits (bits 13 to 16) in instantiation 1 is modeled as

$$IloDiv(ir[1], 2^{13}) - 2^4 * IloDiv(ir[1], 2^{17})$$

Bit-select and Part-select in Verilog lead to complex constraints in the Integer Domain. In contrast when the entire vector is selected then no additional constraint is added in the Integer model. Important point to be noted is that if the Verilog has less number of Bit or Part select then the Integer model that will be generated will be relatively less complex.

3.2.2 Operators

Integer solvers do not support certain operators that are provided by the HDLs like bit-wise operators, concatenation operation [1]. Hence, the technique handles these operators by modeling them as functions in the Integer Domain as described below:

3.2.2.1 Bitwise operators

The integer domain does not provide any support for bitwise operators. Hence we need to model each of the bitwise operators explicitly. Let us take an example of bitwise AND operator :

$$assign\ out = in\_1\ \&\ in\_2$$

where out, in_1 and in_2 are all one bit Verilog wires. The corresponding A^2M graph is shown in Figure 3.2.

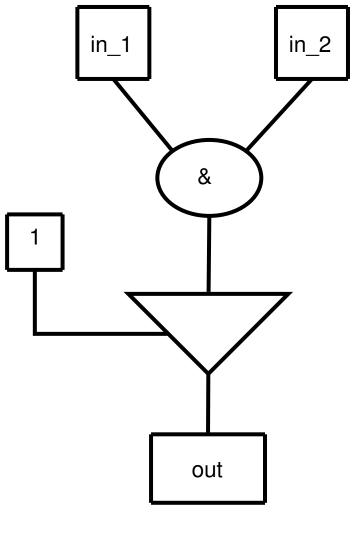

Figure 3.2: Bit-wise operator: AND

The nodes labeled in_1 and in_2 are the read nodes and the node labeled out is a write node. The node labeled & denoted the bit-wise AND operator node. The triangle is the *decision node* [38] that assigns the output of the operator node to the write node *if the input condition is true*. In this case the condition node is always true (node labeled 1). The A^2M structure is converted to the following constraint by a function that is called when a bitwise AND operator node is encountered.

$$out == ((in\_1 == 1)\&\&(in\_2 == 1))$$

The above method is extended to handle the case wherein in_1, in_2 and out are vectors. Similarly all other bit-wise operators are modeled as constraints in the integer domain.

3.2.2.2 Logical operators

Logical operators are of two types:

1. *Logical Boolean operators*: This includes logical AND, logical OR etc. These are handled similar to the bit-wise operators.

2. *Logical Comparison operators*: Consider the following code snap

   ```
   $if(branch\_cond == 1)$
           $pc = npc$
   $else$
           $pc = alu\_out$
   ```

   The corresponding A^2M graph is shown in the Figure 3.3.

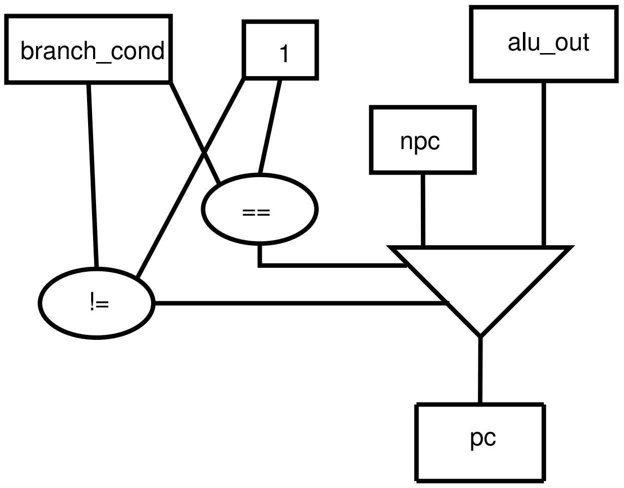

Figure 3.3: Logical Comparison operator

The above graph is modeled as follows:

$pc == ((branch\_cond == 1) * npc) + ((branch\_cond! = 1) * (alu\_out))$

The above constraint is read as pc is equal to npc if $branch\_cond == 1$ and equal to alu_out if $branch\_cond == 0$. As only one condition can be true, pc will be assigned to either npc or alu_out.

3.2.2.3 Concatenation operator

Concatenation operator is modeled using simple arithmetic. For example, consider the Verilog statement:

$$imm\_reg = \{4\{sign\}, imm\_value\}$$

where imm_reg is a 16-bit register, imm_value is 12-bit wire and sign is a 1-bit wire.
The corresponding A^2M graph is shown in Figure 3.4.

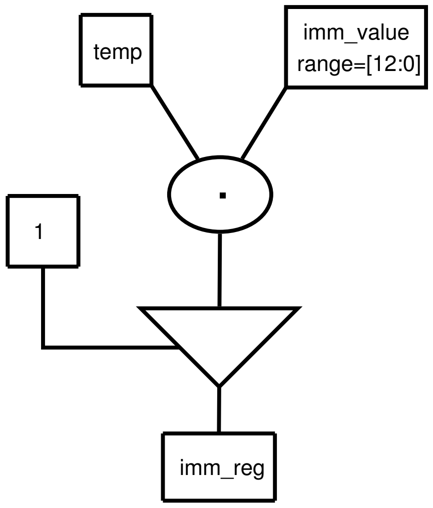

Figure 3.4: Concatenation operator

The corresponding model constraints are generated in the following way:

$$temp == 2^3 * sign + 2^2 * sign + 2^1 * sign + 2^0 * sign$$
$$imm\_reg == imm\_value + temp * 2^{13}$$

3.2.3 Modeling Sequential Circuits

Every sequential circuit represented by the conventional Huffman model [5]. The combinational and the sequential parts are clearly distinguished in this representation. To model the circuit in the integer domain the following two basic principles are used:

1. Each sequential element is a variable in the integer domain; and
2. Each combinational element produces a constraint on its inputs and outputs in the integer domain.

