Gate Level Coverage of a Behavioral Test Generator

by

Gunjeetsingh Baweja

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

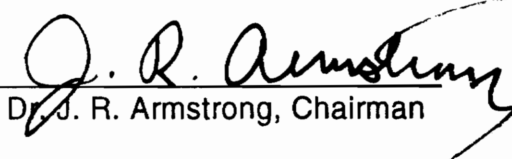
in partial fulfillment of the requirements for the degree of

Master of Science

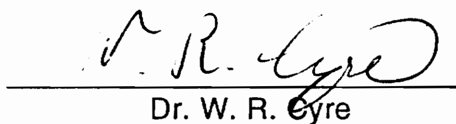
in

Electrical Engineering

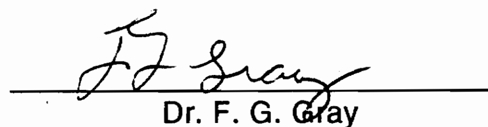
APPROVED:



Dr. J. R. Armstrong, Chairman



Dr. W. R. Cyre



Dr. F. G. Gray

March, 1993

Blacksburg, Virginia

LD
5655
V855
1993
B394
C.2

C.2

Gate Level Coverage

of a Behavioral Test Generator

by

Gunjeetsingh Baweja

Dr. J. R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

Use of traditional gate level test generation techniques is prohibitively expensive and time consuming for VLSI chips. High level approaches to test generation have been proposed to improve the efficiency of test generation, e.g., the Behavioral Test Generator developed at Virginia Tech generates test vectors from high level Behavioral VHDL descriptions. To validate the utility of these test vectors, it needs to be established that they provide adequate coverage at the gate level. This thesis shows that test vectors obtained from the Behavioral Test Generator provide adequate coverage for the equivalent gate level circuit. A system that was developed to effectively evaluate the test vectors is presented. The implementation of Heuristic Test Generator to improve the coverage of the Behavioral Test Generator is explained.

Acknowledgments

I would like to thank my advisor Dr. J. R. Armstrong for the challenging environment he made available, and the support and guidance throughout this research. I would like to thank Dr. W. R. Cyre and Dr. F. G. Gray for serving as members of my committee. I would like to thank my friends and colleagues for their support and encouragement.

Most of all, I would like to thank my parents for their love and support through all my endeavors.

Table of Contents

Chapter 1. Introduction 1

1.1 Contents 3

1.2 List of Contributions 4

Chapter 2. Literature Review 5

2.1 Faults and Testing 5

2.2 Gate Level Test Generation Methods 6

2.3 Functional Level Test Generation 8

2.3.1 Register Level Test Generation 8

2.3.2 Test Generation with Binary Decision Diagrams 10

2.3.3 Test Generation with Hardware Description Languages 10

Chapter 3. Behavioral Test Generator 13

3.1 Behavioral Fault Model 14

3.1.1 Control Faults 14

3.1.2 Micro-operation Faults 16

3.2 Test Generation Approach 16

3.3 VHDL Subset 17

3.4 Implementation of Behavioral Test Generator 18

3.4.1 Test Generation Preprocessor 18

3.4.2 Test Generator 23

Chapter 4. Heuristic Test Generator 24

4.1 Implementation of Heuristic Test Generator 25

4.2 Effect of Using the Heuristic Test Generator 32

Chapter 5. Gate Level Fault Coverage Determination 35

5.1 Criteria for Evaluation of Test Vectors 35

5.2 System for Gate Level Fault Coverage Determination 35

5.2.1 Fault Simulator : HIFAULT 37

5.2.2 Synopsys Synthesis Tool 44

5.3 EDIF to GHDL Conversion 49

Chapter 6. Results 51

6.1 Selection of Models 51

6.2 Register Level Primitives 52

6.2.1 Combinational Logic Primitives 52

6.2.2 Sequential Primitives 61

6.3 Big Models 67

6.4 Conclusions from the Fault Coverage Results 76

Chapter 7. Proposed Future Work 77

7.1 Automatic Conversion of VHDL Circuit Description to Prolog Facts 77

7.2 Automatic Synopsys EDIF to GHDL Conversion 78

7.3 Use of BTG as a Process Test Generator for Hierarchical Test Generator 78

7.4 Synthesis Directed Test Generation 79

Chapter 8. Conclusion 80

Bibliography 81

Appendix A. BTG2DWL Users Guide 84

Appendix B. Circuit Models 86

Vita 158

List of Illustrations

Figure 1. Test Generation Flow 17
Figure 2. Behavioral Test Generator 19
Figure 3. VHDL Behavioral Description of ALU 20
Figure 4. Prolog Facts of ALU Circuit Description 20
Figure 5. .ife File of ALU 21
Figure 6. Fault List of ALU 22
Figure 7. Test Vector for Fault 2 23
Figure 8. Heuristic Test Generator Interface to BTG 25
Figure 9. VHDL Description of Adder 30
Figure 10. Prolog Facts Representation of Adder 30
Figure 11. Fault List 30
Figure 12. Test Vector Generated without HTG 31
Figure 13. Test Vectors Generated after Implementation of HTG 31
Figure 14. Increase in Fault Coverage 32
Figure 15. Gate Level implementation of Adder 33
Figure 16. Cumulative % Coverage for Test Vector 34

Figure 17. Test Setup for Determining Fault Coverage 36
Figure 18. HIFAULT Overview 37
Figure 19. VHDL Description of ALU 38
Figure 20. Implementation of ALU 39
Figure 21. GHDL Description of ALU 40
Figure 22. Test Vectors for an Adder Generated by BTG 41
Figure 23. DWL Representation of BTG Test Vectors 42
Figure 24. Fault Summary after Fault Simulation 43
Figure 25. Hard Detects and Potential Detects 44
Figure 26. Overview of Synopsys Synthesis Tool 45
Figure 27. VHDL Description of 32-Function ALU 46
Figure 28. Gate Level Schematic of 32 Function ALU 48
Figure 29. Circuit Diagram of CKA 52
Figure 30. Fault Coverage Results of CKA 53
Figure 31. Fault Coverage Results of Adder 54
Figure 32. Fault Coverage Results of Gate Level Implementation of Adder 55
Figure 33. Fault Coverage Results of Look Ahead Carry Adder 56
Figure 34. Fault Coverage Results of Subtracter 57
Figure 35. Fault Coverage Results of 4_TO_1 Multiplexer 58
Figure 36. Fault Coverage Results of Quad 4_TO_1 Multiplexer 59
Figure 37. Fault Coverage Results of Decoder 60
Figure 38. Fault Coverage Results of JK Flip-Flop 61
Figure 39. Fault Coverage Results of Register 62
Figure 40. Fault Coverage Results of Counter 63
Figure 41. Fault Coverage Results of Barel Shifter 64

Figure 42. Gate Level Schematic of Barel Shifter 65
Figure 43. Fault Coverage Results of Four Function ALU 67
Figure 44. Fault Coverage Results of Linear Feedback Shift Register 68
Figure 45. Fault Coverage Results of Controlled Counter 70
Figure 46. Fault Coverage Results of 32 Function ALU 72
Figure 47. Fault Coverage Results of Microprocessor Slice 74
Figure 48. Gate Level Schematic of Microprocessor Slice 75

List of Tables

Table 1. Test Patterns 26

Table 2. Test Vectors 27

Table 3. Test Vectors used for Test Generation 28

Table 4. Truth Table of JK Flip Flop 61

Table 5. Fault Coverage Results of Register Level Primitives 66

Table 6. Truth Table of Control Decoder 69

Table 7. Function Table of Controlled Counter 69

Table 8. Functions Implemented by 32 Function ALU 71

Table 9. Selection of Source Operands 73

Table 10. Functions Implemented by Microprocessor Slice 74

Chapter 1. Introduction

VLSI technology has advanced at a very rapid rate in the past few years. The number of transistors on a chip has increased from a few hundred transistors to a few million transistors, e.g., the Intel 486DX microprocessor contains 1.2 million transistors [1]. Traditional gate level test generation techniques are not suitable for these new VLSI chips. It has been shown that gate level test generation exponentially increases with the increase in the number of input lines and gates. Thus fault generation programs at the gate level are NP-complete problems. Hence for reasonable cost and time it is feasible to have automatic test generation at the gate level only for SSI (small scale integration) and MSI (medium scale integration) circuits [2]. There is a need for more efficient test generation techniques to ensure the reliability of VLSI products. The efficiency of test generation can be improved using the higher level approach for test generation. A number of higher level approaches to test generation for VLSI circuits have been proposed [3-18], but the question arises as to the validity of the behaviorally generated test when used on actual circuits.

A "Behavioral Test Generator" (BTG) was developed at Virginia Tech. The BTG is implemented in Prolog and makes use of VHDL (VHSIC Hardware Description Language) descriptions of the circuit under test to generate test waveforms. Behavioral faults are identified in the VHDL description. Test waveforms are then generated which would detect these behavioral faults.

The Behavioral Test Generator has demonstrated that it can generate tests from behavioral VHDL chip descriptions [16]. It needs to be established that test vectors generated using the BTG provide adequate coverage at the gate level. This thesis presents a system that can be used to effectively evaluate the gate level fault coverage of test vectors. This system can be used to determine the fault coverage of any functional test generator, which generates test vectors from a Behavioral VHDL description of the circuit.

The previous implementation of Behavioral Test Generator did not provide adequate coverage at the gate level for certain circuits. Fault coverage improvement was achieved by implementation of a Heuristic Test Generator (HTG). The Heuristic Test Generator generates multiple test vectors for big micro-operation faults to improve coverage.

Finally, it is shown that test vectors generated using Behavioral Test Generator combined with a Heuristic Test Generator provide adequate coverage for the equivalent gate level circuits.

1.1 Contents

Chapter 2, "Literature Review", explains about testing and faults, discusses gate level and functional level test generation techniques.

Chapter 3, "Behavioral Test Generator", explains and discusses the development of the Behavioral Test Generator.

Chapter 4, "Heuristic Test Generator", explains the interface of the Heuristic Test Generator to the Behavioral Test Generator and discusses the Heuristic Test Generator.

Chapter 5, "Gate Level Fault Coverage Determination", describes with examples the system that is used to obtain the equivalent gate level model and the coverage of the test vectors.

Chapter 6, "Results", discusses the coverage results obtained for the test vectors generated by the Behavioral Test Generator for various circuit models.

Chapter 7, "Proposed Future Work", proposes possible future enhancements and suggests new approaches for test generation.

Chapter 8, "Conclusion".

Appendix A, "BTG2DWL Users Guide", a guide on how to use the BTG2DWL program, which converts the test vectors in BTG format to Design Waveform Language Format.

Appendix B, "Circuit Models", gives a behavioral VHDL description, a Prolog representation of the VHDL description and the equivalent gate level GHDL description for each circuit model for which fault coverage was determined.

1.2 List of Contributions

The primary objective of this thesis was to determine the gate level fault coverage of the Behavioral Test Generator. The contributions made by this thesis include the following:-

- Improved the fault coverage of the BTG by implementation of the HTG.
- Implemented the BTG2DWL, to convert the test vectors in the BTG format to Design Waveform Language (DWL).
- Developed a system to evaluate the fault coverage of the test vectors generated using a Behavioral Test Generator.
- Determined the fault coverage of register level primitives and other big models.
- Established that test vectors generated using the BTG combined with the HTG provide high coverage of the equivalent gate level circuit.

Chapter 2. Literature Review

2.1 Faults and Testing

Faults in a digital circuit can cause the circuit to malfunction. These faults are usually a result of imperfect manufacturing processes. To determine that faults exist in the digital circuits test vectors are applied to the circuit under test. The test vector will detect the fault if the observed behavior of the chip is different from the expected behavior of a working chip. The behavior of circuit can be expressed at various levels of abstractions. Traditionally digital circuits were represented at the gate or transistor level. As the complexity of digital circuits has increased, due to rapid advancements in VLSI, the digital circuits are now expressed at the register or chip level.

Test generation is the process of determining the optimum set of test vectors necessary to test a digital system. The method of test generation is dependent on the fault model used. The fault model is closely related to the type of modeling system used. At the gate level the fault model is the *stuck-at fault model* and the *stuck-open fault model* . The corresponding circuit fault to the stuck-at fault is the

signal being stuck at a fixed voltage level [19]. The stuck open fault model assumes an open due to the breaking of the connection. For functional level test generation, fault models are created according to the functional behavior of the circuits.
Some of the methods of test generation at the gate level and functional level are now reviewed.

2.2 Gate Level Test Generation Methods

Gate level test generation can be used very effectively for SSI and MSI circuits. The first algorithmic method of automatic test generation at the gate level, the D-algorithm was proposed by Roth in 1966 [20]. The D-algorithm is based on D-calculus. D represents a good value of 1 and a faulty value of 0. $\overline{D}$ represents a good value of 0 and a faulty value of 1. Fault sensitization is done by justifying the symbol D or $\overline{D}$ at the fault site. The D or $\overline{D}$ are then propagated to the primary outputs, by assigning values to the primary inputs. Justifying the values to the primary inputs may cause conflicts, and thus the D-algorithm uses backtracking to resolve the conflicts.

The 9-V Algorithm was proposed by Cha, Donath and Ozguner in 1978 [21]. The main difference between the 9-V algorithm and the D-algorithm was the use of nine values in 9-V algorithm. The benefit of the flexibility provided by the nine values is that it reduces the amount of search done for multiple path sensitization.

PODEM (Path-Oriented Decision Making) was proposed by Goel in 1981 [22]. This Test Generation algorithm is characterized by a direct search process. In this

method the decisions consist only of primary input assignments. Goel used the same symbols D or $\overline{D}$ for fault sensitization. The method of test generation is as follows. First, values are assigned to the Primary Inputs (PI) to set the output of the gate under test to D or $\overline{D}$. It is then verified that these assigned values are sufficient to sensitize the fault. If the fault is not yet sensitized, additional values are assigned to other primary inputs. After the fault has been sensitized with D or $\overline{D}$, the value is propagated to the next gate towards a primary output by assigning values to the primary inputs [22]. The values of the internal signals due to the primary inputs are determined. Knowing the implied logic values at internal signals can avoid the conflict in logic value assignment of a signal when continuing propagation of D or $\overline{D}$ towards a primary output. Further, the direct search method used by PODEM simplifies the backtracking process, as only the primary input values have to be changed and the internal state need not be restored to their previous values as was required in case of the D-algorithm. Thus backtracking is done by simulation. The PODEM algorithm has been found to be more efficient then the D-algorithm.

FAN (Fanout-Oriented Test Generation) was proposed by Fufiwara and Shimono in 1983 [23]. The fault model for the algorithm is the stuck-at fault and D and $\overline{D}$ are used for fault sensitization. The FAN algorithm improves upon PODEM, by stopping the backtracking at internal lines rather than at the primary inputs. Thus the backtracking to justify the values can be reduced and test generation time is reduced. Further FAN uses multiple-backtrack procedure that attempts to simultaneously satisfy a set of objectives.

2.3 Functional-level Test Generation

As the digital circuits become increasingly complicated, test generation at the gate level has become prohibitively expensive. By doing test generation at higher levels of abstractions, the cost of test generation can be reduced. Other advantages of functional level test generation are as follows:-

Usually the functionality of a chip does not change, but the implementation technology changes. The tests that are generated from the function description of a chip can be useful throughout the functional life of the chip.

Functional testing can be used to verify the functional operation according to the design specifications of the circuit. If it is established that these tests provide adequate coverage at the gate level, the cost of test generation, can be greatly reduced.

Many functional-level test generation techniques have been proposed. A few of the functional-level test generation techniques are reviewed.

2.3.1 Register Level Test Generation

The register level is above the gate level. The primitives at the register level are register, counters, multiplexers, etc. The behavior of the circuit is usually represented in the Register Transfer Language (RTL), that represent the data flow

between the various primitives at the Register Level. The fault models that may be used at this level are stuck-at-fault, micro-operation fault and register transfer fault.

The Register-level Test Generator (RTG) was developed by Shteingart, Nagle and Grason in 1985 [5]. The RTG was implemented for test generation for SSI, MSI and small LSI circuits. The RTG system was designed to detect all classical "stuck-at" faults in the digital circuits. The RTG used the gate level model for combinational components and register level models for sequential components. It used built in truth tables to simulate the behavior of combinational components and a set of primitive routines for sequential components. The RTG used the 9-V algorithm for propagation and justification through combinational part, and used the primitive routines for the sequential components.

The S-Algorithm, was developed by Lin and Su to generate functional test vectors for circuits that are modeled with a register transfer language (RTL) [6] The algorithm used symbolic execution techniques to generate test patterns that can be used for detecting faults. The fault model was based on RTL statement syntax and nine types of register transfer level faults were derived. These were, label fault, jump fault, timing fault, condition fault, data storage fault, data transfer fault, register decoding fault, operator decoding fault and operator execution fault. Symbolic execution is used to find expressions representing good and bad values in terms of registers and input values. This execution includes path constraints and values needed to select an execution path through particular statements. The input values are then selected such that the path constraints are satisfied and good and bad executions differ. Once the values are determined they constitute a test.

2.3.2 Test Generation with Binary Decision Diagrams

The functional behavior of the circuit can be represented as a binary decision diagram [9]. A binary decision diagram is a graph model of the function of the circuit. The traversal through the graph is determined by examining the value at each node. At each node the left or right branch is followed depending on the value (0 or 1) of the corresponding input variable. The fault model of the circuits represented as a binary decision diagram is defined in the following two ways: stuck-at-faults at the input lines, output lines, or internal memory elements and functional faults that effect the execution of one of the experiments of the binary decision diagram [10]. An experiment consists of finding a complete path from the input to the output. The complete path from the input to the output constitutes a test. A complete test set is generated when all the paths have been traversed. The binary decision diagrams are suitable only for simple functional models [8]. The complexity increases for big models and hence this method is no longer suitable for test generation.

2.3.3 Test Generation with Hardware Description Languages

The functional behavior of the circuit can be represented with hardware description languages. The use of hardware description languages such as VHSIC Hardware Description Language (VHDL) has become very popular. Some methods of test generation that use the input circuit description in Hardware Description Languages are now reviewed.

Leveldel and Menon introduced an algorithm based on an extension of the D-algorithm for generating test vectors from the hardware description language model of digital circuits in 1982 [17]. The fault model consisted of stuck-at-1 or stuck-at-0 on the input and output lines, functional faults and control faults. The test generation process used the D-algorithm approach. The D-algorithm approach was used to generate D-equations and D-propagation cubes for each functional module and control construct in the HDL description. This derivation of D-equations and D-propagation cubes for relatively complex functional modules is very complicated and hence test generation efficiency is reduced using this algorithm.

The E-Algorithm was proposed by F.E.Norrod in Feb. 1988 [18]. It generated tests from circuits described in VHDL. The VHDL description is transformed into a graphical representation. The representation is as a directed graph with the nodes representing microoperations and arcs representing data flow. The fault model is based on the data and control logic paths in the circuits. The test generation process is in 3 steps: fault sensitization, propagation, and justification. The E-Algorithm is an extension of the D-Algorithm, with additional rules for handling the control structures. The D-algorithm cannot handle fault propagation through control constructs. Therefore, a special method called E-propagation is used to propagate the fault effect through control constructs [18].

The Behavioral Test Generator (BTG) has been implemented at Virginia Tech. It generates test vectors from a VHDL behavioral description. The next chapter gives a detailed description of the BTG.

The FunTestIC (FUNctional TEST pattern generation for Integrated Circuits) for test generation for circuits, modules and systems modeled in VHDL was proposed by H.D. Hummer, H. Veit and H. Topfer in 1991 [24]. In this approach the circuit model is represented in structures of a Control Flow Graph (CFG), Data Flow Graph (DFG), and Sequence Graph (SG). The CFG resembles a flow chart representation of the circuit model. The DFG represents the possible data flows between the variables of the models. The SG gives information about the sequence in which the data has to be passed in the CFG. In the FunTestIC algorithm, a primary input and a primary output are selected in the DFG. The path from the primary input to the primary output is a complete control/observation path. The test is said to be complete when all the DFG edges have been traversed at least once by the paths. The corresponding test pattern sequence for a given path is achieved by two different methods, Time-Reverse Processing (FunTestIC-1) and Simulation (FunTestIC-2). The FunTestIC-1 is not suitable for sequentially deep models and hence the FunTestIC-2 is used.

Chapter 3. Behavioral Test Generator

The Behavioral Test Generator (BTG) was developed at Virginia Tech. It has shown to be effective in generating tests for multi-process VHDL behavioral models. The BTG is based on a behavioral fault model that was first proposed by D. S. Barclay and J. R. Armstrong [11]. This algorithm was implemented in Prolog, an artificial intelligence language. BTG automatically generates tests for circuits with behavioral VHDL descriptions. For the original algorithm the computer time for test generation was high as it used low-level goals during the test generation process [11]. The algorithm was improved by O'Neill and Armstrong [14]. High-level goals, such as propagation, justification, and execution were used to replace the low-level goals. This resulted in speedup of the test generation process. Jani and Armstrong further improved the algorithm [15]. An improved timing model was developed. The model used a symbol 'R' to represent a '0' to '1' transition. The method of solving the high-level goals (propagation, justification, execution) was also improved to achieve higher test generation efficiency [15]. The algorithm was further developed by Lam to handle reconvergent fanout and feed-back during behavioral level test generation [16].

3.1 Behavioral Fault Model

The BTG fault model is defined in terms of VHDL constructs. This behavioral fault model has three fault types [4,12]:

- *Control Faults:* perturb the control points that switch between incorporation sequences
- *Micro-operation faults:* perturb individual micro-operations.
- *Stuck Data Faults:* Cause the data lines to be Stuck-at-0 or Stuck-at-1.

3.1.1 Control Faults

Control faults change the order of execution or inhibit execution of certain constructs. The various types of control faults defined for the Behavioral Test Generator are:

(i) IF - stuck THEN, stuck ELSE

This fault is defined for the VHDL if-then-else construct. As shown in an example below, the THEN clause will be executed independent of the value of the control expression. A stuck ELSE fault is similarly defined for else clause.

```
if (X = '1')  then independent of the value of X the
 ----------        then clause is executed
 else
 ----------
 end if;
```

(ii) CASE - dead clause

This fault is defined for the VHDL case construct. The clause under the case statement that is selected by the control expression is assumed to be "dead", i.e. this clause will not be executed. In the example shown below, even though X="00", the first clause is not executed

```
case X(0 to 1) is
 when "00" = > F <=  A;   -- Not executed
 when "01" = > F <= not (A);
 when "10" = > F <= A + B;
 when "11" = > F <= A and B;
```

(iii) Assignment fault

In this fault the assignment statement does not execute. Hence the destination object of an assignment statement does not receive the result from the source expression, e.g., in the assignment statement no assignment is made to A.

```
A  <=  B;    -- no assignment occurs
```

(iv) Dead process fault

In this fault a process is inhibited from being activated. Thus in the process statement shown below the process P would not be activated even though X or Y changes.

```
ADD: process(X,Y)
 --------
 begin
 ---------
 end process ADD;
```

The dead process fault can be tested by testing the assignment control fault on one of the assignment statements within the process [16]. Hence the dead process fault is usually not considered separately.

3.1.2 Micro-operation Faults

Micro-operation faults are modeled by perturbing one micro-operation to another. There are two microoperations types: logical and arithmetic. The list of the micro-operations of the two types and examples of the perturbation is shown below.

(i) Logical Micro-operations

NOT, AND, OR, XOR
good: Z <= X and Y; faulty: Z <= X or Y

(ii) Arithmetic Micro-operations

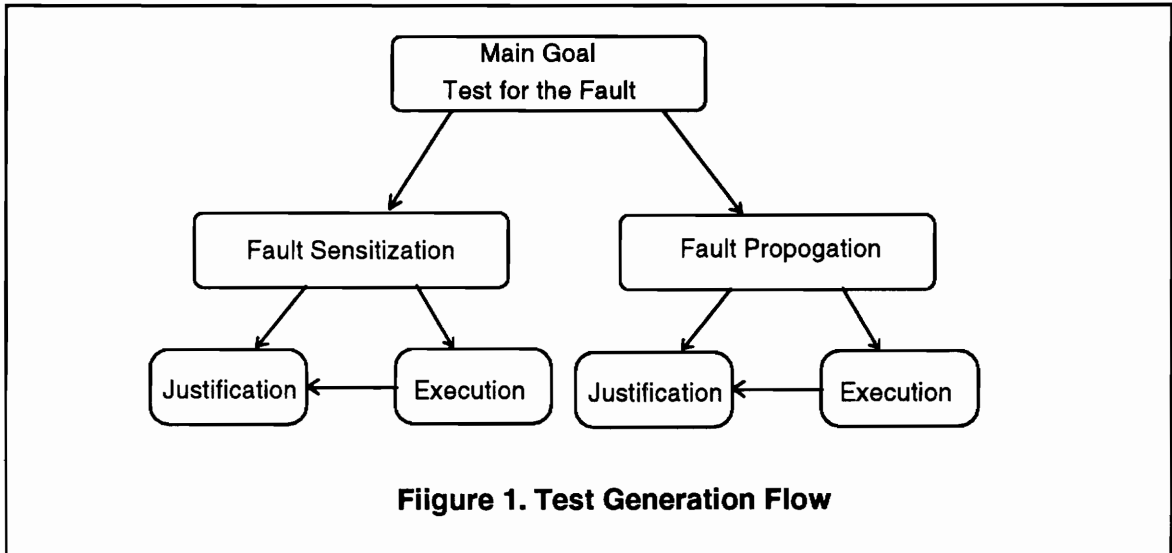
ADD, SUB, COMP, INC, DEC
good: Z <= ADD(A, B) faulty: Z <= SUB(A, B)

3.2 Test Generation Approach

The Behavioral Test Generator has been implemented in Prolog. It uses the goal tree structure to organize the test generation process. The main goal is to generate a test for a particular fault. This main goal is reduced to primitive goals. The primitive goals are to assign appropriate values to the primary inputs and observe the primary outputs [15].

The main goal is solved by solving the next stage of subgoals i.e., fault sensitization and fault propagation. Fault sensitization and fault propagation are solved by satisfying the next level of sub-goals of *justification* and *execution*. The goal *justification* is: given an expression, and the desired value, to determine the input values that place the value in the expression. The goal *execution* determines the sequence of control statements that govern the execution of a statement and

determines the values required for the expressions in the control statements. The justification goal is called to determine the input values, to solve the execution goal. These sub-goals are further broken down to the next level of subgoals, until the sub-goals are primitive goals. The test generation flow is as shown in Figure 1.



Fiigure 1. Test Generation Flow

3.3 VHDL Sub-set

The Input to the BTG is the VHDL behavioral description of the circuit. The BTG generates test vectors based on the behavioral fault model. The VHDL behavioral description is restricted to the use of a sub-set of VHDL [16]. The sub-set includes the following.

- Signal objects (Variable Objects are not supported)
- Signals of type BIT and BIT_VECTOR
- Multiple process statements

- Concurrent signal assignment statements
- Delta delay; test generation assume delayless gates
- If and case Statements
- Basic Operations

  Logical Operators AND, OR, XOR, NOT and EQV

  Unsigned arithmetic operators Add and Sub

  Less than and less than or equal to

  Concatenation and slices of bit vectors

3.4 Implementation of Behavioral Test Generation

The *Behavioral Test Generator* (BTG) is implemented in Prolog. The BTG consists of two parts; the *Test Generator Preprocessor* and the *Test Generator.* The complete Behavioral Test Generation System is as shown in Figure 2 [16].

3.4.1 Test Generation Preprocessor

The Test Generation preprocessor consists of the Intermediate File Extractor and the Fault List Extractor.

The circuit to be tested has to be represented using the VHDL subset given above. The VHDL Source file is then converted to Prolog facts. These Prolog facts are stored in an .hdl file. Consider for example the VHDL description of an ALU as shown in Figure 3. Note that statements are given numbered labels, e.g., s1,s2. The corresponding prolog facts are as shown in Figure 4.

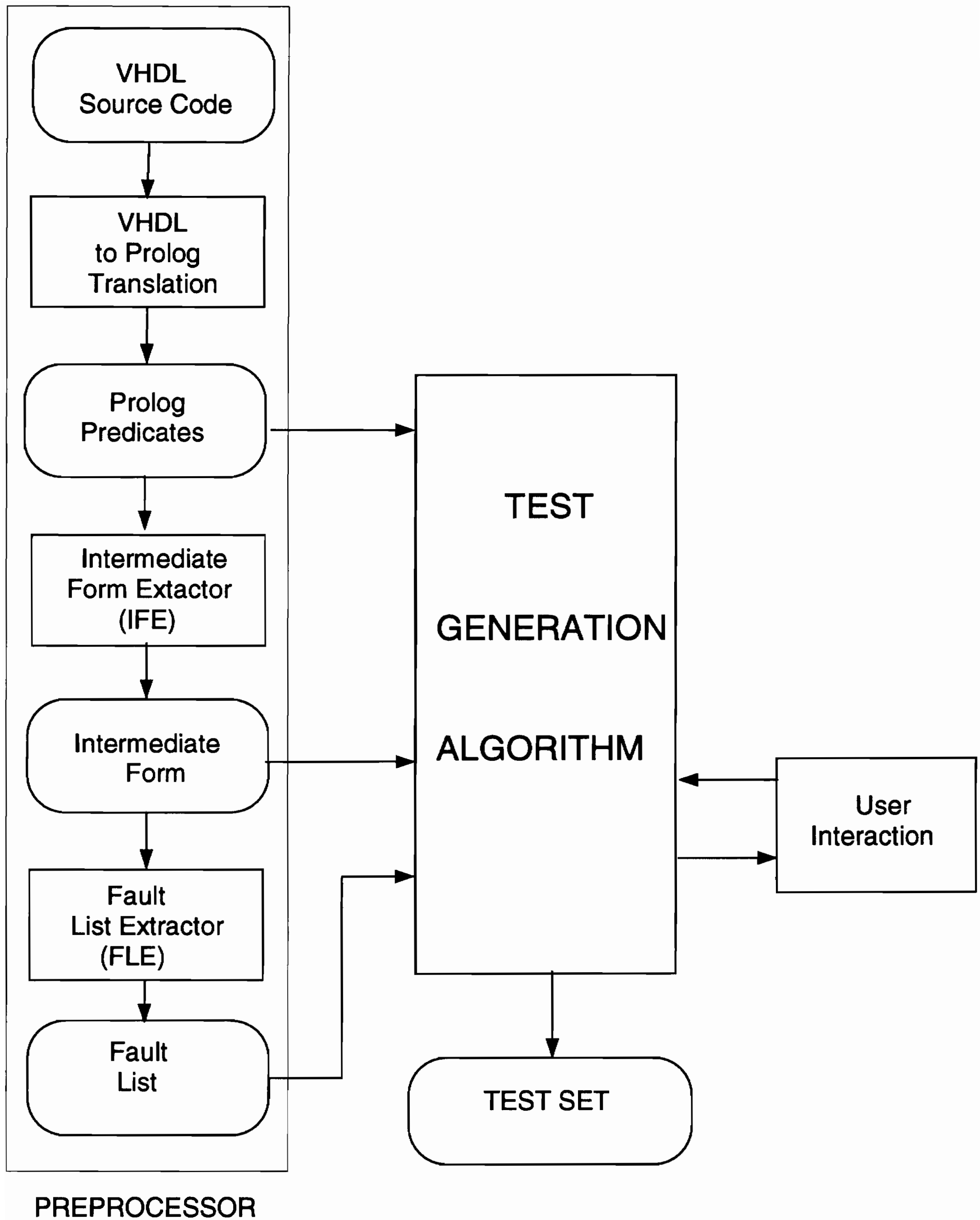


Figure 2. Behavioral Test Generator

```
-- VHDL description of ALU

library SYNOPSYS;
use synopsys.bv_arithmetic.all;

entity ALU is
port(A,B: in BIT_VECTOR(3 downto 0);
    FSEL:in BIT_VECTOR(1 downto 0);
    F:out BIT_VECTOR(3 downto 0));
end ALU;

architecture BEH of ALU is

begin

process(A,B,FSEL)
begin

s1: case FSEL is

s2:   when "00" => F <=A;
s3:   when "01" => F <=not(A);
s4:   when "10" => F <=A + B;
s5:   when "11" => F <=A and B;

end case;
end process;

end BEH;
```

Figure 3. VHDL Behavioral Description of ALU

```
modelspec :-

assert(fileprefix("alu")),
assert(modelname("ALU")),

assert(datatype(a,bv)), assert(bvlength(a,4)), assert(inputpin(a)),
assert(datatype(b,bv)), assert(bvlength(b,4)), assert(inputpin(b)),
assert(datatype(fsel,bv)), assert(bvlength(fsel,2)), assert(inputpin(fsel)),
assert(datatype(f,bv)), assert(bvlength(f,4)), assert(outputpin(f)),

assert(statementtype(s1,case)),
assert(controlexpression(s1,[obj,fsel])),
assert(subordinaterange(s1,[[bv,"00"]],[s2])),
assert(subordinaterange(s1,[[bv,"01"]],[s3])),
assert(subordinaterange(s1,[[bv,"10"]],[s4])),
assert(subordinaterange(s1,[[bv,"11"]],[s5])),

assert(statementtype(s2,assignment)),
assert(destinationobject(s2,f)),
assert(sourceexpression(s2,[obj,a])),

assert(statementtype(s3,assignment)),
assert(destinationobject(s3,f)),
assert(sourceexpression(s3,[bvnot,[obj,a]])),

assert(statementtype(s4,assignment)),
assert(destinationobject(s4,f)),
assert(sourceexpression(s4,[bvadd,[obj,a],[obj,b]])),

assert(statementtype(s5,assignment)),
assert(destinationobject(s5,f)),
assert(sourceexpression(s5,[bvand, [obj,a], [obj,b]])).
```

Figure 4. Prolog Facts of ALU Circuit Description

These Prolog facts are given as an input to the Intermediate Form Extractor (IFE). The IFE has been implemented in Prolog. The IFE produces an .ife file. This file contains the Prolog facts that represent the knowledge of the circuit and information on selecting paths for propagation, justification and execution. The .ife file for the ALU is as shown in Figure 5.

```
objects([a, b, fsel, f]).
clauses(s1, [[[bv,[48,48]]], [[bv,[48,49]]],
[[bv,[49,48]]], [[bv,[49,49]]]]).

assignments(a,[]).
assignments(b,[]).
assignments(fsel,[]).
assignments(f,[s2,s3,s4,s5]).

uses(a, [[s5,[45],[76]], [s4,[45],[76]], [s3,[45],[76]],
[s2,[45],[]]]).

uses(b, [[s5,[45],[82]], [s4,[45],[82]]]).

uses(fsel, [[s1,[45],[]]]).

uses(f, []).

parentstatement(s2,s1).
parentstatement(s3,s1).
parentstatement(s4,s1).
parentstatement(s5,s1).
parentstatement(s1,[]).

subordinatelist(s1,[[bv,[49,49]]],[s5]).
subordinatelist(s1,[[bv,[49,48]]],[s4]).
subordinatelist(s1,[[bv,[48,49]]],[s3]).
subordinatelist(s1,[[bv,[48,48]]],[s2]).

made_vies(s1, [s1],[]) :- !.

made_vies(s2, [s1,s2], [[[bv,[48,48]],[obj,fsel],s1]]) :- !.

made_vies(s3, [s1,s3], [[[bv,[48,49]],[obj,fsel],s1]]) :- !.

made_vies(s4, [s1,s4], [[[bv,[49,48]],[obj,fsel],s1]]) :- !.

made_vies(s5, [s1,s5], [[[bv,[49,49]],[obj,fsel],s1]]) :- !.

evaluated_pos(s3,[76],[],bvnot,[76],[obj,a],[]):-!.

evaluated_pos(s4,[76],[],bvadd,[76],[obj,a],[obj,b]):-!.

evaluated_pos(s4,[82],[],bvadd,[82],[obj,b],[obj,a]):-!.

evaluated_pos(s5,[76],[],bvand,[76],[obj,a],[obj,b]):-!.

evaluated_pos(s5,[82],[],bvand,[82],[obj,b],[obj,a]):-!.

evaluated_pos(_17537,[],[],[],[],[],[]):-!.
```

Figure 5. .ife file of ALU

Then the fault list is generated. The .ife and .hdl file are given as an input to the Fault List Extractor (FLE). The FLE is implemented in Prolog. The FLE generates the fault list based on the behavioral fault model previously explained. The Fault list is given in an .fle file. Figure 6 shows the fault list for the ALU.

```
"ALU".
[1,deadclause,s1,[[bv,[48,48]]]].
[2,deadclause,s1,[[bv,[48,49]]]].
[3,deadclause,s1,[[bv,[49,48]]]].
[4,deadclause,s1,[[bv,[49,49]]]].
[5,assncntl,s2].
[6,assncntl,s3].
[7,assncntl,s4].
[8,microop,[s4,[45],[]],bvadd,bvsub].
[9,microop,[s4,[45],[]],bvadd,bvxor].
[10,assncntl,s5].
[11,microop,[s5,[45],[]],bvand,bvor].
[12,stuckdata,[s1,[45],[]],[bv,[48,48]]].
[13,stuckdata,[s1,[45],[]],[bv,[49,49]]].
[14,stuckdata,[s2,[45],[]],[bv,[48,48,48,48]]].
[15,stuckdata,[s2,[45],[]],[bv,[49,49,49,49]]].
[16,stuckdata,[s3,[45],[]],[bv,[48,48,48,48]]].
[17,stuckdata,[s3,[45],[]],[bv,[49,49,49,49]]].
[18,stuckdata,[s3,[45],[76]],[bv,[48,48,48,48]]].
[19,stuckdata,[s3,[45],[76]],[bv,[49,49,49,49]]].
[20,stuckdata,[s4,[45],[]],[bv,[48,48,48,48]]].
[21,stuckdata,[s4,[45],[]],[bv,[49,49,49,49]]].
[22,stuckdata,[s4,[45],[76]],[bv,[48,48,48,48]]].
[23,stuckdata,[s4,[45],[76]],[bv,[49,49,49,49]]].
[24,stuckdata,[s4,[45],[82]],[bv,[48,48,48,48]]].
[25,stuckdata,[s4,[45],[82]],[bv,[49,49,49,49]]].
[26,stuckdata,[s5,[45],[]],[bv,[48,48,48,48]]].
[27,stuckdata,[s5,[45],[]],[bv,[49,49,49,49]]].
[28,stuckdata,[s5,[45],[76]],[bv,[48,48,48,48]]].
[29,stuckdata,[s5,[45],[76]],[bv,[49,49,49,49]]].
[30,stuckdata,[s5,[45],[82]],[bv,[48,48,48,48]]].
[31,stuckdata,[s5,[45],[82]],[bv,[49,49,49,49]]].
?- end.
```

Figure 6. Fault List of ALU

3.4.2 Test Generator

After the preprocessing stage is over, the three files .hdl, .ife, .fle, are given as an input to the Test Generator as seen in Figure 2. The Test Generator generates the test vectors for a particular fault. In the test generator the faults are sensitized in the VHDL description and propagated through the VHDL statements. No fault simulation is used for test generation. The test generation process is guided by heuristic rules and is organized in the form of a goal tree as explained previously. For example the test vector generated for the 2nd fault in the fault list, a case-deadclause fault (statement s1) of the ALU is as shown in Figure 7. Similarly, test vectors can be obtained for the complete fault list.

ALU
[2,deadclause,s1,[[bv,[48,49]]]]

t\obj	a	b	fsel	f
t0+0	1111	x	00	x
t0+1	1111	x	01	0000/1111

Figure 7. Test Vectors for Fault 2

At time t0 (t0+0) of the test, an input of "1111" on *a*, "00" on *fsel*, propagates a "1111" to the output *f*. The fault is then sensitized. If the fault is present then statement s3 should not execute even though the input is "01" on *fsel* at time t1 (t0+1). If the fault is not present, "0000" should be obtained at the output. If the fault is present, a different value, "1111", is present at the output. Since in this case *f* is the observable output, no further propagation is required.

Chapter 4. Heuristic Test Generator

Originally the Behavioral Test Generator generated a single test vector, regardless of the microoperation fault. The gate level implementation of microoperations such as ADD and SUBTRACT, results in a large number of gates. Such microoperations are called "big" microoperations [32]. A single test vector will not provide adequate coverage in these circuits, e.g., a single test vector will not provide adequate coverage for a four bit adder. Hence some heuristic test generation rules are needed to generate test vectors for such microoperations. These heuristic test generation rules were proposed by C. Cho and J.R. Armstrong [32]. Using these heuristic test generation rules multiple vectors are generated for a single big microoperation fault. These multiple test vectors generated for the big microoperation can increase the fault coverage substantially. Thus a Heuristic Test Generator (HTG) was implemented which maintains the database that stores the heuristic rules for test generation for big microoperations. Figure 8 shows the interface of the HTG to BTG. BTG, when it needs to generate test vectors for a big microoperation, obtains the heuristic test generation rules from the HTG. The HTG provides the BTG with a set of test vectors.