The behavior of a sequential circuit S over k time frames can be modeled as a combinational circuit using the conventional time frame expansion approach, which unrolls the combinational part of S, k times [5]. The above mentioned approach is illustrated by using an example of counter. The following Verilog code models a counter

$reg[4:0]counter;$
$always@(posedge\ clk)$
$\qquad counter = counter + 1;$

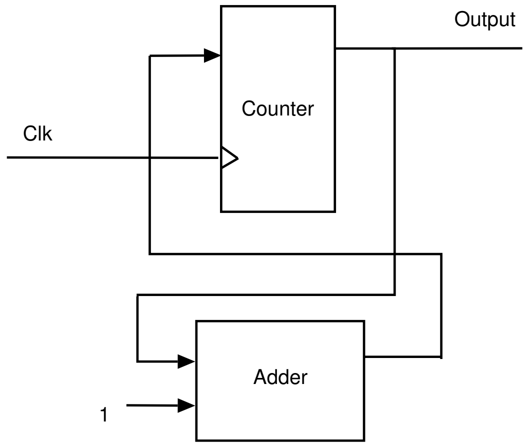

Figure 3.5: Counter

The Huffman Model of the counter is shown in Figure 3.5. The register labeled *Counter* forms the sequential part and the *Adder* forms the combinational part. At every clock pulse the value in the *Counter* gets incremented by 1.

The unrolled model of the counter is shown in Figure 3.6. The circuit is unrolled over 2 time-frames. The *Counter* blocks shown in Figure 3.6 are just wires. Note that by assigning, say, 5 to the *Initial Counter* in Figure 3.6, 6 and 7 are outputs in the *Counter* blocks representing the 1 and 2 time-frames respectively. This indeed captures the functionality of the counter over two time-frames.

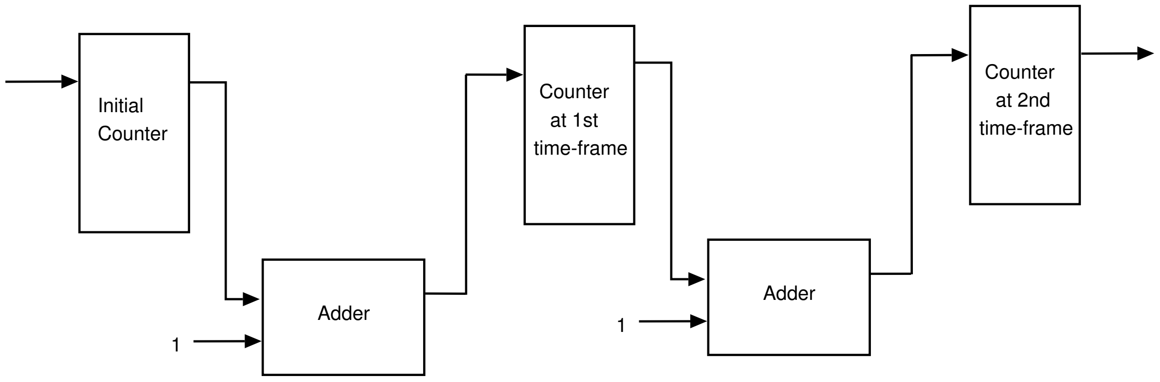

Figure 3.6: Time-frame unrolled Counter

The register Counter in Figure 3.5 is a *reg* in the Verilog description. Without loss of generality, let there be only one instantiation of the module. To unroll the circuit over time frames, one dimension is added to the variable to represent the time frame. Thus, the variable $counter[j]$ denotes the variable $counter$ in the j^{th} time-frame. The combinational part of the Huffman Model is the one that sets constraints on the variable $counter[]$ across time-frames. The Verilog code and the underlying A^2M graph imply the following constraint:

$$counter[j] = counter[j-1] + 1$$

Thus, the clock in Verilog is realized as a time frame in the corresponding constraint model.

3.2.4 Module Instantiation

Two major points have to be addressed to handle module instantiations, namely, handling the variables and addition of interface constraints.

3.2.4.1 Handling variables

There are two issues in variable naming and variable indexing as described below:

- The Verilog design description is hierarchical and hence different modules can have variables with the same name. While deriving the constraint model, the variables with same names in different modules have to be distinguished. To resolve this, a variable *var* inside a module *module_name* is referred to as *module_name_var* in the constraint model. The fact that two module definitions do not have the same name solves the above issue.

- Verilog supports multiple instantiations of a module and hence the wires inside the module are realized as different in different instantiations. To incorporate this in the integer domain, we introduce another dimension to the variables. For example, the variable *out* in the second instantiation of the module *alu* in the time-frame tf is specified in the constraint model as $alu\_out[1][tf]$. While generating the scenario constraints the user needs to specify variables corresponding to a particular instantiation of a module. This demands a mechanism that simplifies the specification process. Specifically, the user should easily infer the index into the variable array corresponding to a particular instantiation. This is achieved as follows:

  Whenever a module M is instantiated with a name, say, M_inst, the size of the arrays corresponding to the *wire*, *reg*, *input* and *output* variables inside M increases by one. This incremented size S also denotes the number of times M is instantiated till that point. The hierarchical name of M_inst is used as a $\#define$ constant with value $S - 1$ in the constraint model. Given that the hierarchical name of every instantiated module is different [1] ensures that there are no conflicts. In the constraint model the user can use the hierarchical name of the instantiated module as an index to specify any variable inside it.

  For example, in the DLX processor design there are module definitions by the name *dlx* and *alu*. The module definition of *alu* is as follows:

$$modulealu(alu\_out, clk, alu\_in1, alu\_in2, opcode)$$

The module instantiates *alu* with the name of *alu*1 as:

$$alu\ alu1(alu\_out, clk, alu\_in1, alu\_in2, opcode2)$$

The *dlx* module in turn, is instantiated by the *top* module of the Verilog hierarchy with the name *dlx*1. Note that the hierarchical name of *alu*1 is *top_DLX1_alu1*, which is a $\#define$ constant in the constraint model. The variable *out* in the instantiation of *alu*1 in the time frame tf is specified as:

$$alu\_out[top\_dlx1\_alu1][tf]$$

3.2.4.2 Interface Constraints

These constraints are used to establish connections between the initiating and the instantiated modules. The number of interface constraints is equal to the number of input and output ports of the instantiated module.For example, given that the circuit is unrolled over MAX_TF time frames, the constraints generated to effect the interface between the *alu* instantiation and the *dlx* are as follows:

```
$for(i = 0;\ i < MAX\_TF;\ i++)$
{
    $alu\_out[top\_dlx1\_alu1][i] == dlx\_alu\_out[top\_dlx1][i]$
    $alu\_in1[top\_dlx1\_alu1][i] == dlx\_alu\_in1[top\_dlx1][i]$
    $alu\_in2[top\_dlx1\_alu1][i] == dlx\_alu\_in2[top\_dlx1][i]$
    $alu\_opcode[top\_dlx1\_alu1][i] == dlx\_opcode2[top\_dlx1][i]$
}
```

As mentioned earlier, the clock in Verilog corresponds to time frames in the constraint model. Hence no constraint is generated for the *clk* port in the above instantiation.