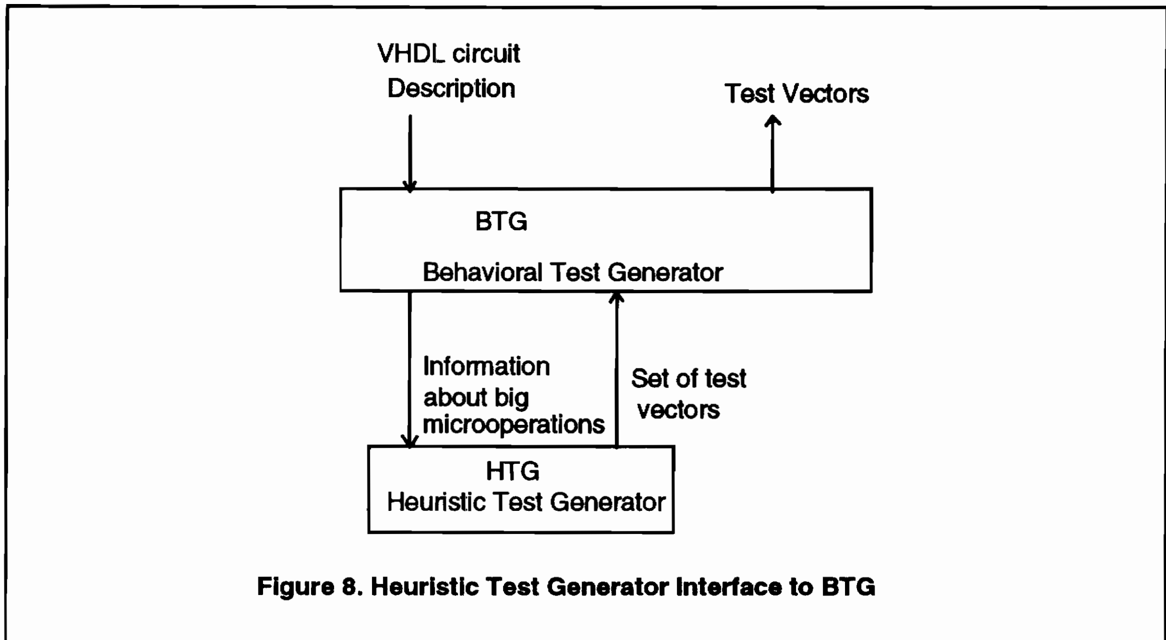


Figure 8. Heuristic Test Generator Interface to BTG

4.1 Implementation of Heuristic Test Generator

The Heuristic Test Generator (HTG) generates multiple vectors for the *bvadd* (bit_vector add) and *bvsub* (bit_vector subtract) microoperations. The HTG is implemented in Prolog. The microoperation fault model for bvadd and bvsub are obtained by perturbing the microoperations as follows:-

good: Z <= bvadd(A(n:0), B(n:0)), faulty Z <= bvsub(A(n:0),B(n:0))

good: Z <= bvsub(A(n:0), B(n:0)), faulty Z <= bvadd(A(n:0),B(n:0))

The bvadd or bvsub function can be defined in the following two ways:-

(i) bvadd(A,B) bvsub(A,B)

(ii) bvadd(A, "constant vector") bvsub(A, "constant vector")

The heuristics used for test vector generation for the above two cases are different.

Heuristics for (i)

There are many ways in which the test vectors for A and B can be selected. The following selection of test vectors was proposed by Cho and Armstrong. It was shown that very high coverage can be obtained by using these heuristics [33]. The microoperation bvadd is being perturbed to be the faulty microoperation bvsub. To generate the test vectors, first the length of vectors A and B are determined. The vector B is selected by assigning a 1 in the n'th position of the n'th vector and the rest of the bits are assigned a value "X" [32]. Consider the example of a four-bit vector. Table 1 shows the test patterns obtained using this condition.

Table 1. Test Patterns

A	B
XXXX	XXX1
XXXX	XX1X
XXXX	X1XX
XXXX	1XXX

The logic values are to be assigned to the don't care bits such that every don't care bit experiences both 0-1 and 1-0 transitions [32]. This method provided the highest detection of the gate level faults in the fault coverage experiments [33]. The above don't care values are assigned '0' and '1' alternately to generate the test vectors [32]. First, all the don't cares are assigned '0', then all the don't cares are assigned '1'. Then the don't cares in A are assigned '0' and all don't cares in B are assigned

'1'. Finally the don't cares in A are assigned '1' and all don't cares in B are assigned '0'. For example, for "XXXX XXX1" the test vectors 1,2,3 and 4 are obtained as seen in Table 2. Similarly the vectors are obtained for all the above patterns. Table 2 shows the vectors generated, with the duplicated vectors eliminated.

Table 2. Test Vectors

Vector No.	A	B
1	0000	0001
2	1111	1111
3	0000	1111
4	1111	0001
5	0000	0010
6	1111	0010
7	0000	0100
8	1111	0100
9	0000	1000
10	1111	1000

The test vectors 9 and 10 produce the same good/bad value pair, i.e. bvadd(A,B) and bvsub(A,B) have the same value. For a fault to be detected at the output the good/bad values should be different. Hence these vectors are eliminated. The test pattern 2 produces the good/bad pair 1110/0000 for A=1111 and B= 1111. In the implementation of the BTG the microoperation *bvadd* is also perturbed to *bvxor.*

This is fault 3 as seen in the fault list in Figure 11. The good/bad value for this fault is 1110/0000 for A=1111 and B=1111. Hence the test is duplicated and thus it has been eliminated. The set of test vectors obtained is as shown in Table 3.

Table 3. Test Vectors used for Test Generation

A	B
0000	0001
0000	1111
1111	0001
0000	0010
1111	0010
0000	0100
1111	0100

Similarly the test vectors can be obtained for a vector of any length. The HTG determines the length of the vector and then generates the test vectors automatically, and eliminates certain test vectors based on the tests explained.

Heuristics for (ii)

In this case the test vectors have to be selected only for A. An approach similar to the previous case is used. Thus the test patterns for A are selected as follows:-

Test patterns for A

XXX1
XX1X
X1XX
1XXX

The logic values are assigned to the don't care bits such that every don't care bit experiences both 0-1 and 1-0 detections. The above don't cares are assigned values '0' and '1' alternatively to generate the test vectors. The duplicate test vectors are eliminated. Hence the test vectors obtained are as follows.

Test Vectors for A

0001
0010
0100
1000
1111

Consider the simple example of an adder implemented in VHDL as shown in Figure 9. Figure 10 gives the Prolog facts representation. Figure 11 shows the fault list. The test vector generated for the microoperation fault number 2 from the fault list, with the previous implementation is as shown in Figure 12. With the previous implementation of the BTG only a single test vector was generated. Figure 13 shows the multiple test vectors generated after the implementation of the HTG. A substantial increase in coverage was obtained for the chips after the implementation of the HTG. A comparison of the increase in fault coverage was carried out for the adder and the ALU model. The adder model is as shown in Figure 9. The ALU model is given in Figure 3, Chapter 3. Chapter 5 explains the process of obtaining this fault coverage.

```
--VHDL description of adder

entity ADDER is
port(
        a,b:in BIT_VECTOR(0 to 3);
        c:out BIT_VECTOR(0 to 3));
end ADDER;

architecture BEH of ADDER is
begin
process(a,b)

begin

c <=add(a,b)
end process;
end BEH;
```

Figure 9.VHDL Description of Adder

```
modelspec :-
assert(fileprefix("adder")),
assert(modelname("ADDER")),

assert(datatype(a,bv)),assert(bvlength(a,4)),assert(inputpin(a)),
assert(datatype(b,bv)),assert(bvlength(b,4)),assert(inputpin(b)),
assert(datatype(c,bv)),assert(bvlength(c,4)),assert(outputpin(c)),

assert(statementtype(s1,assignment)),
assert(destinationobject(s1,c)),
assert(sourceexpression(s1,[bvadd,[obj,a],[obj,b]])).
```

Figure 10. Prolog Facts Representation of Adder

```
"ADDER".
[1,assncntl,s1].
[2,microop,[s1,[45],[]],bvadd,bvsub].
[3,microop,[s1,[45],[]],bvadd,bvxor].
[4,stuckdata,[s1,[45],[]],[bv,[48,48,48,48]]].
[5,stuckdata,[s1,[45],[]],[bv,[49,49,49,49]]].
[6,stuckdata,[s1,[45],[76]],[bv,[48,48,48,48]]].
[7,stuckdata,[s1,[45],[76]],[bv,[49,49,49,49]]].
[8,stuckdata,[s1,[45],[82]],[bv,[48,48,48,48]]].
[9,stuckdata,[s1,[45],[82]],[bv,[49,49,49,49]]].
?- end.
```

Figure 11. Fault List

```
Simple adder, vectors
[2,microop,[s1,[45],[]],bvadd,bvsub]

t\obj a       b       c
t0+0  0001    0001    0010/0000
```

Figure 12. Test Vector Generated without HTG

```
Simple adder, vectors
[2,microop,[s1,[45],[]],bvadd,bvsub]

t\obj a       b       c
t0+0  0000    0001    0001/1111

Simple adder, vectors
[2,microop,[s1,[45],[]],bvadd,bvsub]

t\obj a       b       c
t0+0  0000    0010    0010/1110

Simple adder, vectors
[2,microop,[s1,[45],[]],bvadd,bvsub]

t\obj a       b       c
t0+0  0000    0100    0100/1100

Simple adder, vectors
[2,microop,[s1,[45],[]],bvadd,bvsub]

t\obj a       b       c
t0+0  0000    1111    1111/0001

Simple adder, vectors
[2,microop,[s1,[45],[]],bvadd,bvsub]

t\obj a       b       c
t0+0  1111    0001    0000/1110

Simple adder, vectors
[2,microop,[s1,[45],[]],bvadd,bvsub]

t\obj a       b       c
t0+0  1111    0010    0001/1101

Simple adder, vectors
[2,microop,[s1,[45],[]],bvadd,bvsub]

t\obj a       b       c
t0+0  1111    0100    0011/1011
```

Figure 13. Test Vectors Generated after Implementation of HTG

4.2 Effect of Using the Heuristic Test Generator

	Gate Level Coverage	
	Before Implementation	After Implementation
Adder	60.0%	96.5%
ALU	73.8%	95.4 %

Figure 14. Increase in Fault Coverage

As seen from the above results a substantial increase in coverage was obtained after the implementation of the Heuristic Test Generator. The reason for the increase in coverage was that a single test vector does not cover the data paths well, as a sufficient variety of data vectors are not generated. By optimizing the heuristic method of selection of test vectors a substantial increase in coverage was achieved. For example consider the example of the Adder. Figure 9 shows the VHDL description of the adder. Figure 15 shows the gate level implementation of the adder. As seen from the figure, the stuck-at-0 faults for the nets marked with "**X**" and "**X***" were not detected before the implementation of the HTG. After the implementation of the HTG only the stuck-at-0 faults marked with "**X***" were not detected.

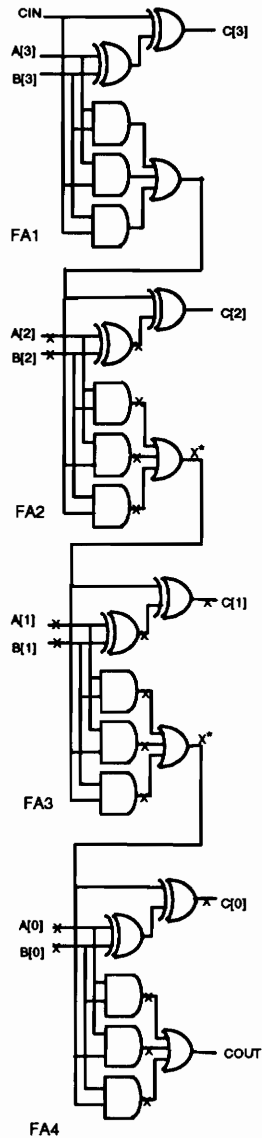


Figure 15. Gate Level Implementation of Adder

The heuristics for the adder are derived such that they provide the coverage for the adder. Figure 16 shows the cumulative gate level fault coverage for the test vectors of the heuristic test generator. As seen from the figure, as the number of test vectors increases the fault coverage increases.

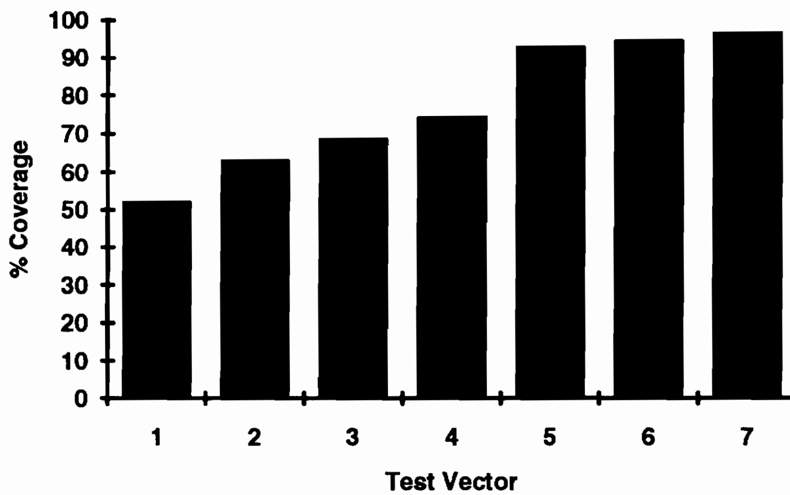


Figure 16. Cumulative % Coverage for Test Vector

It will be shown further that the Heuristic Test Generator combined with the Behavioral Test Generator can be used to achieve a very high coverage for a variety of models.

5. Gate Level Fault Coverage Determination

5.1 Criteria for Evaluation of Test vectors

We need to establish that the tests generated by the Behavioral Test Generator provide adequate coverage for the gate level faults. The test vectors obtained by the Behavioral Test Generator are to be evaluated by determining the fault coverage of the equivalent gate level circuit. The *fault coverage* is defined as the ratio of the number of Stuck-at-1 and Stuck-at-0 faults detected in the equivalent gate level circuit by the test vectors to the total number of these faults present in the equivalent gate level circuit.

5.2 System for Gate Level Fault Coverage Determination

The circuit under test is represented in its behavioral VHDL description. This high level VHDL model is the input to the BTG. The BTG generates the test vectors based on the behavioral fault model as explained previously. These test vectors obtained from the BTG are in the standard BTG format. These test vectors are applied to the equivalent gate model, and the coverage of the waveforms determined. Figure 17 shows the complete system that was developed for determination of the fault coverage.

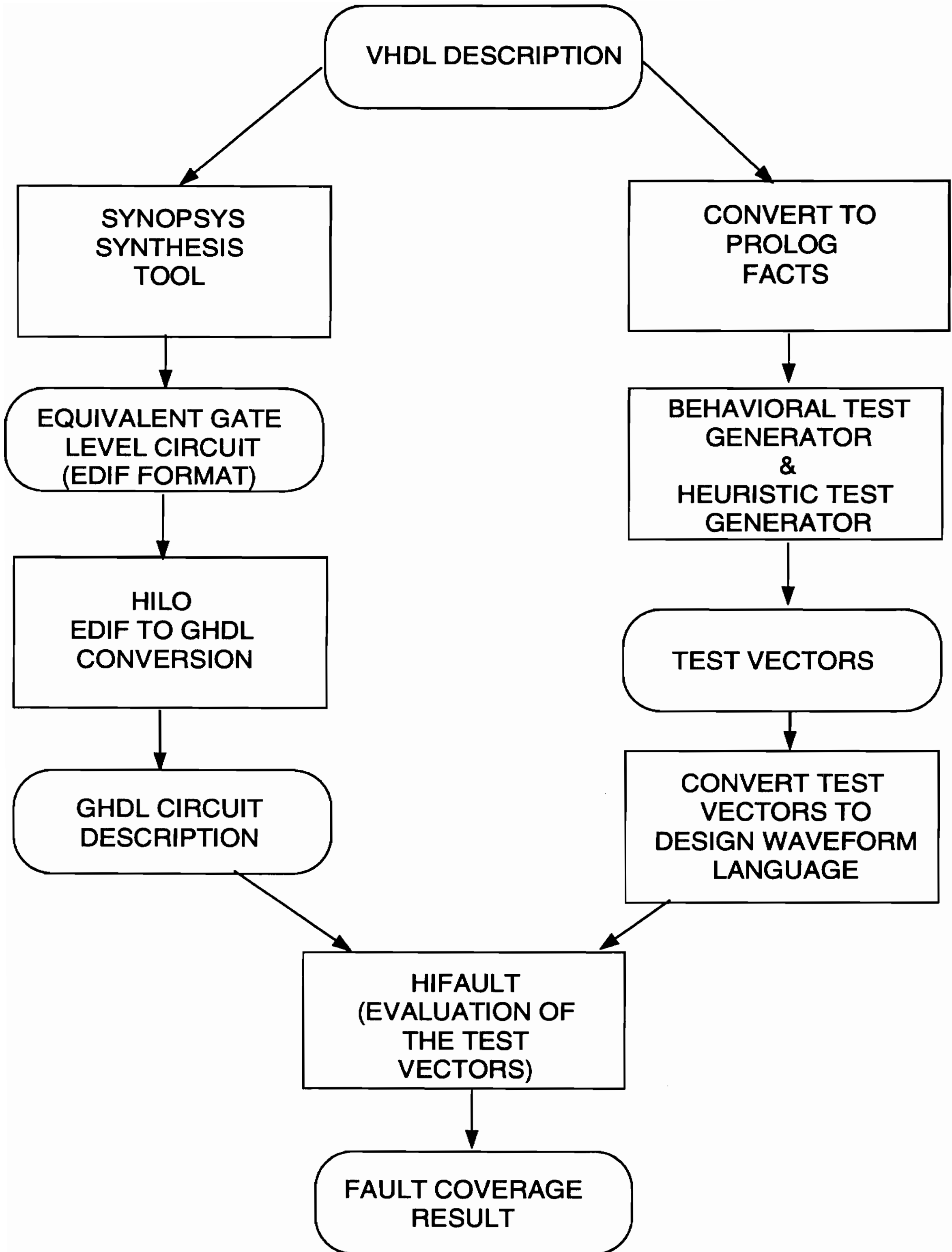


Figure 17. Test Setup for Determining Fault Coverage

5.2.1 Fault Simulator: HIFAULT

For determining the coverage any fault simulator can be used. In our case, the System HILO fault simulator HIFAULT [25] is used for determining the fault coverage. The fault simulator evaluates the waveform test patterns by applying them to a circuit model containing faults. The fault dictionary for the input circuit is generated automatically by HIFAULT or is given as an input by the user. HIFAULT injects all the faults into the circuit. The Waveform is said to *detect* a fault when a circuits primary output value differs from the fault-free value. The actual criteria used for fault detection can also be specified by the user. Once a fault is detected it is dropped, After the fault simulation of the waveform test patterns is over, HIFAULT gives the fault coverage result. Figure 18 shows the overview of HIFAULT.