3.2.5 Always Statement

The constraint model for an always construct in Verilog depends on the *event* in its *sensitivity list*. The event can be either *clocked* or *non-clocked* and is represented by an event node in the corresponding A^2M graph. First, consider the following code where the *event* in the sensitivity list is *clocked*:

$always@(posedge\ clk)$
    $opcode2 <= opcode;$

The A^2M graph for the above code is shown in Figure 3.7.

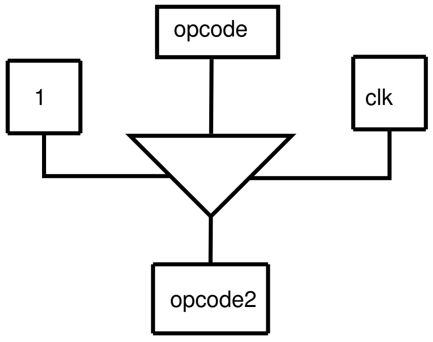

Figure 3.7: Always Statement with *clk*

The model constraint for the above A^2M graph is added for all MAX_TF time-frames. The constraints for the k^{th} instantiation of the module to which it belongs to and in tf^{tf} time frame will be:

$$opcode2[k][tf+1] == opcode[k][tf]$$

Consider the following code which is a non-clocked always structure.

```
$always@(sel\_alu\_in1\ or\ npc2\ or\ a)$
  $if(sel\_alu\_in1 == 1)$
    $alu\_in1 <= npc2;$
  $else$
    $alu\_in1 <= a;$
```

The A^2M graph for the non-clocked always statement is shown in Figure 3.8.

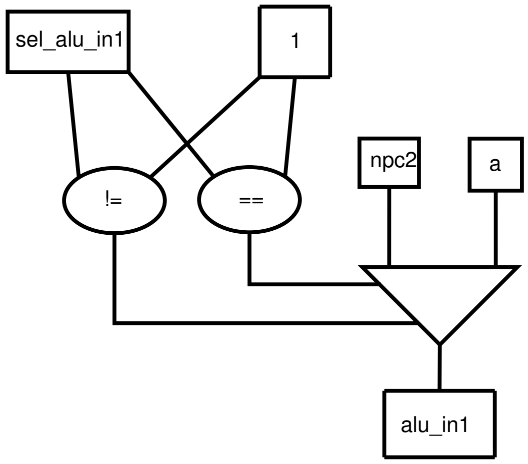

Figure 3.8: Always statement without *clk*

The model constraint for the above A^2M graph will be added in all MAX_TF time-frames. The constraint for the k^{th} instantiation of the module to which it belongs to and the tf^{tf} time frame will be:

$alu\_in1[k][tf] == (sel\_alu\_in1[k][tf] == 1)*npc2[k][tf]+(sel\_alu\_in1[k][tf]! = 0) * a[k][tf]$

The assumptions made in in writing the above constraints from its corresponding Verilog code is that the sensitivity list of a clocked always block can have either *posedge* or *negedge*, not both. The sensitivity list in the non-clocked always block is ignored. Currently, the multiple clocks are also not supported, but it the design can be extended to support multiple-clock domain.

3.2.6 Assign Statement

The assign statements in the Verilog leads to constraints in all time-frames. These statements do not have any notion of clock and hence all the constrains are confined to one time-frame. For example:

$$assign\ sum = input\_1 + input\_2$$

The model constraint for the above A^2M graph are added in all MAX_TF time-frames. The constraints for the k^{th} instantiation of the module to which it belongs to will be:

$$sum[k][tf] == input\_1[k][tf] + input\_2[k][tf]$$

3.2.7 Value Remembrance

The major challenge in modeling Verilog constructs is the modeling of memory units. The Verilog statements do not explicitly state that the value has to be remembered to the next time frame. But, the modeling of constraints should be frames in such a way that the values are carried forward appropriately. The basic assumption in Verilog that if a variable is set under clock edge then it is a Flip-Flop. If a value is not written in the Flip-Flop in a particular clock cycle due to some condition C, then we should copy the contents if the condition C is false. For example, let us consider the code snap form the data memory:

```
always@(posedge clk)
   if(write_enable == 1)
      data0 <= data_in
```

where $write\_enable$ is the enable signal, $data0$ is the first address line and $data\_in$ is the data input to the data memory. The constraint for the above Verilog code will be:

$$data0[k][tf+1] == (write\_enable[k][tf] == 1) * data\_in[k][tf] + \\ write\_enable[k][tf] == 0) * data0[k][tf])$$

The above constraint takes care of the fact that the value of $data0$ is taken forward to the next time-frame if $write\_enable$ is 0. Hence, whenever there are some unhandled conditions then the constraints for those cases should also be explicitly added.

3.2.8 Explicit constraints

The ILOG $IloIfThen$ is as follows:

$$IloIfThen(env, condition, statement);$$

which means that the if the *condition* is constrained then the *statement* constraint will also be added to the model.

If the model is constrained using the Ilog construct, $IloIfThen$, then the following kinds of problems might occur:

1. **Backward constraints**

   When the *statement* is constrained then the *condition* will not be implicitly constrained. This case should be handled explicitly. To solve the above problem the following constraint should be added the the model:

   $$IloIfThen(env, statement, condition);$$

2. **Else Condition**: One must explicitly add constraints for the case when the *condition* in the $IloIfThen$ is not true. Consider the following Verilog code snap:

   ```
   always@(posedge clk)
       if(write_enable == 1)
           data0 <= data_in
   ```

   One can model the constraint for the above as follows:
   $IloIfThen(env, write\_enable[k][tf] == 1, data0[k][tf{+}1] == data\_in[k][tf])$

   The above constraint considers the case when *write_enable* is 1. But, when *write_enable* is 0 then the constraint is not added to the model. Hence, we should explicitly add another constraint that will consider the case when the condition is false in the $IloIfThen$ construct. For the above case, the additional constraint will be:

$$IloIfThen(env, write\_enable[k][tf]! = 1, data0[k][tf+1] == data0[k][tf])$$

3.3 Scenario Constraints

This section explains the way the scenarios constraints are added to the model. We explain this by few examples form the dlx-architecture.

- **Scenario 1**: Generate the sequences of instructions that will not have RAW hazards in the next 5 clock cycles starting at time-frame tf.