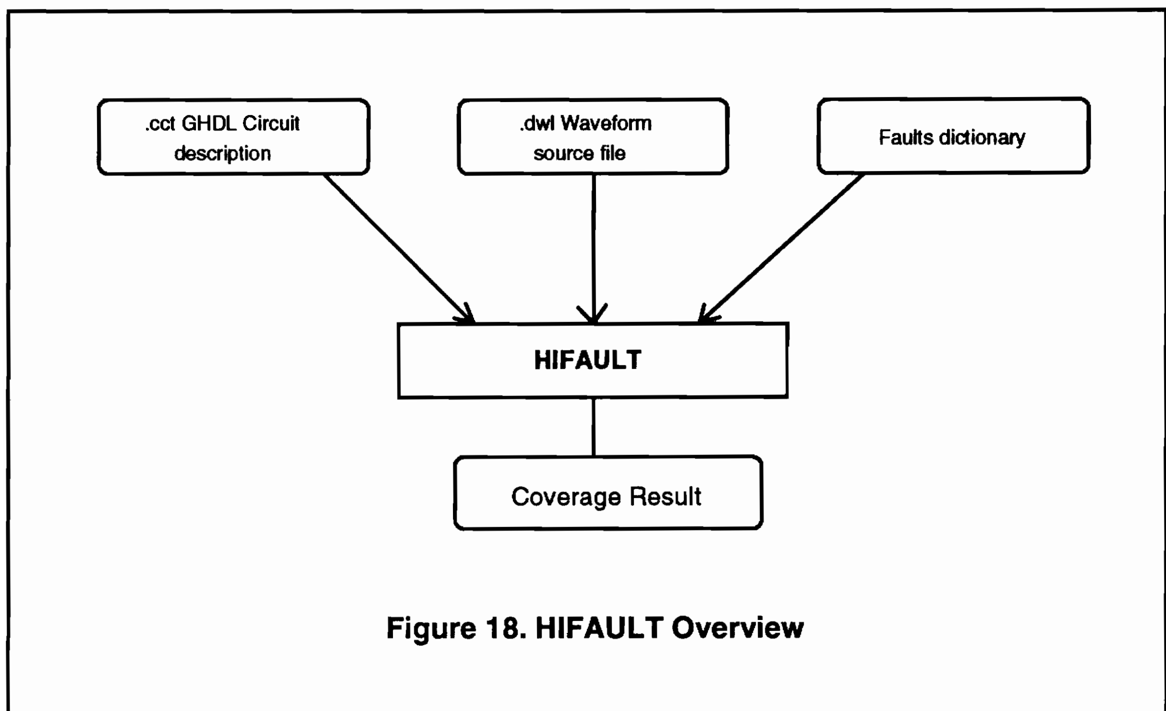


Figure 18. HIFAULT Overview

As seen from Figure 18 HIFAULT requires a GHDL (Genrad Hardware Description Language) description of the gate level circuit. A gate level circuit is developed equivalent to the high level VHDL circuit description. This gate level circuit is then represented in GHDL. The details of this Hardware Description Language can be obtained from the GHDL Reference Manual [26]. For example Figure 19 shows the VHDL description of an ALU.

```
--VHDL description of ALU

library SYNOPSYS;
use synopsys.bv_arithmetic.all;

entity ALU is
port(A,B: in BIT_VECTOR(3 downto 0);
    FSEL:in BIT_VECTOR(1 downto 0);
    F:out BIT_VECTOR(3 downto 0));
end ALU;

architecture BEH of ALU is

begin

process(A,B,FSEL)
  begin

   case FSEL is

   when "00" => F <=A;
   when "01" => F <=not(A);
   when "10" => F <=A + B;
   when "11" => F <=A and B;

   end case;
end process;

end BEH;
```

Figure 19. VHDL Description of ALU

Figure 20 shows the implementation of the ALU. This ALU circuit contains the register level primitives, adder and Quad 4_to_1 MUX. These primitives were first modeled at the gate level. Figure 21 gives GHDL descriptions of these models and the GHDL description of a structural model of the ALU that uses these primitives. This method of generation of equivalent gate level circuits is a very time consuming process. However this task has now been automated using the Synopsys synthesis tool. The Synopsys synthesis tool is explained in Section 5.2.2.

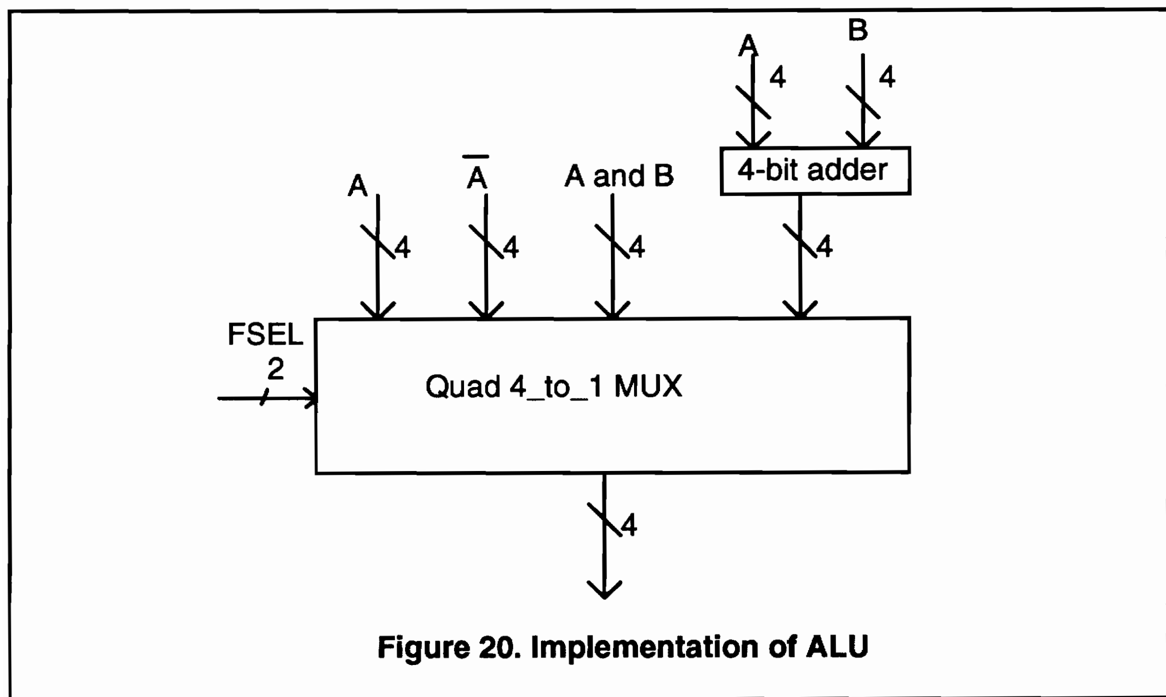


Figure 20. Implementation of ALU

```
CCT ALU (a[0:3],b[0:3],fsel[0:1],f[0:3])
WIRE w1,w2,w3,w4,w5,w6,w7,w8;

ADDER ADDER1  (a[0:3],b[0:3],t[0:3],CO);

and (1,1) g1 (w1,a[0],b[0]);
and (1,1) g2 (w2,a[1],b[1]);
and (1,1) g3 (w3,a[2],b[2]);
and (1,1) g4 (w4,a[3],b[3]);

not(1,1) g5 (w5,a[0]);
not(1,1) g6 (w6,a[1]);
not(1,1) g7 (w7,a[2]);
not(1,1) g8 (w8,a[3]);

MX MX1
          (a[0:3],w5,w6,w7,w8,t[0:3],w1,w2,w3,w4,fsel[0:1],f[0:3]);

ENDCIRCUIT

CCT ADDER (a[0:3],b[0:3],c[0:3],CO)
input a[0:3],b[0:3];
wire w1,w2,w3;

FA        FA1 (a[0],b[0],0,c[0],w1),
          FA2 (a[1],b[1],w1,c[1],w2),
          FA3 (a[2],b[2],w2,c[2],w3),
          FA4 (a[3],b[3],w3,c[3],CO);

ENDCIRCUIT

CCT FA (A,B,CIN,S,COUT)
input A,B,CIN;
wire w1,w2,w3,w4;
xor(1,1) g1 (w1,A,B);
xor(1,1) g2 (S,w1,CIN);
and(1,1) g3(w2,A,B);
and (1,1) g4(w3,A,CIN);
and(1,1) g5(w4,B,CIN);
or(1,1) g6(COUT,w2,w3,w4);

ENDCIRCUIT


CCT MX (in0[0:3],in1[0:3],in2[0:3],in3[0:3],sel[0:1],o[0:3])
input in0[0:3],in1[0:3],in2[0:3],in3[0:3],sel[0:1];

MUX       MUX1 (in0[0],in1[0],in2[0],in3[0],sel[0:1],o[0]),
          MUX2 (in0[1],in1[1],in2[1],in3[1],sel[0:1],o[1]),
          MUX3 (in0[2],in1[2],in2[2],in3[2],sel[0:1],o[2]),
          MUX4 (in0[3],in1[3],in2[3],in3[3],sel[0:1],o[3]);

ENDCIRCUIT

CCT MUX (in0,in1,in2,in3,sel[0:1],o)
input  in0,in1,in2,in3,sel[0:1];
wire w1,w2,w3,w4,w5,w6;
not(1,1) g1 (w1,sel[0]);
not(1,1) g2 (w2,sel[1]);
and (1,1) g3 (w3,w1,w2,in0);
and (1,1) g4 (w4,w1,sel[1],in1);
and (1,1) g5 (w5,sel[0],w2,in2);
and (1,1) g6 (w6,sel[0],sel[1],in3);
or(1,1) g7 (o,w3,w4,w5,w6);

ENDCIRCUIT
```

Figure 21. GHDL Description of ALU

As seen from Figure 18 the test vectors to be given to HIFAULT must be in the DWL (Design Waveform Language) [27] format. The test vectors obtained from the BTG are converted to DWL format using a BTG2DWL converter, which was implemented in C. The users guide of BTG2DWL is as given in Appendix A. For example consider the test vectors generated by BTG for an adder as shown in Figure 22. This is converted to the DWL format by BTG2DWL as shown in Figure 23. For a detailed description of DWL please refer to "System HILO DWL Reference Manual" by Genrad Corporation.

```
Adder
[2,microop,[s1,[45],[]],bvadd,bvsub]
t\obj a      b      c
t0+0  0000   0001   0001/1111

Adder
[2,microop,[s1,[45],[]],bvadd,bvsub]
t\obj a      b      c
t0+0  0000   0010   0010/1110

Adder
[2,microop,[s1,[45],[]],bvadd,bvsub]
t\obj a      b      c
t0+0  0000   0100   0100/1100

Adder
[2,microop,[s1,[45],[]],bvadd,bvsub]
t\obj a      b      c
t0+0  0000   1111   1111/0001

Adder
[2,microop,[s1,[45],[]],bvadd,bvsub]
t\obj a      b      c
t0+0  1111   0001   0000/1110

Adder
[2,microop,[s1,[45],[]],bvadd,bvsub]
t\obj a      b      c
t0+0  1111   0010   0001/1101

Adder
[2,microop,[s1,[45],[]],bvadd,bvsub]
t\obj a      b      c
t0+0  1111   0100   0011/1011
```

Figure 22. Test Vectors for an Adder Generated by BTG

```
WAVEFORM Adder;
BASE    BIN;
** INPUT PIN DECLARATIONS
INPUT
            a[0:3]:=BIN 0000
            b[0:3]:=BIN 0001;
** OUTPUT PIN DECLARATIONS
OUTPUT
            c[0:3]=BIN 0001;
BEGIN

AT  0
            a[0:3]:=BIN 0000
            b[0:3]:=BIN 0001
            c[0:3]=BIN 0001;
AT  75
            strobe  ();
AT  100
            a[0:3]:=BIN 0000
            b[0:3]:=BIN 0010
            c[0:3]=BIN 0010;
AT  175
            strobe  ();
AT  200
            a[0:3]:=BIN 0000
            b[0:3]:=BIN 0100
            c[0:3]=BIN 0100;
AT  275
            strobe  ();
AT  300
            a[0:3]:=BIN 0000
            b[0:3]:=BIN 1111
            c[0:3]=BIN 1111;
AT  375
            strobe  ();
AT  400
            a[0:3]:=BIN 1111
            b[0:3]:=BIN 0001
            c[0:3]=BIN 0000;
AT  475
            strobe  ();
AT  500
            a[0:3]:=BIN 1111
            b[0:3]:=BIN 0010
            c[0:3]=BIN 0001;
AT  575
            strobe  ();
AT  600
            a[0:3]:=BIN 1111
            b[0:3]:=BIN 0100
            c[0:3]=BIN 0011;
AT  675
            strobe  ();
      End
 Endwaveform Adder
```

Figure 23. DWL Representation of BTG Test Vectors

Once we have obtained the GHDL and DWL description of a circuit they are given as an input to HIFAULT. HIFAULT generates the fault coverage results. The fault coverage results generated are of the following format. For example Figure 24 gives the result summary for a 32 Function ALU.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	294	284	278	6	0	10	0
Stuck1	294	288	283	5	0	6	0
All	588	572	561	11	0	16	0
Percentages:		97.28	95.41	1.87	0.00	2.72	

Figure 24. Fault Summary after Fault Simulation

The status of the faults in Figure 24 is explained below:-

At each strobe (see Figure 23) in the waveform description the fault simulator makes a comparison between the fault free value at each primary output of the circuit, and the corresponding faulty values for all the faulty circuits. Faulty values which differ will be marked as ***hard detects (hd)*** or ***potential detects (pd)***. The BTG uses the three valued logic, '1', '0' and 'X'. Figure 25 defines *hard detects* and *potential detects*. After a fault has been hard detected, or potentially detected six times, the fault is ***dropped***, i.e. the fault simulator does not simulate the fault any longer.

Good Values	Faulty Values		
	X	1	0
X	-	-	-
1	pd	-	hd
0	pd	hd	-

hd - hard detects   pd - potential detects

Figure 25. Hard detects and Potential detects

The ***catastrophic faults*** are the faults that were simulated, but were automatically suppressed. The following faults are classified as catastrophic faults:-

- Faults which leave the circuit in an undefined state at initialization.
- Faults that cause the circuit to oscillate in zero delay.

The ***undetectable faults*** are those faults which cannot be propagated to the primary outputs.

The remaining faults are the not detected faults.

5.2.2 Synopsys Synthesis Tool

The generation of the equivalent gate level circuit and then representing it in GHDL is a very time consuming task. Using the Synopsys Synthesis tool, this time of generation of the GHDL has been greatly reduced. The Synopsys Synthesis tool reads in designs in a variety of formats and synthesizes a circuit in a given technology. Usually for the Synopsys synthesis tool the circuit is described in a

VHDL Behavioral Description. The circuits were optimized for CMOS technology. Figure 26 gives an overview of the Synopsys synthesis tool [28].

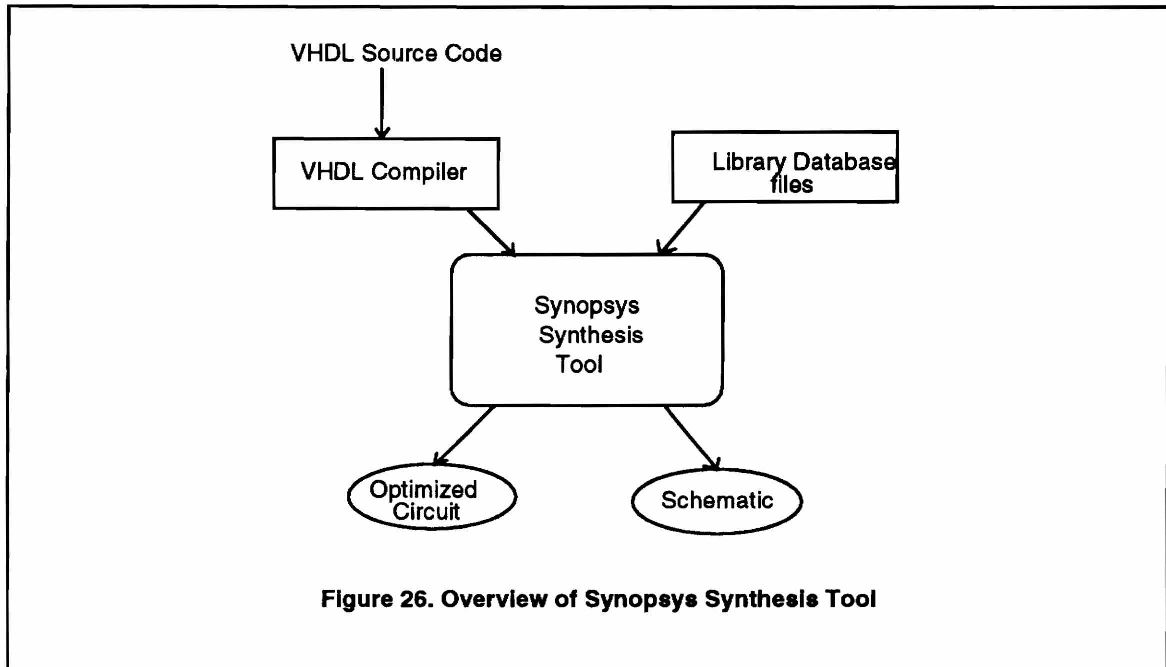


Figure 26. Overview of Synopsys Synthesis Tool

The VHDL description of the 32-Function ALU shown in Figure 27 was given as an input to the Synopsys synthesis tool. Figure 28 shows the corresponding gate level circuit. This gate level circuit description must be represented in the GHDL format. To facilitate the transfer of circuit information from Synopsys to HILO the gate level circuit is represented in the Electronic Design Interchange Format (EDIF) using the Synopsys synthesis tool [31]. The EDIF format consists of a *netlist,* "a hierarchical list of components and how they are interconnected" and a *schematic,* "a graphical representation of the netlist". For transfer of circuit description from Synopsys to

HILO only the *netlist* is used. For a detailed description of EDIF refer to the EDIF Reference Manual [31].

```
library SYNOPSYS;

--VHDL description of Function Generator
use synopsys.bv_arithmetic.all;

entity FUNGEN is
port(A,B,M: in BIT_VECTOR(0 to 3);
    FSEL:in BIT_VECTOR(0 to 3);
    F:out BIT_VECTOR(0 to 3));
end FUNGEN;

architecture BEH of FUNGEN is

begin

process(M,FSEL,A,B)

begin

if M ='0' then


case FSEL is
        when "0000" => F <= A - "0001" ;

        when "0001" => F <= (A and B) - "0001" ;

        when "0010" => F <= (A and (not B)) - "0001" ;

        when "0011" => F <= "0000" - "0001" ;

        when "0100" => F <= A + (A or (not B)) ;

        when "0101" => F <= (A and B) + (A or (not B)) ;

        when "0110" => F <= A - B -"0001" ;

        when "0111" => F <= (A or (not B)) ;

        when "1000" => F <= A + (A or B) ;

        when "1001" => F <= A + B ;

        when "1010" => F <= (A and (not B)) + (A or B) ;

        when "1011" => F <= (A or B)  ;
```

Figure 27. VHDL Description of 32 Function ALU

```
        when "1100" => F <= A + A  ;

        when "1101" => F <= (A and B) + A  ;

        when "1110" => F <= (A and (not B)) + A  ;

        when "1111" => F <= A ;

end case;

        end if;

if M ='1' then

case FSEL is
        when "0000" => F <= not A;

        when "0001" => F <= not (A and B);

        when "0010" => F <= (not A) or B;

        when "0011" => F <="0001";

        when "0100" => F <= not (A or B);

        when "0101" => F <= not B;

        when "0110" => F <=not ( A xor B);

        when "0111" => F <= A or (not B);

        when "1000" => F <= (not A) and B;

        when "1001" => F <=  A xor B;

        when "1010" => F <= B;

        when "1011" => F <= (A or B);

        when "1100" => F <= "0000";

        when "1101" => F <= (A and (not B));

        when "1110" => F <= A and  B;

        when "1111" => F <= A;

end case;
end if;


end process;
end BEH;
```

Figure 27. VHDL Description of 32 Function ALU (continued)

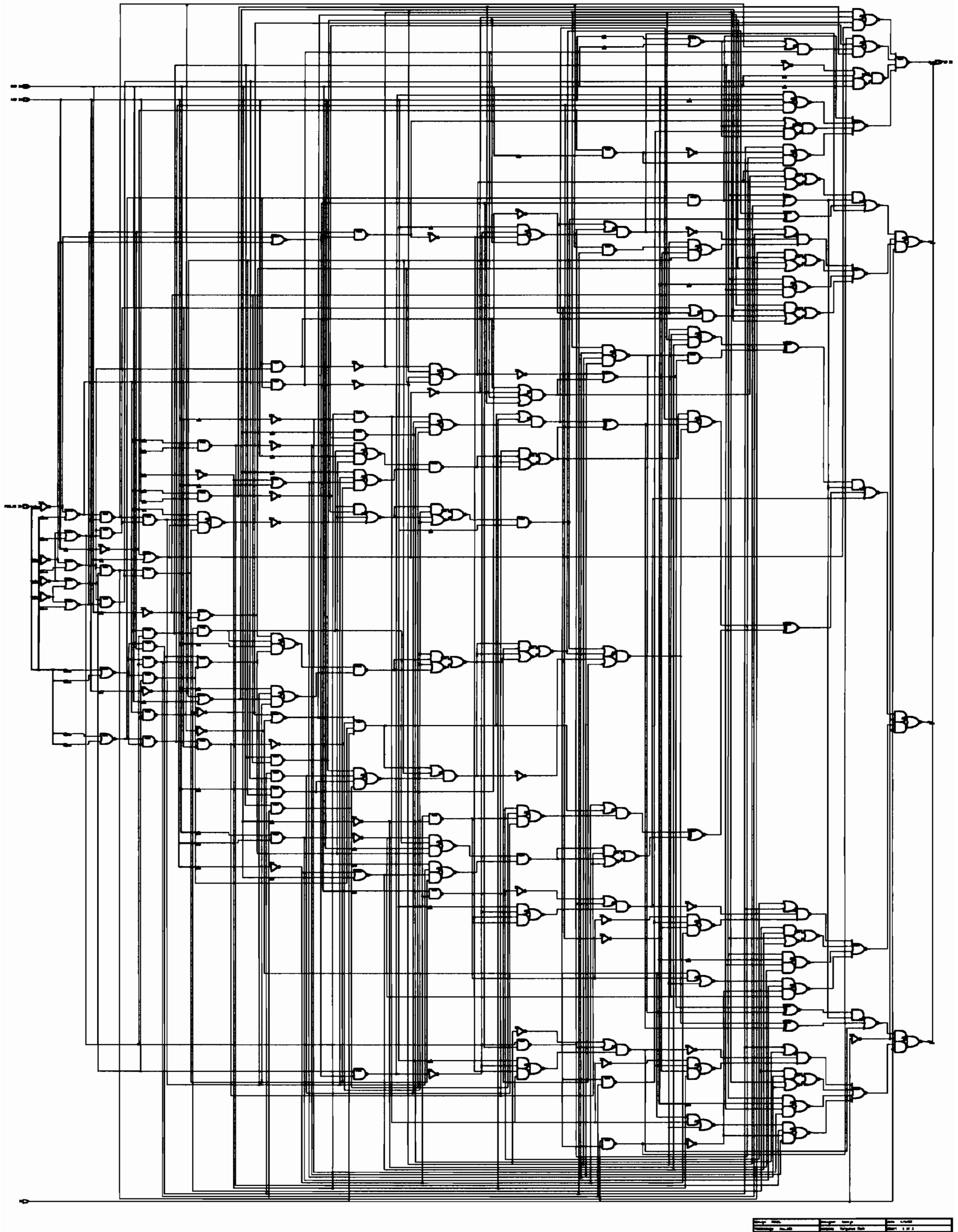


Figure 28. Gate Level Schematic of 32 Function ALU

5.3 EDIF to GHDL Conversion

The Synopsys synthesis tool synthesizes the circuit in the specified primitive library components. The lsi_10k primitive library was used in synthesizing the circuits analyzed in this thesis. The EDIF netlist representation of the synthesized gate level circuit is in terms of this library. For the conversion of EDIF to GHDL the corresponding HILO_LSI10k libraries should be available. At present these libraries are not available. Thus the translation of the EDIF netlist to the required GHDL representation is achieved as follows:-

The EDIF is converted to GHDL using the HILO EDIF-->GHDL converter. The GHDL obtained has the correct syntax but is in terms of lsi_10k primitive components. These primitive components consist of primitive logic gates and other non-standard cells. For the non-standard lsi_10k primitive cells the corresponding primitive GHDL cells were created. The listing of these cells is given in Appendix B under "cells.cct". These GHDL cells map directly to the lsi_10k cells.

The representation of the lsi_10k gates and GHDL gates is different. To obtain the required GHDL description the lsi_10k logic gate representation is mapped to the corresponding GHDL primitive gate representation.

e.g., in lsi_10k library a 2-input AND gate is represented as follows:-

```
AN2 g1 (A= s1, B=s2, Z=w1);
```

In the above representation A,B are the inputs, and Z is the output. In this case keyword mapping is used.

To obtain the correct format of GHDL the above lsi_10k primitive gate is to be mapped to the corresponding GHDL primitive gate, e.g., the GHDL representation for the above AND gate is as follows :-

AND g1 (w1, s1 ,s2).

GHDL primitive gates use positional mapping in the structural description. The formal terminal list for the AND gate is defined as :-

AND g1 (OUTPUT,INPUT[1:n]).

Similarly the mapping was obtained for all the lsi_10k primitive gates. This mapping was carried out manually in the GHDL representation obtained using the HILO EDIF --> HILO converter. After the mapping the required GHDL gate level representation was obtained.

Chapter 6. Results

6.1 Selection of Models

Various register level primitives have been suggested [30] that can be used for modeling large circuits. The coverage results were first determined for these register level primitives. Then the coverage results were determined for large circuits modeled using these primitives. In most cases, this evaluation of stuck-at-fault coverage is only carried out for one gate level equivalent circuit. However, it was shown by Chao and Gray [29] that for combinational circuits the fault coverage did not vary significantly with change in implementation at the gate level.

The description and fault coverage of the models are presented in this chapter. The VHDL behavioral description, the Prolog representation, and the equivalent gate level GHDL description of the models are given in Appendix B.

6.2 Register Level Primitives

6.2.1 Combinational Logic Primitives

CKA

Figure 29 shows the circuit description of CKA. This circuit is made up of two flip-flops. The output of the first flip-flop is given as an input to the second flip-flop. The outputs of the first and second flip-flop are "anded" to obtain the output. The two flip-flops are given separate clock inputs CLK1 and CLK2. The flip-flops were implemented at the gate level using nand gates. The faults are defined in terms of stuck-at-1 and stuck-at-0 faults at the gate level. The gate level implementation of CKA has 13 gates. Figure 30 shows the coverage results obtained. The interpretation of the coverage results was explained in Chapter 5.

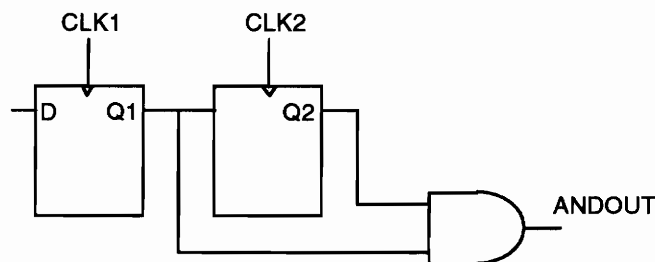


Figure 29. Circuit Diagram of CKA

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	16	16	14	2	0	0	0
Stuck1	16	16	14	2	0	0	0
All	32	32	28	4	0	0	0
Percentages:		100.00	87.50	12.50	0.00	0.00	

Percentages are of DETECTABLE faults.

Figure 30. Fault Coverage Results of CKA

ADDER

The model is of a four-bit unsigned adder. The circuit model consists of the vector addition of A and B and output C. The equivalent gate level implementation is in xor gates. The gate level implementation is as shown in Chapter 4 Figure 15. The gate level implementation of the adder has 24 gates. As seen from the figure the simple higher level model results in a big model as the gate level. As explained previously before the HTG was implemented the coverage was very low. The Heuristic Test Generator was implemented and a substantial increase in coverage was obtained. The coverage results in Figure 31 are after the implementation of the HTG.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	26	24	24	0	0	2	0
Stuck1	28	28	28	0	0	0	0
All	54	52	52	0	0	2	0
Percentages:		96.30	96.30	0.00	0.00	3.70	

Percentages are of DETECTABLE faults

Figure 31. Fault Coverage Results of Adder

Gate Level Implementation of Adder

As explained above before the implementation of the HTG, the coverage obtained was very low for the adder. To determine a set of test vectors that would provide adequate coverage for the adder, the test vectors were generated using BTG from the gate level VHDL model of the adder. A total of 124 test vectors were generated. The coverage results obtained are as shown in Figure 32. As seen from Figure 32 a complete coverage is obtained. This coverage was higher than that obtained from the implementation of the HTG (see Figure 31), as individual bits and not bit-vectors were used for propagating the fault to the output. Using the heuristic bit vectors it was not possible to propagate the undetected faults to the output. The HTG used only 7 test vectors to achieve the coverage. This method of generating tests from the gate level description could be used in "Synthesis Directed Test Generation" which is explained in the next chapter.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	28	28	28	0	0	0	0
Stuck1	28	28	28	0	0	0	0
All	56	56	56	0	0	0	0
Percentages:		100.00	100.00	0.00	0.00	0.00	

Percentages are of DETECTABLE faults.

Figure 32. Fault Coverage Results of Gate Level Implementation of Adder

LOOK AHEAD CARRY ADDER

The previous implementation of the adder was using XOR gates. To establish that the coverage results do not depend on the implementation, the adder was implemented at the gate level as a look ahead carry adder. The gate level implementation of the look ahead carry adder has 20 gates. Figure 33 shows the coverage results. As seen from the figure a very high coverage was obtained. Thus in this case we can conclude that the coverage does not depend on the gate level implementation.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	23	23	23	0	0	0	0
Stuck1	23	23	23	0	0	0	0
All	46	46	46	0	0	0	0
Percentages:		100.00	100.00	0.00	0.00	0.00	

Percentages are of DETECTABLE faults.

Figure 33. Fault Coverage Results of Look Ahead Carry Adder

SUBTRACTER

The circuit model is of a four-bit unsigned subtracter. Similar to the adder, the high level VHDL implementation of the subtracter is very simple. The circuit was implemented as the subtraction of vector A and B and output vector C. The implementation of the subtracter at the gate level results in a large number of gates. The gate level implementation of the subtracter has 28 gates. Figure 34 shows the coverage results for the subtracter after the implementation of the Heuristic Test Generator.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	32	29	29	0	0	3	0
Stuck1	32	32	32	0	0	0	0
All	64	61	61	0	0	3	0
Percentages:		95.31	95.31	0.00	0.00	4.69	

Percentages are of DETECTABLE faults.

Figure 34. Fault Coverage Results of Subtracter

4_TO_1 MUTIPLEXER

The circuit model is a four-input mux. The inputs are single bits. The inputs are *IN0,IN1,IN2,IN3*. The output is selected by the *SEL* signal. The output of the MUX is *O*. The gate level implementation of the 4_to_1 multiplexer has 7 gates. Figure 35 shows the fault coverage results.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	13	13	13	0	0	0	0
Stuck1	13	11	11	0	0	2	0
All	26	24	24	0	0	2	0
Percentages:		92.31	92.31	0.00	0.00	7.69	

Percentages are of DETECTABLE faults.

Figure 35. Fault Coverage Results of 4_TO_1 Multiplexer

As seen from Figure 35 and Figure 36, the percentage fault coverage of the 4_to_1 multiplexer is lower than that for the quad 4_to_1 multiplexer. The reason why the two stuck-at-1 faults in the 4_to_1 multiplexer were not detected, was due to the selection of the good/bad value pair by the BTG. For the generation of test vectors for the 4_to_1 multiplexer the good/bad value pair was "1/0" and for quad 4_to_1 multiplexer the good/bad value pair was "0000/1111". Thus the coverage depends on the selection of good/bad value pairs. The coverage results can be improved by optimizing the method of selection of test vectors.

Quad 4_TO_1 MULTIPLEXER

The circuit model of the Quad 4_to_1_MUX is a four-input multiplexer. The inputs are 4 bits wide. The inputs are *IN0,IN1,IN2,IN3*. The output is selected by the SEL lines. The output of the MUX is *O*. The gate level implementation of the quad 4_to_1 multiplexer has 28 gates. Figure 36 shows the fault coverage results.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	46	46	46	0	0	0	0
Stuck1	46	46	45	1	0	0	0
All	92	92	91	1	0	0	0
Percentages:		100.00	98.91	1.09	0.00	0.00	

Percentages are of DETECTABLE faults.

Figure 36. Fault Coverage Results of Quad 4_to_1 Multiplexer

DECODER

The circuit model is of a two to four line decoder. The input is a two-bit vector *I* and the output *O* is four bits wide. The gate level implementation of the decoder has 6 gates. Figure 37 shows the coverage results obtained.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	8	8	8	0	0	0	0
Stuck1	8	8	8	0	0	0	0
All	16	16	16	0	0	0	0
Percentages:		100.00	100.00	0.00	0.00	0.00	

Figure 37. Fault Coverage Results of Decoder

6.2.1 Sequential Primitives

JK FLIP - FLOP

The circuit model of the JK flip-flop has the truth table as shown below. The S and R signals in the VHDL description are the Set and Reset signals of the flip-flop. The flip-flop was implemented using gates. The gate level implementation of the flip-flop has 16 gates. Figure 38 shows the coverage results obtained.

Table 4: Truth Table of JK Flip-Flop

J	K	Q_{n+}	$\overline{Q}_{n+}$
0	0	Q_n	$\overline{Q}_n$
0	1	0	1
1	0	1	0
1	1	$\overline{Q}_n$	Q_n

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	20	20	20	0	0	0	0
Stuck1	20	20	20	0	0	0	0
All	40	40	40	0	0	0	0
Percentages:		100.00	100.00	0.00	0.00	0.00	

Percentages are of DETECTABLE faults

Figure 38. Fault Coverage Results of JK Flip-Flop

REGISTER

The register circuit consists of four inputs and one output. The STRB input is used to latch the input vector into an internal register. The output is enabled by two other input signals DSI and NDS2. At the gate level the register is modeled using flip-flops. The flip-flops were implemented at the gate level. The gate level implementation of the register has 58 gates. Figure 39 shows the coverage results.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	77	77	76	1	0	0	0
Stuck1	77	77	76	1	0	0	0
All	154	154	152	2	0	0	0
Percentages:		100.00	98.70	1.30	0.00	0.00	

Percentages are of DETECTABLE faults.

Figure 39. Fault Coverage Results of Register

COUNTER

The circuit is of a two-bit counter with an asynchronous clear. The inputs are *clk* and *clear*. The counter is incremented at each clock. The clear signal resets the counter. The counter is implemented at the gate level using positive edge triggered flip-flops. The flip-flops were implemented at the gate level using nand gates. The gate level implementation of the counter has 16 gates. Figure 40 shows the coverage results.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	18	17	16	1	0	1	0
Stuck1	18	17	16	1	0	1	0
All	36	34	32	2	0	2	0
Percentages:		94.44	88.89	5.56	0.00	5.56	

Percentages are of DETECTABLE faults.

Figure 40. Fault Coverage Results of Counter

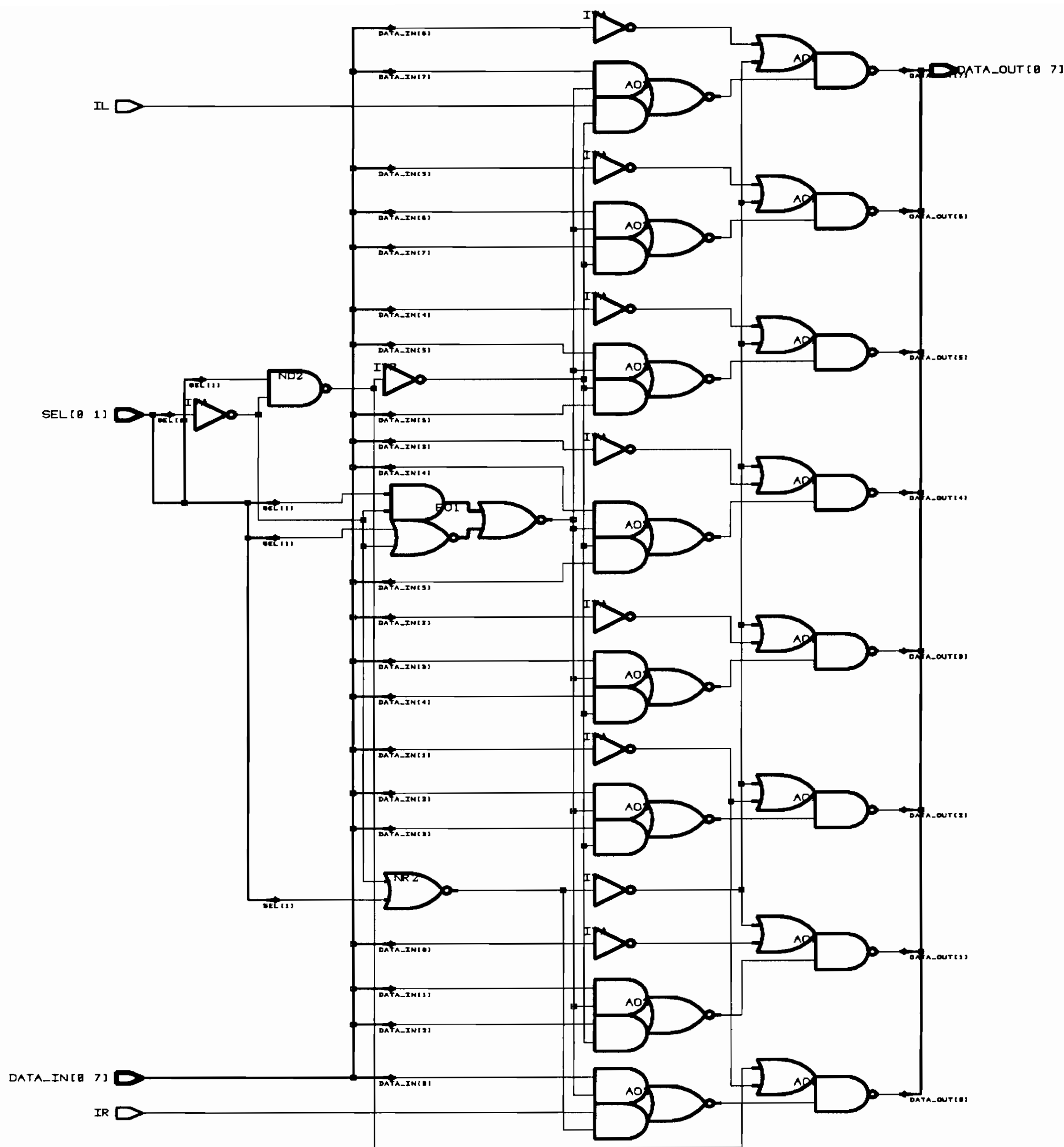
BAREL SHIFTER

The shifter is used for data operations. The input signals to the shift register are:- *DATA_IN*: the input data; *SEL*: selects the function to be performed; *IR* and *IL*: the serial inputs for right and left shift respectively. The shifter is loaded when *SEL* = "00" or "11". The contents are shifted left when *SEL* = "01" and shifted right when *SEL* = "10". Figure 42 shows an equivalent gate level implementation of the shifter that was obtained using the Synopsys synthesis tool. The gate level implementation of the barel shifter has 55 gates. Figure 41 shows the coverage results obtained.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	67	67	67	0	0	0	0
Stuck1	67	67	67	0	0	0	0
All	134	134	134	0	0	0	0
Percentages:		100.00	100.00	0.00	0.00	0.00	

Figure 41. Fault Coverage Results of Barel Shifter



design SHIFTER	designer baweja	date 12/26/92
technology lsi_10k	company Virginia Tech	sheet 1 of 1

Figure 42. Gate Level Schematic of Barel Shifter

Summary of Coverage Results of Register Level Primitives

An average of 98.11% fault coverage was obtained for the register level primitives. The coverage results for the primitives are summarized in Table 5.

Table 5. Fault Coverage Results of Register Level Primitives

Combinational Logic Primitives

Name	Description	Coverage
CKA	Implementation of D flip-flop	100.0
Adder	xor implementation	96.3
Adder	Look Ahead Carry Adder	100.0
Subtracter	High level subtracter	95.3
MUX1	4 to 1 Multiplexer	92.3
MUX4	Quad 4 to 1 Multiplexer	100.0
Decoder	2 to 4 decoder	100.0

Sequential Logic Primitives

Name	Description	Coverage
JKFF	JK-Flip Flop	100.0
REG	8 bit Register	100.0
Counter	Simple Counter	95.4
Barel Shifter	Shift left and Right	100.0

6.3 Big Models

These models can be constructed from the circuit primitives, discussed in the previous sections.

FOUR FUNCTION ALU

The ALU is used to perform the basic arithmetic and logic operations in computer systems. The circuit model of the ALU has data inputs *A* and *B,* bit vectors of length 4. The output is F, bit vector of length 4. The function is selected by *FSEL,* bit vector of length 2. The functions implemented by ALU as follows.

1. F = A
2. F = not(A)
3. F = A + B
4. F = A and B

The gate level implementation of the ALU has 60 gates. A high fault coverage for the ALU was obtained as shown in Figure 43.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	64	58	58	0	0	6	0
Stuck1	66	66	66	0	0	0	0
All	130	124	124	0	0	6	0
Percentages:		95.38	95.38	0.00	0.00	4.62	

Percentages are of DETECTABLE faults.

Figure 43. Fault Coverage Results of Four Function ALU

LINEAR FEEDBACK SHIFT REGISTER

The four stage linear feedback shift register (LFSR) is implemented. The LFSR is implemented using edge-triggered flip-flops and xor gates. The flip-flops were implemented using nand gates. The gate level implementation of LFSR has 28 gates. The LFSR is initialized to "0001". The only input to the LFSR is *clk*. At the rising edge of each clock pulse the LFSR is clocked to the next stage. The coverage results obtained for the LFSR are very low as shown in Figure 44. The coverage results were low because the test vectors could not be generated for a certain number of faults. The reason that the test vectors could not be generated was that during fault propagation, because of the feedback in the LFSR circuit, the traversal of the fault site cannot be avoided and the fault sensitization is destroyed due to the traversal of the fault site.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	30	24	22	1	1	6	0
Stuck1	30	15	13	1	1	15	0
All	60	39	35	2	2	21	0
Percentages:		65.00	58.33	3.33	3.33	35.00	

Percentages are of DETECTABLE faults

Figure 44. Fault Coverage Results of LFSR

CONTROLLED COUNTER

The circuit model is a controlled two-bit up/down counter with load limit function and asynchronous clear function. The control section of the counter has a 2-bit register which feeds a 1-of-4 decoder. The 2-bit signal CON, is used to control the decoder. The decoder outputs a 4-bit signal CONSIG that controls the different functions of the controlled counter [16]. A truth table of the decoder is as follows:

Table 6. Truth Table of Control Decoder

CON	STRB	CONSIG
00	R	1000
01	R	0100
10	R	0010
11	R	0001

The functions implementation by the controlled counter are as follows:

Table 7. Function table of Controlled Counter

CONSIG	STRB	CLK	COUNT≠LIM	FUNCTION
1000	R	X	X	clear
0100	F	X	X	load limit
0010	X	R	COUNT≠LIM	count up
0001	X	R	COUNT≠LIM	count down

Figure 45 shows the fault coverage results of the controlled counter.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	34	26	23	3	0	8	0
Stuck1	34	32	28	4	0	2	0
All	68	58	51	7	0	10	0
Percentages:		85.29	75.00	10.29	0.00	14.71	

Percentages are of DETECTABLE faults

Figure 45. Fault Coverage Results of Controlled Counter

As seen from Figure 45, 10 faults were not detected. These faults occur in the limit register of the controlled counter. The BTG could not generate test vectors for these faults as there is no propagation path that can be used to propagate these faults to an observable output.