  The *scenario constraints* will be as follows:

```
$for(IloInt\ i = 1; i <= 4; i++)$
        $for(IloInt\ j = 1; j <= 5; j++)\{$
            $(dlx\_rd[top\_dlx][tf+i]! = dlx\_r1\_id[top\_dlx][tf+j])$
            $(dlx\_rd[top\_dlx][tf+i]! = dlx\_r2\_id[top\_dlx][tf+j])$
        $\}$
```

  The above constraints ensure that rd must not be equal to either $r1\_id$ or $r2\_id$ for consecutive 5 time-frames starting from tf, where rd is the destination register, $r1\_id$ is the value of the register 1 and $r2\_id$ is the value of the register 2.

- **Scenario 2**: Generate the sequences of instructions that will not have WAW hazards for 3 consecutive clock cycles starting from tf time-frame.

  The *scenario constraints* for the above case will be as follows:

```
$for(IloInt\ i = 1; i <= 2; i++)$
        $for(IloInt\ j = 1; j <= 3; j++)$
```

$$(dlx\_rd[top\_dlx][tf+i]! = dlx\_rd[top\_dlx][tf+j])$$

The above constraints ensures that rd will not be same for 3 consecutive time-frames starting from tf.

- **Scenario 3**: Generate the sequences of instructions that will generate ALU output to be $value1$ in time-frame $tf1$ and $value2$ in time-frame tf

  The *scenario constraints* for the above case will be as follows:

$$dlx\_alu\_out[top\_dlx][tf1] == value1$$
$$dlx\_alu\_out[top\_dlx][tf2] == value2$$

- **Scenario 4**: Generate the sequences of instructions that will access consecutive memory address starting from *addr* in 5 consecutive clock cycles starting from tf time-frame.

  The *scenario constraints* for the above case will be as follows:

```
$for(IloInti = 0; i < 5; i++)$
    $(data\_memory\_addr\_in[top\_dlx][tf+i] == addr + i)$
```

  The above constraints state that the input address of the data memory is set to $address + i$ in time-frames $tf + i$

- **Scenario 5**: Generate the sequences of instructions that will access same memory address, *addr*, in 5 consecutive clock cycles starting from tf time-frame

  The *scenario constraints* for will be as follows:

```
$for(IloInt\ i = 0; i < 5; i++)$
    $(data\_memory\_addr\_in[top\_dlx][tf+i] == addr)$
```

- **Hybrid Scenarios**: Hybrid scenarios can be easily created by adding the desired constraints together, for example we can have a hybrid scenario where *Scenario 4* and *Scenario 1* should occur together.

  The model constraints for this hybrid scenario will look like:

  $for(IloInt\ i = 1; i <= 4; i++)$
  $\quad for(IloInt\ j = 1; j <= 5; j++)\{$
  $\quad\quad (dlx\_rd[top\_dlx][tf + i]! = dlx\_r1\_id[top\_dlx][tf + j])$
  $\quad\quad (dlx\_rd[top\_dlx][tf + i]! = dlx\_r2\_id[top\_dlx][tf + j])$
  $\quad \}$

  $for(IloInt\ i = 0; i < 5; i++)$
  $\quad (data\_memory\_addr\_in[top\_dlx][tf + i] == addr + i)$

3.4 Algorithm

As mentioned earlier, the conversion from Verilog to the equivalent A^2M graph is done automatically using a syntax-directed translation. This section presents the algorithm *i-Gen* 3.4.1 that automates the conversion from the A^2M graph to a constraint model.

The working of the algorithm *i-Gen* is described as follows: The block in the A^2M graph is either a *module declaration*, *Signal* or *Component*. The routine *read block* used by *i-Gen*, reads a *block* from the graph. The first *block* read (in line step 3) from the A^2M graph is always going to be a *module declaration* corresponding to the top module of the Verilog design. Hence the condition in the step 4 will be true during the first time. The list of module declarations is updated in step 5. As the topmost module of the Verilog is not instantiated explicitly in the design, step 6 and 7 adds the topmost module in the list of module instantiations. Steps 10 to 13 update the list of variables in the particular module by parsing all the *Signals* in it. Steps 14 to 23 handles a *Component* block. A *Component* in the A^2M graph corresponds either to a module instantiation, a stem , an assign statement or an always statement. The steps 14 to 16 handles

Algorithm 3.4.1 The i-Gen Algorithm

```
1: igen ($A_2M$)
2: begin
3: read *block*
4: **while** $block = ModuleDeclaration$ **do**
5:    update list of *module declarations*
6:    **if** *current module* = *top module* **then**
7:       update list of *modules instantiations*
8:    **end if**
9:    read block
10:   **while** $block = Signals$ **do**
11:      update list of *module variables*
12:      read *block*
13:   **end while**
14:   **while** $block = Component$ **do**
15:      **if** $Component = MODULE$ **then**
16:         update list of *module instantiations*
17:      **else if** $Component = Stem$ **then**
18:         constrain $fan-outs$ to be equal to $fan-ins$
19:      **else**
20:         generate constraints for the *always* or *assign block*
21:      **end if**
22:      read *block*
23:   **end while**
24: **end while**
25: **for all** *module instantiations* **do**
26:    generate connection constraints
27: **end for**
28: end
```

the case when the component is the *module instantiation* by updating the list for module instantiations. The steps 17 and 18 handles the case when the component is a *Stem* by adding constraints that sets fan-ins to be equal to the fan-outs. Step 20 is executed when the component is either an *always* or an *assign* statement. In this case, the constraints corresponding to the ADD graph of the component is generated. Steps 25 to 27 generated the *interface* constraints for all the module instantiations. The above algorithm generates the Integer model by parsing the A^2M graph and maintains the hierarchy present in the Verilog design.

Chapter 4

Experimental Results

This chapter presents the results obtained by several experiments. Observations based on the obtained results is also made which provide a good insight of the future research in this area. All the results reported are obtained by experimentation on an HP workstation xw4200. The time and memory utilized are as reported by the ILOG constraint Solver. Extensive experimentation was done on the 16-bit DLX processor model. The model was the behavioral description of DLX processor written in Verilog. Rest of the chapter is divided in to two categories as follows:

4.1 Scalability of the Constraint Model

It is a well known fact that the generation of a constraint model for multiple time frames consumes time and memory that grows linearly in the number of time frames. The results obtained for a Full adder circuit is shown in Table 4.1 (*Experiment* 0). The table shows the time and memory required to generate and solve the constraint model of the full adder unrolled for different number of time frames and solving for some value of output. The important point to note is that in each of the cases shown in Table 4.1 a single constraint model is generated that represents the complete unrolled circuit.