32 FUNCTION ALU

The circuit model of the ALU has data inputs *A* and *B,* bit vectors of length 4. The output is F, bit vector of length 4. The function is selected by *FSEL,* bit vector of length 4. Table 8 shows the functions implemented by ALU.

Table 8: Functions Implemented by 32 Function ALU

FSEL	M=0 Arithmetic Function	M=1 Logic Function
0000	F = A MINUS 1	$F = \overline{A}$
0001	F = AB MINUS 1	$F = \overline{AB}$
0010	$F = A\overline{B}$ MINUS 1	$F = \overline{A} + B$
0011	F = MINUS 1	F = 0001
0100	$F = A$ PLUS $(A + \overline{B})$	$F = A + B$
0101	$F = AB$ PLUS $(A + \overline{B})$	$F = \overline{B}$
0110	F = A MINUS B MINUS 1	$F = A \oplus B$
0111	$F = A + \overline{B}$	$F = A + \overline{B}$
1000	F = A PLUS (A + B)	$F = \overline{A}B$
1001	F = A PLUS B	$F = A \oplus B$
1010	$F = A\overline{B}$ PLUS $(A + B)$	F = B
1011	F = A + B	F = A + B
1100	F = A PLUS A	F = 0000
1101	F= AB PLUS A	$F = A\overline{B}$
1110	$F = A\overline{B}$ PLUS A	F = AB
1111	F = A	F = A

To find the equivalent implementation at the gate level, the Synopsys Synthesis tool was used. The circuit diagram is as shown in Chapter 5 Figure 12. The gate level implementation of 32-Function ALU has 294 gates. Figure 46 shows the coverage results obtained for the 32-Function ALU.

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	294	284	278	6	0	10	0
Stuck1	294	288	283	5	0	6	0
All	588	572	561	11	0	16	0
Percentages:		97.28	95.41	1.87	0.00	2.72	

Percentages are of DETECTABLE faults.

Figure 46. Fault Coverage Results of 32 Function ALU

MICROPROCESSOR SLICE

The circuit has four input data vectors *A, B, D* and *Q* each of bit-vector length 4. From these inputs the source operands for the ALU *RE* and *S* are selected. The input signal *IL* is used to select the source inputs. Table 9 shows the source inputs selected by *IL*. The functions performed by the ALU are selected by the *IH* signal. Table 10 shows the functions performed by the ALU. Figure 48 shows the circuit description generated using the Synopsys synthesis tool. The gate level implementation has 214 gates. Figure 47 shows the coverage results obtained.

Table 9. Selection of Source Operands

	Source Operands Selected for	
IL	RE	S
000	A	Q
001	A	B
010	0000	Q
011	0000	B
100	0000	A
101	D	A
110	D	Q
111	D	0000

Table 10. Functions Implemented by Microprocessor Slice

IH	Function
000	F = RE PLUS S
001	F = S - RE
010	F = RE - S
011	F = RE + S
100	$F = RE \cdot S$
101	$F = \overline{RE} \cdot S$
110	$F = RE \oplus S$
111	$F = \overline{RE} \oplus \overline{S}$

SUMMARY RECORDS

type	total	faults dropped	hard detects	pot'l detects	cat'phic	notdet'd	undet'bl
Stuck0	229	205	200	5	0	24	0
Stuck1	229	221	213	8	0	8	0
All	458	426	413	13	0	32	0
Percentages:		93.01	90.17	2.84	0.00	6.99	

Percentages are of DETECTABLE faults.

Figure 47. Fault Coverage Results of Microprocessor Slice

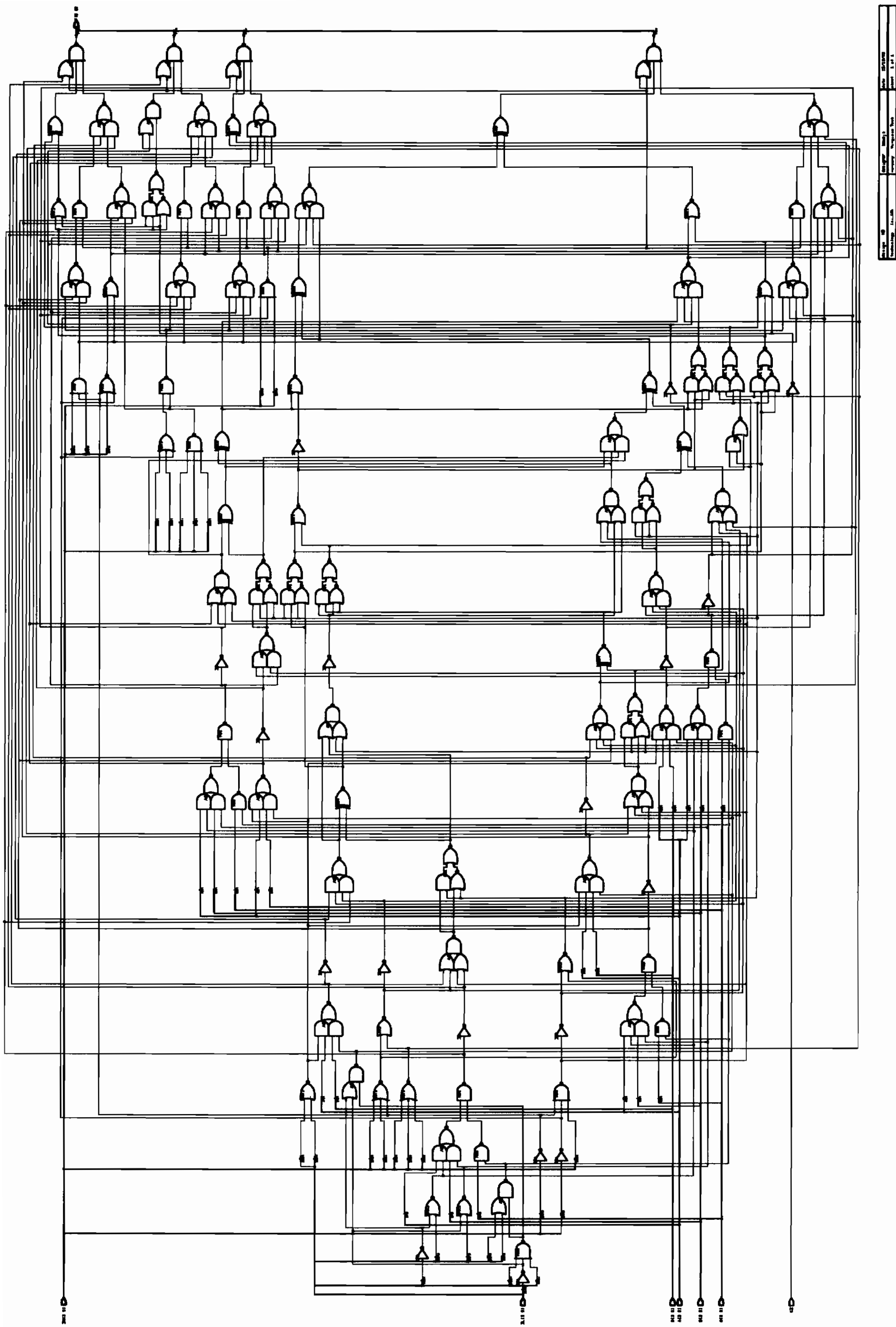


Figure 48. Gate Level Schematic of Microprocessor Slice

6.4 Conclusions from the Fault Coverage Results

Table 5 shows that a high coverage of 98.11% was obtained for the register level primitives. A high gate level coverage was also obtained for the various big circuit models, which can be modeled using the register level primitives.

Hence the Behavioral Test Generator combined with the Heuristic Test Generator can be used as an effective test generator for register level primitives and other big circuits modeled using the register level primitives.

Chapter 7. Proposed Future Work

The BTG and the system for determining gate level fault coverage of the BTG has been explained in the previous chapters. This chapter suggests methods by which the BTG and the system for fault coverage determination can be improved. A method for synthesis directed test generation is also proposed.

7.1 Automatic Conversion of VHDL Circuit Descriptions to Prolog Facts

As is explained in Chapter 3, in the pre-processor of the BTG the conversion of the VHDL description of the circuit to the Prolog facts is done manually. If the BTG were to be used extensively as a test generation tool then a program should be implemented to convert the VHDL description to the Prolog facts automatically. The implementation of the program should use a database that matches the VHDL constructs to the Prolog facts. In future enhancements of the BTG if the mapping of the VHDL constructs to the Prolog facts is changed, the change has to be made in

the database only. Another advantage of using this approach should be, that if additional VHDL constructs are supported by the BTG in future, it will be easy to enhance the translator to support the additional constructs.

7.2 Automatic Synopsys EDIF to GHDL conversion

As explained in Chapter 5 the EDIF to GHDL conversion is being done manually. This process can be automated by obtaining the HILO_LSI10k libraries. The mapping of the lsi_10k primitives to the GHDL primitives was explained in Chapter 5. Another solution to achieve the conversion will be to implement a program that does this mapping automatically.

7.3 Use of BTG as a Process Test Generator (PTG) for Hierarchical Test Generator

The Hierarchical Test Generator is being developed at Virginia Tech for test generation of VHDL behavioral models. The test generation is done to verify the VHDL behavioral models. The Hierarchical Test Generator uses the Process Model Graph (PMG) generated by the Modeler's Assistant as the base for test generation. (The Modeler's Assistant is a tool developed at Virginia Tech that is used for rapid development of behavioral VHDL descriptions using the Process Model Graph (PMG).) In the Hierarchical Test Generator, for each process in the PMG, precomputed tests are stored in a library. Using these primitive tests the

Hierarchical Test Generator constructs a test sequence for the whole entity. The test for the primitives at present is being generated manually. The BTG could be modified to be used as a process test generator for the HTG.

7.4 Synthesis Directed Test Generation

The behavioral VHDL description is synthesized to gate level circuit by mapping the high level VHDL constructs such as case, if, for, etc., to the corresponding gate level circuits. After we obtain the mapping of the high level VHDL constructs such as case, if, for, etc., to the corresponding gate level, we can optimize the method of selection of test vectors for behavioral test generation for these constructs. This method of test generation should result in a high coverage.

Chapter 8. Conclusion

In this thesis it has been shown that the test vectors generated from a Behavioral Test Generator can give a high coverage for the equivalent gate level circuits. The Behavioral Test Generator and the system used for determining the fault coverage were explained. Using this system the time for determination of the gate level coverage of a Behavioral Test Generator is substantially reduced. A Heuristic Test Generator was also described which improved the fault coverage for large microoperation faults.

Suggestions are given in Chapter 7, on the ways to further improve the BTG and decrease the time in obtaining the equivalent gate level GHDL description. A new method of test generation is proposed.

With the high coverage results obtained, test generation could be done using higher level approaches to test generation, which is less complex and expensive then test generation from gate level models.

Bibliography

[1] D. Methvin, "A Chip For All Seasons," PC Week, Nov. 11, 1991, Vol. 8, No. 45.

[2] Ibarra O.H. and Sahni S.K., "Polynomially Complete Fault Detection Problems," IEEE Transactions of Computers, March 1975, pp. 242-249.

[3] S. Y. H. Su and Y. Hsieh, "Testing Functional Faults in Digital Systems Described by Register Transfer Language," IEEE Test Conference, pp. 922-929, 1985.

[4] O'Neill, M.D., Jani, D.D., Cho, C.H. and Armstrong, J.R., "BTG: A Behavioral Test Generator," Proceedings of the Ninth International Conference of Computer Hardware Description Languages and their Applications, Washington D.C., June 1989, pp. 347-361.

[5] Shteingart, Nagle and Grason, "RTG: Automatic Register Level Test Generator," IEEE 22nd Design Automation Conference, pp. 803-807, 1985.

[6] T.S. Lin and S.Y.H. Su, "The S-Algorithm : A Promising Solution for Systematic Functional Test Generation," IEEE Transaction on Computer Aided Design, pp. 250-263, July 1985.

[7] T.S. Lin and S.Y.H. Su, "VLSI Functional Test Pattern Generation - A Design and Implementation," IEEE International Test Conference, pp. 922-929, 1985.

[8] H.P. Chang, W.A. Rogers and J.A. Abraham, "Structured Functional Level Test Generation Using Binary Decision Diagrams," IEEE International Test Conference, pp. 97-104, 1986.

[9] S.B. Akerss, "Binary Decision Diagrams," IEEE Transactions of Computers, pp. 506-516, June, 1978.

[10] M.S. Abadir and H.K. Reghbati, "Functional Test Generation for LSI Circuits Described by Binary Decision Diagrams," IEEE International Test Conference, pp. 483-492, 1985.

[11] Barclay, D. and Armstrong J. R., "A Heuristic Chip-Level Test Generation Algorithm," Proceedings of the 23rd DAC, Las Vegas, NV, June 1986, pp. 257-262.

[12] Chang H. Cho, J. R. Armstrong, "VHDL Semantics for Behavioral Test Generation," 10th International Symposium on CHDLs and their Applications, pp. 395-412, April 1991.

[13] D.S. Barclay, "An Automatic Test Generation Method for Chip-Level Circuit Descriptions," Master's Thesis, VPI & SU, February, 1987.

[14] M.D. O'Neill, "An Improved Chip-Level Test Generation Algorithm," Master's Thesis, VPI & SU, January, 1988.

[15] D.D. Jani, "An Efficient Test Generation Algorithm for Behavioral Descriptions of Digital Devices," Master's Thesis, VPI & SU, December, 1988.

[16] F.S. Lam, "Test Generation for Behavioral Models with Reconvergent Fanout and Feedback," Master's Thesis, VPI & SU, September 1989.

[17] Y.H. Leveldal, P.R. Menon, "Test Generation Algorithm for Computer Hardware Description Languages," IEEE Transactions on Computers, pp. 577-588, June, 1982.

[18] F.E. Norrod, "The E-Algorithm: An Automatic Test Generation Algorithm for Hardware Description Languages", Masters Thesis, VPI & SU, February, 1988.

[19] M. Abramovici, M.A. Breuer and A.D. Friedman, "Digital Systems Testing and Testable Design," Computer Science Press, 1990

[20] J.P. Roth, "Diagnosis of Automatic Failures: A Calculus and a Method," IBM Journal of Research and Development, vol. 10, pp. 278-291, July 1966.

[21] C.W. Cha, W.E. Donath and F. Ozguner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits," IEEE Trans. on Computers, Vol. C-27, No. 3, pp. 193-200, March, 1978.

[22] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," IEEE Trans., on Computers, Vol. C-30, No. 3, pp. 215-222, March 1981.

[23] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," IEEE Trans. on Computers, Vol. C-32, No. 12, pp. 1137-1144, December 1983.

[24] H.D.Hummer, H. Veit, and H. Topfer, "Functional Tests for Hardware Derived from VHDL descriptions", 10th International Symposium on CHDL's and their applications, pp. 433-445, April 1991.

[25] Genrad Incorporated, "System HILO HIFAULT Reference Manual".

[26] Genrad Incorporated, "System HILO GHDL Reference Manual".

[27] Genrad Incorporated, "System HILO DWL Reference Manual".

[28] Synopsys Inc., "Design Compiler Reference Manual".

[29] C. Chao and F.G. Gray, "Microoperation Perturbations in Chip Level Fault Modeling," 25th Design Automation Conference, pp. 579-582, June 1988.

[30] J.R. Armstrong and F.G. Gray, "Structured Logic Design with VHDL," Prentice Hall, March 1993 (to appear).

[31] Paul Stanford and Paul Mancuso, "EDIF Electronic Design Interchange Format Version 2 0 0," Electronic Industries Association EDIF Steering Committee, 1989.

[32] C. H. Cho and J.R. Armstrong, "Heuristic Test Generation For Chip Level Circuit Descriptions," IEEE VLSI Test Workshop, 1988

[33] C.H. Cho, "A Chip Level Fault Coverage Experiment," Research Report, Department of Electrical Engineering, VPI & SU, August 1986.

Appendix A. BTG2DWL Users Guide

From the terminal window invoke the following command (UNIX syntax)

```
% <waveform dir> /btg2dwl
```

where <waveform dir> is the directory were the test vectors are in the BTG format

The program will execute as follows:-

This program reads test patterns generated by BTG
and generates the test pattern in SYSTEM-HILO format

Enter the input file's name:
Type *adder.wav* <CR> (enter the name of the input file in BTG format, e.g. the file as shown in Figure 6)

Enter the output file's name:
Type *adder.dwl* <CR> (enter the name of the output file as required)

Enter The number of inputs:
Type *2* <CR> (enter the number of inputs in circuit model and then enter the description of the inputs, e.g., adder circuit model)

Enter name of input 1 (e.g. clock, address[8]):
Type *a[4]* <CR> (e.g., the a[4] input of the adder)
Enter name of input 2 (e.g. clock, address[8]):
Type *b[4]* <CR> (e.g., the b[4] input of the adder)

Enter The number of outputs:
Type *1* <CR> (enter the number of outputs in circuit model and then enter the description of the outputs, e.g., the adder circuit model)

Enter name of output 1 (e.g. read, data[8]):
Type *c[4]* <CR> (e.g., the c[4] output of the adder)

Appendix B. Circuit Models

CKA (Implemented using D Flip Flop)

VHDL Behavioral Description of CKA

```
entity CKA is
       port(D,CLK1,CLK2 :in BIT;
        ANDOUT: out BIT);
end CKA;

architecture BEH of CKA is
       signal Q1, Q2 :BIT;

begin

process(CLK1)
begin
s1:       if(CLK1= '1' and not(CLK1'stable)) then
s2:       Q1 <= D;
       end if;
end process;

process(CLK2)
begin
s3:       if (CLK2=  '1' and not(CLK1'stable)) then
s4:       Q2 <= Q1;
       end if;
end process;

s5: ANDOUT <= Q1 AND Q2;
end BEH;
```

Prolog Representation of CKA

```
modelspec :-

assert(fileprefix("cka")),
assert(modelname("CKA")),

assert(datatype(d,bit)),      assert(inputpin(d)),
assert(datatype(clk1,bit)),   assert(inputpin(clk1)),
assert(datatype(clk2,bit)),   assert(inputpin(clk2)),
assert(datatype(q1,bit)),
assert(datatype(q2,bit)),
assert(datatype(andout,bit)), assert(outputpin(andout)),

assert(statementtype(s1,if)),
assert(controlexpression(s1,[biteqv,[obj,clk1],[lit,[bit,"R"]]])),
assert(subordinaterange(s1,then,[s2])),
assert(subordinaterange(s1,else,[])),

assert(statementtype(s2,assignment)),
assert(sourceexpression(s2,[obj,d])),
assert(destinationobject(s2,q1)),
assert(edge_response(s2,true)),

assert(statementtype(s3,if)),
assert(controlexpression(s3,[biteqv,[obj,clk2],[lit,[bit,"R"]]])),
assert(subordinaterange(s3,then,[s4])),
assert(subordinaterange(s3,else,[])),

assert(statementtype(s4,assignment)),
assert(sourceexpression(s4,[obj,q1])),
assert(destinationobject(s4,q2)),
assert(edge_response(s4,true)),

assert(statementtype(s5,assignment)),
assert(sourceexpression(s5,[bitand,[obj,q1],[obj,q2]])),
assert(destinationobject(s5,andout)).
```

Gate Level GHDL Description of CKA

```
CCT CKA (andout,d,clk1,clk2)
DFFS   DFF1 (q1,d,clk1)
       DFF2 (q2,q1,clk2);

and (1,1) g1(andout,q1,q2);


wire q1 q2;
input d clk1 clk2;
output andout;

ENDCIRCUIT


CCT DFFS (Q,D,CLK);

input D,CLK;
output Q;

wire w1,w2,w3,w4,w5;

NAND (1,1)
        g1 (w1, w4, w2),
        g2 (w2, CLK, w1),
        g3 (w3, w4, CLK, w2),
        g4 (w4, D, w3),
        g5 (Q, w2, w5),
        g6 (w5, w3, Q);

ENDCIRCUIT
```