The Figure 4.1 shows that the solver performance is linear with the number of time-frames unrolled.

The Tables 4.3(a), 4.3(b) and 4.3(c) compare the behavioral verilog model of the DLX processor with its corresponding A^2M graph and the constraint model.

No of Time frames Unrolled	Time (in sec)	Memory (in MB)
10	0.02	1.01
20	0.04	1.95
50	0.1	4.72
100	0.21	9.33
500	1.65	46.6
1000	4.48	92.13
10000	260.94	922.98

Table 4.1: Results of Full Adder

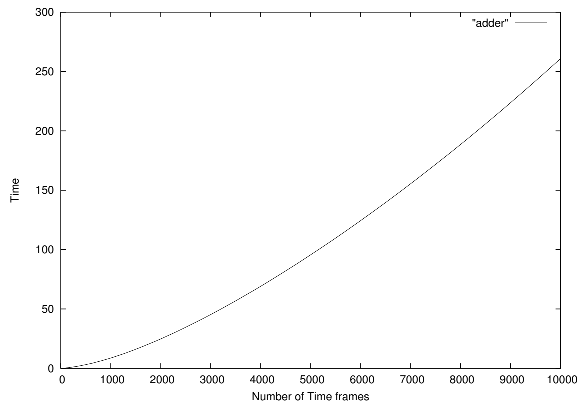

Figure 4.1: Adder Graph

It is evident from the Tables 4.3(a) and 4.3(c) that the number of variables (746) and number of constraints (530) in the optimized constraint model is far less than the number of gates (13104) needed to realize the behavioral Verilog model. This justifies the claim that a tool dealing with higher level models handles data structures of lesser size than a tool dealing at lower levels.

The benefits of optimizing the A^2M graph can be inferred from Tables 4.3(b) and 4.3(c). As seen in Table 4.3(b) there is a 46.7% and 35.3% decrease in the number of *signals* and *operators* respectively of the A^2M graph due to the optimization. As an effect of this optimization there is a corresponding 46.9% and 61.6% decrease in the number of variables and constraints in the underlying ILOG based constraint model, Table 4.3(c).

Table 4.3 shows the time required by various phases in generating the ILOG

Table 4.2: Analysis of scalability between Verilog, A^2M, Ilog

No of lines	No of Gates
804	13104

(a) Verilog Model

	No of Signals	No of Operators	No of AND Nodes
Before Optimization	1411	561	214
After Optimization	751	363	214

(b) A^2M Graph

	No of Variables	No of Constraints
Before Optimization	1406	1381
After Optimization	746	530

(c) Ilog Model

based constraint model for the 16-bit DLX processor. The total time consumed for the entire three phases was less than $0.4s$. This illustrates the efficiency of the proposed methodology in handling large and complex designs.

4.2 FTG for user-defined Scenarios

This section deals with time and memory requirements for FTG under various scenario constraints. The specification of the scenario constrints is also explained. The following experiments are done on the 16-bit DLX processor.

- **Experiment 1**: *Instructions generation without Hazards*

  This experiment is based on *Scenario 1* and *Scenario 2* discussed in *Chapter 3*. The table 4.4 shows the time taken for unrolling the DLX model to 6 or 10 time-frames and generating test vectors that ensures no RAW (Read After Write) hazard over all the time-frames and no WAW (Write After Write) hazard within 3 or 5 consecutive time-frames as stated in the *Scenario* column of the same.

Phase	Timing (in sec)
A^2M graph generation	0.022
A^2M graph optimization	0.145
Constraint model generation	0.198
Total	0.365

Table 4.3: Timing Analysis

Scenario	No of Time Frames	Time (in sec)	Memory (in MB)
No RAW No WAW for 5 time-frames	6	32.96	182.42
No RAW No WAW for 5 time-frames	10	84.98	379.92
No RAW No WAW or 3 time-frames	6	31.86	182.16
No RAW No WAW for 3 time-frames	10	81.97	379.31

Table 4.4: Instruction generation without Hazards

Observation: From table 4.4, it is seen that the constraint solver consumes more time and memory with increasing number of time frames to which the model is unrolled. Hence, the time and memory requirements are proportional to the size of the model.

- **Experiment 2**: *Instruction generation for specified ALU outputs*

This experiment is based on *Scenario 3* discussed in *Chapter 3*. The *Scenario* column shows the constraints on the ALU output (*alu_out*) in particular time-frames. The time taken for unrolling the DLX model to 6 time-frames and generating test vectors for the specified *scenario* is shown in table 4.5. The generated test vector may have hazards, i.e. the constraints of *Experiment 1* are not included.

The scenarios stated in this experiment and other experiments that are based on ALU operations have additional constraints that initialize the 16 registers of the DLX processor such that $R_i = i$, $0 \leq i \leq 15$.

Scenario	No of Time Frames	Time (in sec)	Memory (in MB)
ALU Out = 5 in time frame 2 ALU Out = 13 in time frame 3	6	11.08	88.52
ALU Out = 5 in time frame 2 ALU Out = 13 in time frame 3 ALU Out = 10 in time frame 4	6	10.36	88.77
ALU Out = 5 in time frame 2 ALU Out = 13 in time frame 3 ALU Out = 10 in time frame 4 ALU Out = 20 in time frame 5	6	10.87	89.1

Table 4.5: Instruction generation for ALU operations

Observations: From table 4.5 it is seen that the addition of more number of independent constraints (across time-frames) does not significantly impact the performance of the constraint solver.

- **Experiment 3**: *Instruction generation for Memory Access*

  This experiment is based on *Scenario 4* and *Scenario 5* as discussed in *Chapter 3*. Table 4.6 shows the time taken for unrolling the DLX model to 10 time-frames and generating test vectors to access two specified memory addresses under three different scenarios. In the first scenario the time-frame numbers for the memory access and the memory addresses were different and independent of each other, while, in the remaining two scenarios, the two memory addresses were dependent, leading to constraints that were dependent on each other.

  Observation: From table 4.6 it is seen that the constraint solver consumes less time and memory to solve independent constraints in comparison to dependent constraints. The dependency is measured across time-frames. Hence, the approach will produce the results faster if the scenario is independent across time-frames.