ADDER

VHDL Behavioral Description of Adder

```
entity ADDER is
port(a,b:in BIT_VECTOR(0 to 3);
      c:out BIT_VECTOR(0 to 3));
end ADDER;

architecture BEH of ADDER is

begin

process(a,b)

begin

s1: c <=add(a,b)
end process;

end BEH;
```

Prolog Representation of Adder

```
modelspec :-

assert(fileprefix("adder")),
assert(modelname("Simple adder, vectors")),


assert(datatype(a,bv)),assert(bvlength(a,4)),assert(inputpin(a)),
assert(datatype(b,bv)),assert(bvlength(b,4)),assert(inputpin(b)),
assert(datatype(c,bv)),assert(bvlength(c,4)),assert(outputpin(c)),

assert(statementtype(s1,assignment)),
assert(destinationobject(s1,c)),
assert(sourceexpression(s1,[bvadd,[obj,a],[obj,b]])).
```

Gate Level GHDL Description of Adder

```
CCT ADDER (a[0:3],b[0:3],c[0:3],CO)
input a[0:3],b[0:3];
wire w1,w2,w3;

FA

      FA1 (a[0],b[0],0,c[0],w1),
      FA2 (a[1],b[1],w1,c[1],w2),
      FA3 (a[2],b[2],w2,c[2],w3),
      FA4 (a[3],b[3],w3,c[3],CO);

ENDCIRCUIT


CCT FA (A,B,CIN,S,COUT)
input A,B,CIN;
wire w1,w2,w3,w4;
xor(1,1) g1 (w1,A,B);
xor(1,1) g2 (S,w1,CIN);
and(1,1) g3(w2,A,B);
and (1,1) g4(w3,A,CIN);
and(1,1) g5(w4,B,CIN);
or(1,1) g6(COUT,w2,w3,w4);

ENDCIRCUIT
```

Gate Level Implementation of Adder

VHDL Behavioral Description of Adder

```
entity GADD is

port(a1,a2,a3,a4,
    b1,b2,b3,b4:IN BIT;
    s1,s2,s3,s4,c4:OUT BIT);

end GADD;

architecture STRUCTURE of GADD is

signal c1,c2,c3:BIT

begin

s1:   s1<=a1 XOR b1;
s2:   c1<=(a1 AND b1);
s3:   s2<=a2 XOR (b2 XOR c1);
s4:   c2<=(a2 AND b2) OR (c1 AND (a2 OR b2));
s5:   s3<=a3 XOR (b3 XOR c2);
s6:   c3<=(a3 AND b3) OR (c2 AND (a3 OR b3));
s7:   s4<=a4 XOR (b4 XOR c3);
s8:   c4<=(a4 AND b4) OR (c3 AND (a4 OR b4));


end STRUCTURE;
```

Prolog Representation of Adder

```
assert(fileprefix("gadd")),
assert(modelname("gadd")),

assert(datatype(a1,bit)),     assert(inputpin(a1)),
assert(datatype(a2,bit)),     assert(inputpin(a2)),
assert(datatype(a3,bit)),     assert(inputpin(a3)),
assert(datatype(a4,bit)),     assert(inputpin(a4)),
assert(datatype(b1,bit)),     assert(inputpin(b1)),
assert(datatype(b2,bit)),     assert(inputpin(b2)),
assert(datatype(b3,bit)),     assert(inputpin(b3)),
assert(datatype(b4,bit)),     assert(inputpin(b4)),

assert(datatype(s1,bit)),     assert(outputpin(s1)),
assert(datatype(s2,bit)),     assert(outputpin(s2)),
assert(datatype(s3,bit)),     assert(outputpin(s3)),
assert(datatype(s4,bit)),     assert(outputpin(s4)),
assert(datatype(c4,bit)),     assert(outputpin(c4)),

assert(datatype(c1,bit)),
assert(datatype(c2,bit)),
assert(datatype(c3,bit)),

assert(statementtype(s1,assignment)),
assert(sourceexpression(s1,[bitxor,[obj,a1],[obj,b1]])),
assert(destinationobject(s1,s1)),

assert(statementtype(s2,assignment)),
assert(sourceexpression(s2,[bitand,[obj,a1],[obj,b1]])),
assert(destinationobject(s2,c1)),

assert(statementtype(s3,assignment)),
assert(sourceexpression(s3,[bitxor,[obj,a2],[bitxor,[obj,b2],[obj,c1]]])),
assert(destinationobject(s3,s2)),

assert(statementtype(s4,assignment)),
assert(sourceexpression(s4,[bitor,[bitand,[obj,a2],[obj,b2]],[bitand,[obj,c1],
[bitor,[obj,a2],[obj,b2]]]])),
assert(destinationobject(s4,c2)),
```

```
assert(statementtype(s5,assignment)),
assert(sourceexpression(s5,[bitxor,[obj,a3],[bitxor,[obj,b3],[obj,c2]]])),
assert(destinationobject(s5,s3)),

assert(statementtype(s6,assignment)),
assert(sourceexpression(s6,[bitor,[bitand,[obj,a3],[obj,b3]],[bitand,[obj,c2],
[bitor,[obj,a3],[obj,b3]]]])),
assert(destinationobject(s6,c3)),

assert(statementtype(s7,assignment)),
assert(sourceexpression(s7,[bitxor,[obj,a4],[bitxor,[obj,b4],[obj,c3]]])),
assert(destinationobject(s7,s4)),

assert(statementtype(s8,assignment)),
assert(sourceexpression(s8,[bitor,[bitand,[obj,a4],[obj,b4]],[bitand,[obj,c3],
[bitor,[obj,a4],[obj,b4]]]])),
assert(destinationobject(s8,c4)).
```

Gate Level GHDL Description of Adder

```
CCT GADD (a1,a2,a3,a4,b1,b2,b3,b4,s1,s2,s3,s4,c4);
input a1,a2,a3,a4,b1,b2,b3,b4;
wire w2,w3,w4,c1,c2,c3;

xor (1,1) g2 (s1,b1,a1);
xor (1,1) g3 (w2,b2,c1);
xor (1,1) g4 (s2,w2,a2);
xor (1,1) g5 (w3,b3,c2);
xor (1,1) g6 (s3,w3,a3);
xor (1,1) g7 (w4,b4,c3);
xor (1,1) g8 (s4,w4,a4);

and (1,1) g1 (c1,a1,b1);

CS

    CS2(a2,b2,c1,c2),
    CS3(a3,b3,c2,c3),
    CS4(a4,b4,c3,c4);


ENDCIRCUIT

CCT CS (A,B,CIN,COUT)
input A,B,CIN;
wire w1,w2,w3;

and(1,1) g1 (w1,A,B);
or(1,1) g2 (w2,A,B);
and(1,1) g3 (w3,CIN,w2);
or(1,1) g4 (COUT,w1,w3);

ENDCIRCUIT
```

LOOK AHEAD CARRY ADDER (LACA)

VHDL Behavioral Description of LACA

```
entity ADDER is
port(a,b:in BIT_VECTOR(0 to 3);
     c:out BIT_VECTOR(0 to 3));
end ADDER;

architecture BEH of ADDER is

begin

process(a,b)

begin

s1: c <=add(a,b)
end process;

end BEH;
```

Prolog Representation of LACA

```
modelspec :-

assert(fileprefix("adder")),
assert(modelname("ADDER")),


assert(datatype(a,bv)),assert(bvlength(a,4)),assert(inputpin(a)),
assert(datatype(b,bv)),assert(bvlength(b,4)),assert(inputpin(b)),
assert(datatype(c,bv)),assert(bvlength(c,4)),assert(outputpin(c)),

assert(statementtype(s1,assignment)),
assert(destinationobject(s1,c)),
assert(sourceexpression(s1,[bvadd,[obj,a],[obj,b]])).
```

Gate Level GHDL Description of LACA

```
CCT LACADD (a[0:3],b[0:3],c[0:3],CO)
input a[0:3],b[0:3];
wire p1,p2,p3,g0,g1,g2,g3,c2,c3,s1,s2,s3,s4,s5,s6;

xor(1,1)
        gt1 (c[3],a[3],b[3]),
        gt2 (p1,a[2],b[2]),
        gt3 (p2,a[1],b[1]),
        gt4 (p3,a[0],b[0]),
        gt18 (c[2],g0,p1),
        gt19 (c[1],c2,p2),
        gt20 (c[0],c3,p3);

and(1,1)
        gt5 (g0,a[3],b[3]),
        gt6 (g1,a[2],b[2]),
        gt7 (g2,a[1],b[1]),
        gt8 (g3,a[0],b[0]),
        gt9 (s1,p1,g0),
        gt10 (s2,p2,g1),
        gt11 (s3,p2,p1,g0),
        gt12 (s4,p3,g2),
        gt13 (s5,p3,p2,g1),
        gt14 (s6,p3,p2,p1,g0);

or (1,1)
        gt15 (c2,g1,s1),
        gt16 (c3,g2,s2,s3),
        gt17 (CO,g3,s4,s5,s6);


ENDCIRCUIT
```

SUBTRACTOR

VHDL Behavioral Description of Subractor

```
entity SUB is
port(a,b:in BIT_VECTOR(0 to 3);
      c:out BIT_VECTOR(0 to 3));
end SUB;

architecture BEH of SUB is

begin

process(a,b)

begin

s1: c <=sub(a,b)
end process;

end BEH;
```

Prolog Representation of Subtractor

```
modelspec :-

assert(fileprefix("sub")),
assert(modelname("sub")),


assert(datatype(a,bv)),assert(bvlength(a,4)),assert(inputpin(a)),
assert(datatype(b,bv)),assert(bvlength(b,4)),assert(inputpin(b)),
assert(datatype(c,bv)),assert(bvlength(c,4)),assert(outputpin(c)),

assert(statementtype(s1,assignment)),
assert(destinationobject(s1,c)),
assert(sourceexpression(s1,[bvsub,[obj,a],[obj,b]])).
```

Gate Level GHDL Description of Subtractor

```
CCT SUB (a[0:3],b[0:3],c[0:3],CO)

input a[0:3],b[0:3];
wire w1,w2,w3,n1,n2,n3,n4;
output c[0:3];

not (1,1)
        g1 (n1, b[3]),
        g2 (n2, b[2]),
        g3 (n3, b[1]),
        g4 (n4, b[0]);
FA

      FA1 (a[3],n1,1,c[3],w1),
      FA2 (a[2],n2,w1,c[2],w2),
      FA3 (a[1],n3,w2,c[1],w3),
      FA4 (a[0],n4,w3,c[0],CO);

ENDCIRCUIT


CCT FA (A,B,CIN,S,COUT)
input A,B,CIN;
wire w1,w2,w3,w4;
xor(1,1) g1 (w1,A,B);
xor(1,1) g2 (S,w1,CIN);
and(1,1) g3(w2,A,B);
and (1,1) g4(w3,A,CIN);
and(1,1) g5(w4,B,CIN);
or(1,1) g6(COUT,w2,w3,w4);

ENDCIRCUIT
```

4_TO_1 MULTIPLEXER

VHDL Behavioral Description of 4_TO_1 Multiplexer

```
entity 4_TO_1_MUX is
port (IN0,IN1,IN2,IN3: in BIT;
     SEL: in BIT_VECTOR(0 to 1);
     O: out BIT);
end 4_TO_1_MUX;

architecture BEHAVIOR of 4_TO_1_MUX is
begin
process (SEL)
begin

s1:  case SEL(0 to 1) is
s2:  when "00"=> O <= IN0;
s3:  when "01"=> O <= IN1;
s4:  when "10"=> O <= IN2;
s5:  when "11"=> O <= IN3;
 end case;

end process;
end BEHAVIOR;
```

Prolog Representation of 4_TO_1 Multiplexer

```
modelspec :-

assert(fileprefix("mux")),
assert(modelname("MULTIPLEXER")),

assert(datatype(in0,bit)), assert(inputpin(in0)),
assert(datatype(in1,bit)), assert(inputpin(in1)),
assert(datatype(in2,bit)), assert(inputpin(in2)),
assert(datatype(in3,bit)), assert(inputpin(in3)),

assert(datatype(sel,bv)), assert(bvlength(sel,2)), assert(inputpin(sel)),
assert(datatype(o,bit)), assert(outputpin(o)),

assert(statementtype(s1,case)),
assert(controlexpression(s1,[obj,sel])),
assert(subordinaterange(s1,[[bv,"00"]],[s2])),
```

```
assert(subordinaterange(s1,[[bv,"01"]],[s3])),
assert(subordinaterange(s1,[[bv,"10"]],[s4])),
assert(subordinaterange(s1,[[bv,"11"]],[s5])),

assert(statementtype(s2,assignment)),
assert(destinationobject(s2,o)),
assert(sourceexpression(s2,[obj,in0])),

assert(statementtype(s3,assignment)),
assert(destinationobject(s3,o)),
assert(sourceexpression(s3,[obj,in1])),

assert(statementtype(s4,assignment)),
assert(destinationobject(s4,o)),
assert(sourceexpression(s4,[obj,in2])),

assert(statementtype(s5,assignment)),
assert(destinationobject(s5,o)),
assert(sourceexpression(s5,[obj,in3])).
```

Gate Level GHDL Description of 4_TO_1 Multiplexer

```
CCT MUX (in0,in1,in2,in3,sel[0:1],o)
input  in0,in1,in2,in3,sel[0:1];
wire w1,w2,w3,w4,w5,w6;

not(1,1) g1 (w1,sel[0]);
not(1,1) g2 (w2,sel[1]);

and (1,1) g3 (w3,w1,w2,in0);
and (1,1) g4 (w4,w1,sel[1],in1);
and (1,1) g5 (w5,sel[0],w2,in2);
and (1,1) g6 (w6,sel[0],sel[1],in3);

or(1,1) g7 (o,w3,w4,w5,w6);


ENDCIRCUIT
```

QUAD 4_TO_1 MULTIPLEXER

VHDL Behavioral Description of QUAD 4_TO_1 Multiplexer

```
-- BEHAVIORAL MODEL FOR Quad 4 TO 1 MUX

entity Quad_4_TO_1_MUX is

port (IN0,IN1,IN2,IN3: in BIT_VECTOR (0 to 3);
     SEL: in BIT_VECTOR(0 to 1);
     O: out BIT_VECTOR (0 to 3));

end Quad_4_TO_1_MUX;

architecture BEHAVIOR of Quad_4_TO_1_MUX is

begin

process (SEL,IN0,IN1,IN2,IN3)

begin
s1:  case SEL(0 to 1) is
s2:   when "00"=> O <= IN0;
s3:   when "01"=> O <= IN1;
s4:   when "10"=> O <= IN2;
s5:   when "11"=> O <= IN3;
 end case;

end process;

end BEHAVIOR;
```

Prolog Representation of QUAD 4_TO_1 Multiplexer

```
modelspec :-

assert(fileprefix("mx")),
assert(modelname("MX")),


assert(datatype(in0,bv)), assert(bvlength(in0,4)), assert(inputpin(in0)),
assert(datatype(in1,bv)), assert(bvlength(in1,4)), assert(inputpin(in1)),
assert(datatype(in2,bv)), assert(bvlength(in2,4)), assert(inputpin(in2)),
assert(datatype(in3,bv)), assert(bvlength(in3,4)), assert(inputpin(in3)),

assert(datatype(sel,bv)), assert(bvlength(sel,2)), assert(inputpin(sel)),

assert(datatype(o,bv)), assert(bvlength(o,4)), assert(outputpin(o)),

assert(statementtype(s1,case)),
assert(controlexpression(s1,[obj,sel])),
assert(subordinaterange(s1,[[bv,"00"]],[s2])),
assert(subordinaterange(s1,[[bv,"01"]],[s3])),
assert(subordinaterange(s1,[[bv,"10"]],[s4])),
assert(subordinaterange(s1,[[bv,"11"]],[s5])),

assert(statementtype(s2,assignment)),
assert(destinationobject(s2,o)),
assert(sourceexpression(s2,[obj,in0])),

assert(statementtype(s3,assignment)),
assert(destinationobject(s3,o)),
assert(sourceexpression(s3,[obj,in1])),

assert(statementtype(s4,assignment)),
assert(destinationobject(s4,o)),
assert(sourceexpression(s4,[obj,in2])),

assert(statementtype(s5,assignment)),
assert(destinationobject(s5,o)),
assert(sourceexpression(s5,[obj,in3])).
```

Gate Level GHDL Description of QUAD 4_TO_1 Multiplexer

```
CCT MX (in0[0:3],in1[0:3],in2[0:3],in3[0:3],sel[0:1],o[0:3])
input in0[0:3],in1[0:3],in2[0:3],in3[0:3],sel[0:1];

MUX
      MUX1 (in0[0],in1[0],in2[0],in3[0],sel[0:1],o[0]),
      MUX2 (in0[1],in1[1],in2[1],in3[1],sel[0:1],o[1]),
      MUX3 (in0[2],in1[2],in2[2],in3[2],sel[0:1],o[2]),
      MUX4 (in0[3],in1[3],in2[3],in3[3],sel[0:1],o[3]);


ENDCIRCUIT

CCT MUX (in0,in1,in2,in3,sel[0:1],o)
input  in0,in1,in2,in3,sel[0:1];
wire w1,w2,w3,w4,w5,w6;

not(1,1) g1 (w1,sel[0]);
not(1,1) g2 (w2,sel[1]);

and (1,1) g3 (w3,w1,w2,in0);
and (1,1) g4 (w4,w1,sel[1],in1);
and (1,1) g5 (w5,sel[0],w2,in2);
and (1,1) g6 (w6,sel[0],sel[1],in3);

or(1,1) g7 (o,w3,w4,w5,w6);


ENDCIRCUIT
```

DECODER

VHDL Behavioral Description of Decoder

```
entity DEC2 is
port(I: in BIT_VECTOR(1 downto 0);
     O: out BIT_VECTOR(3 downto 0));
end DEC2;

architecture BEH of DEC2 is
begin
process(I)
      begin
s1:             case I is
s2:               when "00" => O <= "0001";
s3:               when "01" => O <= "0010";
s4:               when "10" => O <= "0100";
s5:               when "11" => O <= "1000";
            end case;
end process;
end BEH;
```

Prolog Representation of Decoder

```
modelspec :-

assert(fileprefix("dec2")),
assert(modelname("DEC2")),

assert(datatype(i,bv)), assert(bvlength(i,2)), assert(inputpin(i)),
assert(datatype(o,bv)), assert(bvlength(o,4)),assert(outputpin(o)),

assert(statementtype(s1,case)),
assert(controlexpression(s1,[obj,i])),
assert(subordinaterange(s1,[[bv,"00"]],[s2])),
assert(subordinaterange(s1,[[bv,"01"]],[s3])),
assert(subordinaterange(s1,[[bv,"10"]],[s4])),
assert(subordinaterange(s1,[[bv,"11"]],[s5])),

assert(statementtype(s2,assignment)),
assert(destinationobject(s2,o)),
assert(sourceexpression(s2,[lit,[bv,"0001"]])),

assert(statementtype(s3,assignment)),
```

```
assert(destinationobject(s3,o)),
assert(sourceexpression(s3,[lit,[bv,"0010"]])),

assert(statementtype(s4,assignment)),
assert(destinationobject(s4,o)),
assert(sourceexpression(s4,[lit,[bv,"0100"]])),

assert(statementtype(s5,assignment)),
assert(destinationobject(s5,o)),
assert(sourceexpression(s5,[lit,[bv,"1000"]])).
```

Gate Level GHDL Description of Decoder

```
CCT DEC2 (i[1:0],o[3:0])
input i[1:0];
wire w1,w2;

not (1,1)
        g1 (w1,i[0]),
        g2 (w2,i[1]);

and (1,1)
        g3 (o[0],w1,w2),
        g4 (o[1],i[0],w2),
        g5 (o[2],w1,i[1]),
        g6 (o[3],i[0],i[1]);

ENDCIRCUIT
```

SEQUENTIAL MODELS

JK FLIP-FLOP

VHDL Behavioral Description of JK Flip-Flop

```
entity JKFF is
      port(S,R,J,K,CLK: in BIT; Q,QN:inout BIT);
end JKFF;

architecture BEH of JKFF is

begin
      process(CLK,S,R)
      begin

s1:     if S= '1' and R = '0' then
s2:           Q <= '1';
s3:           QN <= '0';

s4:     elseif S='0' and R = '1' then
s5:           Q <= '0';
s6:           QN <= '1';

s7:     elseif CLK = '1'and not CLK'stable and S='0' and R='0' then

s8:           if J= '1' and K= '0' then
s9:                 Q <= '1';
s10:                QN <= '0';

s11:          elseif J= '0' and K= '1' then
s12:                Q <= '0';
s13:                QN <= '1';

s14:          elseif J= '1' and K= '1' then
s15:                Q <= not Q;
s16:                QN <= not QN;
            end if;
      endif
end process;
end BEH;
```