- **Experiment 4**: *Instruction generation for ALU operations without hazards*

Scenario	No of Time Frames	Time (in sec)	Memory (in MB)
Access two different memory addresses in two different time-frames	10	14.8	134.53
Access same memory addresses in time-frames 7 & 8	10	66.74	380.66
Access consecutive Memory addresses in time-frames 7 & 8	10	68.3	380.69

Table 4.6: Instruction generation for Memory Access

In this exercise we experiment the *hybrid scenarios* by combining scenarios from *Experiment 1* and *Experiment 2*. The table 4.7 shows the time taken for unrolling the DLX model to 6 time-frames and generating test vectors that ensures the scenarios specified in table 4.5 without RAW hazards over all the time-frames and no WAW hazard within 3 consecutive time frames.

Scenario	No of Time Frames	Time (in sec)	Memory (in MB)
Scenarios 3 & 1 from Table 4.5 & 4.4 respectively	6	10.47	45.62
Scenarios 3 & 2 from Table 4.5 & 4.4 respectively	6	959.98	242.34
Scenarios 3 & 3 from Table 4.5 & 4.4 respectively	6	959.95	242.63

Table 4.7: Instruction generation for ALU operations without hazards

The assembly code generated for the *Scenario 3* of table 4.7 is as follows:

```
ADD R1 R2 R3
ADD R3 R13 R0
```

MUL R2 R5 R2
ADD R1 R11 R9

As mentioned earlier, given that registers R_i initially store the value i, $0 \leq i \leq 15$, implies that the above assembly code indeed generates the specified scenario.

Observation: Results in tables 4.5 and 4.7 reveals interesting issues. The solver took less time to solve the first scenario in table 4.7 when compared to the first one in table 4.5. This is in spite of the fact that the former is a combination of the latter and additional constraints. The reason for the same would be that the additional constraints in the former scenario have reduced the search space, essentially directing the solver towards the goal.

- **Experiment 5**: *Simulation*

  It is interesting to note that the constraint model of a given HDL behavioral code can be used for logic simulation. Table 4.8 presents details on simulation of the Ilog model of the DLX processor. The model was unrolled for the number of time frames as stated in table 4.8. The instructions in the Scenario column of table 4.8 were applied to the initial time frame as constraints. Solving the constraints is equivalent to simulating the DLX processor over the next k cycles, where, k is the number of times the model is unrolled. In other words, for the first scenario in table 4.8, the DLX constraint model was unrolled for 7 time frames and the variables corresponding to the instruction memory of the first frame was constrained to the binary equivalent of the instruction $LD\ R1$, $[R2]$. The initialization constraints included constraints on the variables corresponding to the registers of the processors. These variables were constrained to some known values.

  Observation: In the scenarios for simulation the flow of the data is in the forward direction and hence easy to solve. As the inputs are constrained it becomes very easy for the solver to follow the constraints and assigning values to intermediate variables leading to the values to the outputs. This is

Scenario	No of Time frames unrolled	Time (in sec)	Memory (in MB)
LD R1, [R2]	7	0.09	5.51
LD R1, [R2] ST [R6], R2 LD R12, [R1]	7	0.13	7.61
LD R1, [R2] ST [R6], R2 LD R12, [R1] ADD R3, R1, R2	12	0.23	12.90

Table 4.8: Forward Simulation of DLX

reflected by the lesser amount of time and memory consumed by the ILOG solver as reported in table 4.8.

Chapter 5

Conclusions and Future Work

This chapter is divided in to two parts each describing the conclusions and the future research directions.

5.1 Conclusions

In this project we proposed a methodology for automatic conversion of a any given behavioral HDL model into a set of integer constraints. The constraint model was used for generation of directed functional tests. The conversion was fully automated and the A^2M graph was optimized for the integer model generation. The optimization process was similar to that of logic synthesis. Integer (word-level) constraint solvers were employed by the proposed methodology in contrast to the widely reported SAT-solvers. The effectiveness of the proposed approach was illustrated by employing the same on a 16-bit DLX processor behavioral model and using the same to generate functional tests for different interesting scenarios. The methodology enabled the user to express the corner case to be tested as an higher-level constraint-based test specification spreading across multiple time frames.

There are several advantages of the methodology:

1. The entire process of integer model generation is automated and very little human intervention is required for the generation of functional test.

2. The main focus was to generate models for behavioral HDL designs. The behavioral designs are much more abstract as compared to the gate-level description and hence are more scalable.

3. Sequential circuit designs are handled with ease. Dynamic unrolling was used which in turn makes it beneficial as the scenarios can span over multiple time-frames

4. The Scenario specification is very easy as compared to the gate-level techniques. At word-level the describing a complex scenario boils down to just specifying constraints on the variables that directly corresponds to the buses in the Verilog description. Whereas, at bit-level, it is cumbersome to specify a value to a bus.

5. The automatic integer model generation can be applied to a lot of applications. One of them is the FTG which has been proved to be promising for processor environment. Another application is the generation of *Power Virus* [41]. In the parallel project on *Power Virus generation*, we have used the proposed technique and found a significant improvement over the random vector generation technique. Another application where this approach can be effective is to find the set of instructions that will cause a circuit to malfunction due to excessive power dissipation. One can identify the regions that are active (places where maximum toggling is taking place) when the circuit misbehaves and try to change the design accordingly. The power of defining scenarios and objectives makes it applicable to a large number of applications.

5.2 Future Work

This project has opened up a whole new space for research in this direction and has thrown light on many dark areas that have a lot of potential to shape the future testing techniques. The following features can further enhance the versatility of the proposed methodology:

1. *Multi-clock designs and Latches*: The concept of time-frames discussed so far assumed that all the storage elements are flip-flops, and, they are driven by a single clock. This can be extended for Multi-clock and Latch based designs by employing a verilog type technique, where in, a basic simulator time-unit is assumed and every event (clocks and changes in wires) in the design is sampled at this time-unit [1]. A similar approach may be employed here, wherein, the time frame is determined by the basic time-unit.

2. *Handling large vectors*: As the size of vectors increases, the search space increases exponentially. One approach to reduce the search space will be to reduce the range of the variables. For example, none of the current integer solvers can efficiently support a 64-bit vector. Even assuming that solvers have such large integer support, it increases the search space enormously, thereby decreasing the performance of the solver. A solution to this problem is to split a large vector into small variables. For Example, a 32-bit vector *ir* can be split into 4 different variables, say, *ir_1*, *ir_2*, *ir_3* and *ir_4* each of size 8-bits. Each new variable has a range of 256 thereby decreasing the per variable search space for the solver. This can boost up the performance of the approach.

3. *Conditioned Slicing*: There has been some effort on program slicing in formal verification techniques [42, 43]. A similar approach can be followed to improve the performance in this scenario also. Consider a case where a *scenario* is sensitizing only a small portion of the circuit. In that case, conditioned slicing will create a very small integer model unlike the standard approach. This can boost the performance of the constraint based approach as the load on the solver will be reduced significantly reducing the memory and time requirement.

4. *High-level Testing*: This project has provided a new dimension for testing and verification: *Test generation at A^2M graph level.* A future direction of the research will be focused on high-level test generation directly from the A^2M graph instead of deriving an Integer or SAT model from the graph for

test generation. The basic testing techniques [5] could be applied on the A^2M graph which in-turn would make the whole process faster.

5. *Solver oriented modeling*: As observed during the experiments that the independent constraints make it easier for the solver to solve the model. One of the future directions can be to find ways to model the constraints from A^2M graph such that the dependence in the constraints is less hence improving the performance.

Appendix A
Structure of Integer Model

A.1 Ilog Model Structure

```
#include <ilsolver/ilosolver.h>
#include <vector>
ILOSTLBEGIN

#define top_dlx 0
#define im1 0
#define r_file 0
#define alu1 0
#define dm 0
class circuit
{
  public:
    vector <IloIntVarArray> dlx_ir;
    .
    .// declaring all the 2-d arrays for each signal in a2m file.

    void IloBitwiseXOR(IloModel & model, IloIntVar & a, IloInt a1, IloInt a2,
                        loIntVar & b, IloInt b1, IloInt b2, IloIntVar & c);
    .
    . // defining all bit-wise operators
```

```
            //functions to push a set of variables for a
            //module instantiation

            //one function per module
void dlx_variables(IloModel & model);
void alu_variables(IloModel & model);
void registers_variables(IloModel & model);
void data_memory_variables(IloModel & model);
void instr_memory_variables(IloModel & model);

            // functions to set constratins for each module
void dlx_constraints(IloModel & model, IloInt n);
void alu_constraints(IloModel & model, IloInt n);
void registers_constraints(IloModel & model, IloInt n);
void data_memory_constraints(IloModel & model, IloInt n);
void instr_memory_constraints(IloModel & model, IloInt n);

void create_model(IloModel & model)
{

   for(IloInt j=0; j<num_dlx_instantiations; j++)
                        //instantiating module-dlx
                        //(top module)
   {
      dlx_variables(model);
      dlx_constraints(model, j);
   }
   .
   .
   //similar loops for every module instantiation

   for(IloInt tf=0; tf<num_tfs; tf++)    //connections between modules
```

```
      {
         //constraints that connect modules are added here
      }
   }
                             //Output format is specified here
   bool solve(IloEnv & env, IloModel & model, IloInt tf);

                             //Scenarios constraints
                             //are written in this function
   void set_user_constraints(IloModel & model);
}


int main()
{
   IloEnv env;
   IloModel model(env);
   int tfs = 6;
   while (!found)     //do until solution is not found
   {
      circuit c(tfs);
      c.create_model(model);
      c.set_user_constraints(model);
      found = c.solve(env, model, tfs); //if solution is found
                                        //then "found" is set to true
      tf++;
   }

   //binary search on the number of time-frames can also be easily
   //implemented in a similar way

}
```

Bibliography

[1] S. Palnitkar, *Verilog HDL:A Guide to Digital Design and Synthesis*, Pearson Education, Singapore, 2004. 1.1, 3.1, 3.2.1, 3.2.2, 3.2.4.1, 1

[2] S. Palnitkar, *Design Verification with e*, Pearson Education, Singapore, 2005. 1.1

[3] I. Ghosh and M. Fujita, "Automatic test pattern generation for functional register transfer level circuits using assignment decision diagrams," *IEEE Trans.on Computer-aided Design of Integrated Circuits and Systems*, vol. 20, no. 3, pp. 402–415, March 2001. 1.2, 2.1.1, 2.2, 4

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A quantitative approach*, Morgan Kaufmann, USA, 2003. 1.3

[5] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, 2001. 2.1, 2.1.1, 2.1.2, 3.2.3, 3.2.3, 4

[6] I. Ghosh, L. Zhang, and M. Hsiao, "Automatic design validation framework for hdl descriptions via rtl atpg," in *Proceedings of the Asian Test Symposium*, 2003, pp. 148–153. 2.1.1, 4

[7] P. Vishakantaiah, J. A. Abraham, and D. G. Saab, "Cheetaa:composition of hierarchical sequential tests using atket," in *Proceedings of International Test Conference*, October 1993, pp. 606–615. 2.1.2

[8] R. S. Tupuri, A. Krishnamachary, and J. A. Abraham, "Test generation for gigahertz processors using an automatic functional constraint extractor," in *Proceedings of International Test Conference*, June 1999, pp. 646–652. 2.1.2

[9] J. Lee and J. H. Patel, "Architectural level test generation for microprocessors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 3, pp. 1288–1300, October 1994. 2.1.2

[10] A. Ghosh, S. Devadas, and A. R. Newton, "Sequential test generation and synthesis for testability at the register transfer and logic levels," *IEEE Transactions on Computer-Aided Design*, vol. 12, pp. 579–598, May 1993. 2.1.2

[11] V. M. Vedula, J. A. Abraham, J. Bhadra, and R. Tupuri, "A hierarchical test generation approach using program slicing techniques on hardware description languages," *Journal of Electronic Testing*, vol. 19, no. 2, pp. 149–160, April 2003. 2.1.2

[12] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Instruction based self testing of processor cores," *J.Electron.Test.*, vol. 19, no. 2, pp. 103–112, 2003. 2.1.2

[13] L. Cheni, S. Ravi, A. Raghunathan, and S. Dey, "A scalable software-based self-test methodology for programmable processors," in *Proceedings of 40th Design Automation Conference*, June 2003, pp. 548–553. 2.1.2

[14] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 3, pp. 369–380, March 2001. 2.1.2

[15] W. C. Lai, A. Krstic, and K. T. Cheng, "On testing the path delay faults of a microprocessor using its instruction set," in *Proceedings of IEEE VLSI Test Symposium*, May 2000, pp. 15–20. 2.1.2

[16] T. Larrabee, "Test pattern generation using boolean satisfiablilty," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 1, pp. 4–15, January 1992. 2.1.2

[17] R. E. Bryant, "Graph based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. C, pp. 677–691, August 1986. 2.1.2

[18] K. S. Brace, R. Rudell, and R. E. Bryant, "Efficient implementation of the bdd package," in *Proceedings of the Design Automation Conference*, 1990, pp. 40–45. 2.1.2

[19] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in sat-based formal verification," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, April 2005. 2.1.2

[20] S. Ravi, G. Lakshminarayana, and N. K. Jha, "Tao: Regular expression based register transfer level testability analysis and optimization," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 9, no. 6, pp. 824–832, December 2001. 2.1.2

[21] S. Ravi and N. K. Jha, "Fast test generation for circuits with rtl and gate level views," in *Proceedings of the International Test Conference*, 2001, pp. 1068–1077. 2.1.2

[22] L. Lingappan, S. Ravi, and N. K. Jha, "Satisfiability-based test generation for nonseparable rtl controller-datapath circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 3, pp. 544–557, March 2006. 2.1.2, 2

[23] T. Li, Y. Guo, and S. K. Li, "Assertion-based automated functional vectors generation using constraint logic programming," in *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI*. 2004, pp. 288–291, ACM Press. 2.1.2

[24] R. Vemuri and R. Kalyanaraman, "Generation of design verification tests from behavioral vhdl programs using path enumeration and constraint programming," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 3, no. 2, pp. 201–214, 1995. 2.1.2

[25] A. K. Chandra and V. S. Iyengar, "Constraint solving for test case generation: a technique for high-level design verification," in *Proceedings of International Conference on Computer Design: VLSI in Computers and Processors*, 1992, pp. 245–248. 2.1.2

[26] F. Fallah, S. Devadas, and K. Keutzer, "Functional vector generation for hdl models using linar programming and boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, pp. 994–1002, August 2001. 2.1.2

[27] C. Y. Huang and K. T. Cheng, "Assertion checking by combinated word-level atpg and modular arithmatic constraint solving techniques," in *Proceeding of Design Automation Conference*, 2000, pp. 118–123. 2.1.2

[28] F. Fallah, S. Devadas, and K. Keutzer, "Functional vector generation for hdl models using linar programming and 3-satisfiability," in *Proceeding of Design Automation Conference*, 1998, pp. 528–533. 2.1.2

[29] Z. Zeng, P. Kalla, and M. Ciesielski, "Lpsat: A unified approach to rtl satifiability," in *Proceeding of DATE conference*, 2001, pp. 398–4002. 2.1.2

[30] D. Lewin, L. Fournier, M. Levinger, E. Roytman, and G. Shurek, "Constraint satisfaction for test program generation," *IEEE International Phoenix Conference on Communication and Computers*, pp. 45–48, 1995. 2.1.2

[31] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek, "Test program generation for functional verification of powerpc processors in IBM," in *Design Automation Conference*, 1995, pp. 279–285. 2.1.2

[32] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-pro: Innovations in test program generation for functional processor verification," *IEEE Design and Test of Computers*, vol. 21, no. 2, pp. 84–93, Mar/Apr 2004. 2.1.2

[33] J. Yuan, C. Pixley, A. Aziz, and K. Albin, "A framework for constrained functional verification," in *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, Washington, DC, USA, 2003, pp. 142–145, IEEE Computer Society. 2.1.2

[34] A. Adir, H. Azatchi, E. Bin, O. Peled, and K. Shoikhet, "A generic micro-architectural test plan approach for microprocessor verification," in *DAC '05: Proceedings of the 42nd annual conference on Design automation*, New York, USA, 2005, pp. 769–774, ACM Press. 2.1.2

[35] F. Corno, P. Prinetto, and M. S. Reorda, "Testability analysis and atpg on behavioral rt-level vhdl," in *Proceedings of International Test Conference*, 1997, pp. 753–759. 2.1.2

[36] S. Chuisano, F. Corno, and P. Prinetto, "Rt-level tpg exploiting highlevel synthesis information," in *Proceedings of VLSI Test Symposium*, 1999, pp. 341–346. 2.1.2

[37] D. Prasad, R. Archana, V. Karthik, Senthilkumar, V. Kamakoti, S. M. Kailasnath, and V. M. Vedula, "A novel unified framework for functional verification of processors using constraint solvers," in *Proceedings of VLSI Design and Test symposium (VDAT)*, 2006, pp. 418–426. 2.1.2

[38] V. Chaiyakul and D. D. Gajski, "Assignment decision diagram for high-level synthesis," Tech. Rep. 92-103, University of California, Irvine, CA, December 1992. 2.2, 4, 3.2.2.1

[39] ILOG, "Technology web site," http://www.ilog.com, 2006. 2.2, 3

[40] V. Chaiyakul, D. D. Gajski, and L. Ramachandran, "High-level transformations for minimizing syntatic variances," in *Proceedings of Design Automation Conference*, June 1993, pp. 413–418. 4

[41] K. Najeeb, V. V. Konda, S. K. S. Hari, V. Kamakoti, and V. M. Vedula, "Power virus generation using behavioural models of circuits," in *Proceedings of VLSI Test Symposium*, May 2007. 5

[42] S. Vasudevan, V. Viswanath, and J. A. Abraham, "Efficient microprocessor verification using antecedent conditioned slicing," in *Proceedings of the International Conference on VLSI Design*, January 2007. 3

[43] V. M. Vedula, J. A. Abraham, and J. Bhadra, "Program slicing for hierarchical test generation," in *Proceedings of* 20th *IEEE VLSI Test Symposium*, 2002, pp. 237–243. 3

List of Publications

1. Siva Kumar Sastry Hari, Vishnu Vardhan Reddy Konda, V. Kamakoti, Vivekananda M Vedula and Kailasnath S Maneperambil, "**Automatic Constraint Based Test Generation for Behavioral HDL Models**", submitted to *IEEE Transactions on VLSI Systems for a Special Section on Design Verification and Validation: Theory and Techniques*

2. K. Najeeb, Siva Kumar Sastry Hari, Vishnu Vardhan Reddy Konda, V. Kamakoti and Vivekananda M Vedula, "**Test Generation for Peak Power Dissipation Using Behavioral Models of Circuits**" submitted to *ACM Transactions on Design Automation of Electronic Systems for a Special Issue on High Level Design Validation and Test*

3. K. Najeeb, Vishnu Vardhan Reddy Konda, Siva Kumar Sastry Hari, V. Kamakoti and Vivekananda M Vedula "**Power Virus Generation Using Behavioural Models of Circuits**", *In the Proceedings of* 25th *IEEE VLSI Test Symposium*, 2007, pp. 35-42