Prolog Representation of JK Flip-Flop

```
modelspec :-

assert(fileprefix("jkff")),
assert(modelname("JKFF")),


assert(datatype(s,bit)), assert(inputpin(s)),
assert(datatype(r,bit)), assert(inputpin(r)),
assert(datatype(j,bit)), assert(inputpin(j)),
assert(datatype(k,bit)), assert(inputpin(k)),
assert(datatype(clk,bit)), assert(inputpin(clk)),
assert(datatype(q,bit)), assert(outputpin(q)),
assert(datatype(qn,bit)), assert(outputpin(qn)),

assert(statementtype(s1,if)),
assert(controlexpression(s1,[bitand,[biteqv,[obj,s],[lit,[bit,"1"]]],[biteqv,[obj,r],
[lit,[bit,"0"]]]]])),
assert(subordinaterange(s1,then,[s2,s3])),
assert(subordinaterange(s1,else,[s4])),

assert(statementtype(s2,assignment)),
assert(sourceexpression(s2,[lit, [bit,"1"]])),
assert(destinationobject(s2,q)),

assert(statementtype(s3,assignment)),
assert(sourceexpression(s3,[lit, [bit,"0"]])),
assert(destinationobject(s3,qn)),

assert(statementtype(s4,if)),
assert(controlexpression(s4,[bitand,[biteqv,[obj,s],[lit,[bit,"0"]]],[biteqv,[obj,r],
[lit,[bit,"1"]]]]])),
assert(subordinaterange(s4,then,[s5,s6])),
assert(subordinaterange(s4,else,[s7])),

assert(statementtype(s5,assignment)),
assert(sourceexpression(s5,[lit, [bit,"0"]])),
assert(destinationobject(s5,q)),

assert(statementtype(s6,assignment)),
assert(sourceexpression(s6,[lit, [bit,"1"]])),
assert(destinationobject(s6,qn)),

assert(statementtype(s7,if)),
assert(controlexpression(s7,[biteqv,[obj,clk],[lit,[bit,"R"]]])),
```

```
assert(controlexpression(s7,[bitand,[biteqv,[obj,clk],[lit,[bit,"R"]]],[bitand,[biteqv,[obj,
s],[lit,[bit,"0"]]],[biteqv,[obj,r],[lit,[bit,"0"]]]]]])),
assert(subordinaterange(s7,then,[s8])),
assert(subordinaterange(s7,else,[])),

assert(statementtype(s8,if)),
assert(controlexpression(s8,[bitand,[biteqv,[obj,j],[lit,[bit,"1"]]],[biteqv,[obj,k],
[lit,[bit,"0"]]]]])),
assert(subordinaterange(s8,then,[s9,s10])),
assert(subordinaterange(s8,else,[s11])),

assert(statementtype(s9,assignment)),
assert(sourceexpression(s9,[lit, [bit,"1"]])),
assert(destinationobject(s9,q)),

assert(statementtype(s10,assignment)),
assert(sourceexpression(s10,[lit, [bit,"0"]])),
assert(destinationobject(s10,qn)),


assert(statementtype(s11,if)),
assert(controlexpression(s11,[bitand,[biteqv,[obj,j],[lit,[bit,"0"]]],[biteqv,[obj,k],
[lit,[bit,"1"]]]]])),
assert(subordinaterange(s11,then,[s12,s13])),
assert(subordinaterange(s11,else,[s14])),

assert(statementtype(s12,assignment)),
assert(sourceexpression(s12,[lit, [bit,"0"]])),
assert(destinationobject(s12,q)),

assert(statementtype(s13,assignment)),
assert(sourceexpression(s13,[lit, [bit,"1"]])),
assert(destinationobject(s13,qn)),

assert(statementtype(s14,if)),
assert(controlexpression(s14,[bitand,[biteqv,[obj,j],[lit,[bit,"1"]]],[biteqv,[obj,k],
[lit,[bit,"1"]]]]])),
assert(subordinaterange(s14,then,[s15,s16])),
assert(subordinaterange(s14,else,[])),

assert(statementtype(s15,assignment)),
assert(sourceexpression(s15,[bitnot,[obj,q]])),
assert(destinationobject(s15,q)),

assert(statementtype(s16,assignment)),
assert(sourceexpression(s16,[bitnot,[obj,qn]])),
assert(destinationobject(s16,qn)).
```

Gate Level GHDL Circuit Description of JK Flip-Flop

```
CIRCUIT JKFF (S,R,J,K,CLK,Q,QN)
input S,R,J,K,CLK;
output Q,QN;
wire w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13;

not (1,1)
        n1 (w1,CLK),
        n2 (w2,R),
        n3 (w4,S);


and (1,1)
        g1 (w3,QN,J,w1,w2),
        g2 (w5,Q,K,w1,w4),
        g5 (w10,w8,clk,w2),
        g6 (w11,clk,w4,w9);


or (1,1)
       g3 (w6,w3,S),
       g4 (w7,w5,R),
       g7 (w12,w10,S),
       g8 (w13,w11,R);

SRFF
    SRFF1 (w8,w9,w7,w6),
    SRFF2 (Q,QN,w13,w12);

ENDCIRCUIT


CIRCUIT SRFF (Q,QN,S,R)

input S,R;
output Q,QN;

nor (1,1)
        g1 (QN, S,Q),
        g2 (Q,QN,R);

ENDCIRCUIT
```

REGISTER

VHDL Behavioral Description of Register

```
entity REGISTER is
  port(DI : in BIT_VECTOR(0 to 7);
        STRB,DS1,NDS2 : in BIT;
        DOUT : out BIT_VECTOR(0 to 7));
end REGISTER;

architecture BEH of REGISTER is


  signal DID: BIT_VECTOR(0 to 7);
  signal ENBLD: BIT;CTOR(0 to 7);

begin
  process(STRB)
   begin
s1:     if (STRB ='1'and not(STRB'stable)) then
s2:        DID <= DI;
         endif;

   end process;

s3:     ENBLD <= DS1 and not NDS2;

 process(DID,ENBLD)
   begin
s4:     if (ENBLD ='1') then
s5:        DO <= DID;
       else
s6:       DO <= "11111111";
       endif;
   end process;

  end BEH;
```

Prolog Representation of Register

```
modelspec :-

assert(fileprefix("register")),
assert(modelname("Register")),

assert(datatype(di,bv)), assert(bvlength(di,8)),
assert(statevar(di,t)), assert(inputpin(di)),

assert(datatype(strb,bit)), assert(statevar(strb,t)),
assert(inputpin(strb)),

assert(datatype(ds1,bit)), assert(statevar(ds1,t)),
assert(inputpin(ds1)),

assert(datatype(nds2,bit)), assert(statevar(nds2,t)),
assert(inputpin(nds2)),

assert(datatype(did,bv)), assert(bvlength(did,8)),
assert(statevar(did,t)),

assert(datatype(enbld,bit)), assert(statevar(enbld,t)),

assert(datatype(dout,bv)), assert(bvlength(dout,8)),
assert(statevar(dout,[])), assert(outputpin(dout)),

assert(statementtype(s1,if)),
assert(controlexpression(s1,[biteqv,[obj,strb],[lit,[bit,"R"]]])),
assert(subordinaterange(s1,then,[s2])),
assert(subordinaterange(s1,else,[])),

assert(statementtype(s2,assignment)),
assert(sourceexpression(s2,[obj,di])),
assert(destinationobject(s2,did)),

assert(statementtype(s3,assignment)),
assert(sourceexpression(s3,[bitand,[obj,ds1],[bitnot,[obj,nds2]]])),
assert(destinationobject(s3,enbld)),

assert(statementtype(s4,if)),
assert(controlexpression(s4,[biteqv,[obj,enbld],[lit,[bit,"1"]]])),
assert(subordinaterange(s4,then,[s5])),
assert(subordinaterange(s4,else,[s6])),

assert(statementtype(s5,assignment)),
assert(sourceexpression(s5,[obj,did])),
```

```
assert(destinationobject(s5,dout)),

assert(statementtype(s6,assignment)),
assert(sourceexpression(s6,[lit,[bv,"11111111"]])),
assert(destinationobject(s6,dout)).
```

Gate Level GHDL Circuit Description of Register

```
CCT REGISTER (dout[0:7],ds1,nds2,di[0:7],strb)

input ds1,nds2,di[0:7],strb;
wire d1,d2,d3,d4,d5,d6,d7,d8,w1,enable;

DFFS
    FF1 (d1,di[0],strb),
    FF2 (d2,di[1],strb),
    FF3 (d3,di[2],strb),
    FF4 (d4,di[3],strb),
    FF5 (d5,di[4],strb),
    FF6 (d6,di[5],strb),
    FF7 (d7,di[6],strb),
    FF8 (d8,di[7],strb);

not (1,1) g1 (w1,nds2);
and (1,1) g2 (enable,w1,ds1);

BUFFER
      B1 (dout[0],d1, enable),
      B2 (dout[1],d2, enable),
      B3 (dout[2],d3, enable),
      B4 (dout[3],d4, enable),
      B5 (dout[4],d5, enable),
      B6 (dout[5],d6, enable),
      B7 (dout[6],d7, enable),
      B8 (dout[7],d8, enable),

ENDCIRCUIT
```

```
CCT DFFS (Q,D,CLK);

input D,CLK;
output Q;

wire w1,w2,w3,w4,w5;

NAND (1,1)
            g1 (w1, w4, w2),
            g2 (w2, CLK, w1),
            g3 (w3, w4, CLK, w2),
            g4 (w4, D, w3),
            g5 (Q, w2, w5),
            g6 (w5, w3, Q);

ENDCIRCUIT

CIRCUIT BUFFER (OUT,INP,ENABLE)
input INP,ENABLE;
register (1,1) out = loadcase enable,
      0=1,
      1=inp,
      endcase
ENDCIRCUIT
```

COUNTER

VHDL Behavioral Description of Counter

```
 entity CTR is
       port(CLK,CLEAR : in BIT;
             COUNT : out BIT_VECTOR( 0 to 1)) is
 end CTR;


architecture BEH of CTR is

begin

s0:  process (CLEAR)
     begin
s1:     if CLEAR = '1' then
s2:        COUNT <= "00";
       endif;

     end process ;

s3: process (CLK)
    begin
s4:     if (CLK = '1' AND NOT CLK'STABLE AND CLEAR = '0') then
s5:             COUNT <= ADD(COUNT,"01");
        end if;
     end process;

end BEH;
```

Prolog Representation of Counter

```
modelspec :-

assert(modelname("CTR")),
assert(fileprefix("ctr")),

assert(datatype(clk,bit)), assert(inputpin(clk)),
assert(datatype(clear,bit)), assert(inputpin(clear)),
assert(datatype(count,bv)), assert(bvlength(count,2)), assert(outputpin(count)),

assert(statementtype(s1,if)),
assert(controlexpression(s1,[biteqv,[obj,clear],[lit,[bit,"1"]]])),
assert(subordinaterange(s1,then,[s2])),
assert(subordinaterange(s1,else,[])),

assert(statementtype(s2,assignment)),
assert(destinationobject(s2,count)),
assert(sourceexpression(s2,[lit,[bv,"00"]])),

assert(statementtype(s4,if)),
assert(controlexpression(s4,[bitand,[biteqv,[obj,clk],[lit,[bit,"R"]]],
                         [biteqv,[obj,clear],[lit,[bit,"0"]]]])),
assert(subordinaterange(s4,then,[s5])),
assert(subordinaterange(s4,else,[])),

assert(statementtype(s5,assignment)),
assert(destinationobject(s5,count)),
assert(sourceexpression(s5,[bvadd,[obj,count],[lit,[bv,"01"]]])).
```

Gate Level GHDL Description of Counter

```
CCT CTR (count[0:1],clk,clear);
input clk,clear;

wire  abar,bbar,w1,w2, da;

DFFCP
    DFF1 (count[0], abar,da,clk,bclear,vdd),
    DFF2 (count[1],bbar,bbar,clk,bclear,vdd);


not(1,1) n1 (bclear,clear);

and(1,1) a1 (w1,count[0],bbar)
         a2 (w2,abar,count[1]);

or(1,1) a3 (da,w1,w2);

ENDCIRCUIT


CCT DFFCP (q,qn,d,clk,clr,preset);

input d,clk,clr,preset;
output q;

wire w1,w2,w3,w4;

NAND (1,1)
          g1 (w1, preset, w4, w2),
          g2 (w2, clr, clk, w1),
          g3 (w3, w4, clk, w2),
          g4 (w4, d, clr, w3),
          g5 (q, w2, preset, qn),
          g6 (qn, clr, w3, q);

ENDCIRCUIT
```

BAREL SHIFTER

VHDL Behavioral Description of Barel Shifter

```
entity SHIFTER is
port(DATA_IN: in BIT_VECTOR(0 to 7);
     SEL: in BIT_VECTOR(0 to 1); IL,IR: in BIT;
     DATA_OUT: out BIT_VECTOR(0 to 7));
end SHIFTER;

architecture ALG of SHIFTER is
begin
     process(SEL,DATA_IN,IL,IR)

     begin

s1:      case SEL is
s2:           when "00" => DATA_OUT <= DATA_IN;
s3:           when "01" => DATA_OUT <= DATA_IN(1 to 7) & IL;
s4:           when "10" => DATA_OUT <= IR & DATA_IN(0 to 6);
s5:           when "11" => DATA_OUT <= DATA_IN;
     end case;
     end process;
end ALG;
```

Prolog Representation of Barel Shifter

```
modelspec :-

assert(fileprefix("shifter")),
assert(modelname("SHIFTER")),


assert(datatype(il,bv)),     assert(inputpin(il)), assert(bvlength(il,1)),
assert(datatype(ir,bv)),     assert(inputpin(ir)),  assert(bvlength(ir,1)),

assert(datatype(sel,bv)),      assert(bvlength(sel,2)),
assert(inputpin(sel)),

assert(datatype(data_in,bv)),      assert(bvlength(data_in,8)),
assert(inputpin(data_in)),

assert(datatype(data_out,bv)),     assert(bvlength(data_out,8)),
```

```
assert(outputpin(data_out)),

assert(statementtype(s1,case)),
assert(controlexpression(s1,[obj,sel])),
assert(subordinaterange(s1,[[bv,"00"]],[s2])),
assert(subordinaterange(s1,[[bv,"01"]],[s3])),
assert(subordinaterange(s1,[[bv,"10"]],[s4])),
assert(subordinaterange(s1,[[bv,"11"]],[s5])),

assert(statementtype(s2,assignment)),
assert(destinationobject(s2,data_out)),
assert(sourceexpression(s2,[obj,data_in])),

assert(statementtype(s3,assignment)),
assert(destinationobject(s3,data_out)),
assert(sourceexpression(s3,[bvcat,[[bvsubv,6,0],[obj,data_in]],[obj,il]])),

assert(statementtype(s4,assignment)),
assert(destinationobject(s4,data_out)),
assert(sourceexpression(s4,[bvcat,[obj,ir],[[bvsubv,7,1],[obj,data_in]]])),

assert(statementtype(s5,assignment)),
assert(destinationobject(s5,data_out)),
assert(sourceexpression(s5,[obj,data_in])).
```

Gate Level GHDL Description of Barel Shifter

Generated using Synopsys Synthesis Tool

```
** GHDL file generated from EDIF by GENRAD Translator Version 0.8
**==========================================================
**
** Translated on :          Sat Dec 26 16:27:32 1992
**
** EDIF Source filename :     SYNOPSYS_EDIF
** EDIF Version :           2 0 0
**
**************************************************************
** Circuit 'SHIFTER' translated from edif cell 'SHIFTER'
CIRCUIT BIN VERSION=NETLIST SHIFTER (IL, IR, DATA_IN[0], DATA_IN[1],
DATA_IN[2], DATA_IN[3], DATA_IN[4], DATA_IN[5], DATA_IN[6], DATA_IN[7],
SEL[0], SEL[1], DATA_OUT[0], DATA_OUT[1], DATA_OUT[2], DATA_OUT[3],
DATA_OUT[4], DATA_OUT[5], DATA_OUT[6], DATA_OUT[7])

NOT (1,1) U20 ( N50, N33);
AO2 U30 (DATA_IN_1_, N36, DATA_IN_2_, N38, N40);
```

```
AO2 U21 (DATA_IN_5_, N36, N38, DATA_IN_6_, N42);
AO7 U40 (N32, N50, N34, DATA_OUT_7_);
NOT (1,1) U12 (N32, DATA_IN_6_);
AO7 U31 (N50, N39, N40, DATA_OUT_1_);
NOT (1,1) U22 (N47, DATA_IN_2_);
EO1 U13 (SEL_1_, N31, SEL_1_, N31, N36);
AO2 U32 (DATA_IN_0_, N36, IR, N33, N48);
AO2 U23 (DATA_IN_4_, N36, N38, DATA_IN_5_, N37);
NOT (1,1) U14 (N46, DATA_IN_1_);
AO7 U33 (N50, N46, N43, DATA_OUT_2_);
NOT (1,1) U24 (N35, DATA_IN_3_);
NOT (1,1) U15 ( N31, SEL_0_);
AO2 U34 (DATA_IN_7_, N36, IL, N38, N34);
AO2 U25 (DATA_IN_3_, N36, DATA_IN_4_, N38, N49);
NOT (1,1) U16 (N38, N44);
AO7 U35 (N50, N47, N49, DATA_OUT_3_);
NOT (1,1) U26 (N41, DATA_IN_4_);
NAND (1,1) U17 (N44, SEL_1_, N31);
AO2 U36 (DATA_IN_6_, N36, DATA_IN_7_, N38, N30);
AO2 U27 (DATA_IN_2_, N36, DATA_IN_3_, N38, N43);
NOT (1,1) U18 (N39, DATA_IN_0_);
AO7 U37 (N50, N35, N37, DATA_OUT_4_);
AO7 U28 (N44, N46, N48, DATA_OUT_0_);
NOR (1,1) U19 (N33, N31, SEL_1_);
AO7 U38 (N41, N50, N42, DATA_OUT_5_);
NOT (1,1) U29 (N45, DATA_IN_5_);
AO7 U39 (N45, N50, N30, DATA_OUT_6_);

INPUT IL;
INPUT IR;
INPUT DATA_IN[0] IS DATA_IN_0_;
INPUT DATA_IN[1] IS DATA_IN_1_;
INPUT DATA_IN[2] IS DATA_IN_2_;
INPUT DATA_IN[3] IS DATA_IN_3_;
INPUT DATA_IN[4] IS DATA_IN_4_;
INPUT DATA_IN[5] IS DATA_IN_5_;
INPUT DATA_IN[6] IS DATA_IN_6_;
INPUT DATA_IN[7] IS DATA_IN_7_;
INPUT SEL[0] IS SEL_0_;
INPUT SEL[1] IS SEL_1_;
OUTPUT DATA_OUT[0] IS DATA_OUT_0_;
OUTPUT DATA_OUT[1] IS DATA_OUT_1_;
OUTPUT DATA_OUT[2] IS DATA_OUT_2_;
OUTPUT DATA_OUT[3] IS DATA_OUT_3_;
OUTPUT DATA_OUT[4] IS DATA_OUT_4_;
OUTPUT DATA_OUT[5] IS DATA_OUT_5_;
OUTPUT DATA_OUT[6] IS DATA_OUT_6_;
```

```
OUTPUT DATA_OUT[7] IS DATA_OUT_7_;
END
ENDCIRCUIT
```

The GHDL Description of the Above Cells is given in cells.cct

cells.cct

```
CCT AO1P (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1;

and(1,1) g1 (w1,A,B);
nor(1,1) g2 (OUT,w1,C,D);

ENDCIRCUIT
************

CCT AO1 (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1;

and(1,1) g1 (w1,A,B);
nor(1,1) g2 (OUT,w1,C,D);

ENDCIRCUIT

************
CCT AO2 (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1,w2;

and(1,1) g1 (w1,A,B);
and(1,1) g2 (w2,C,D);
nor(1,1) g3 (OUT,w1,w2);

ENDCIRCUIT

************
CCT AO3 (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1;
```

```
or(1,1) g1 (w1,A,B);
nand(1,1) g2 (OUT,w1,C,D);

ENDCIRCUIT

************
CCT AO4 (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1,w2;

or(1,1) g1 (w1,A,B);
or(1,1) g2 (w2,C,D);
nand(1,1) g3 (OUT,w1,w2);

ENDCIRCUIT

************
CCT AO6 (A,B,C,OUT)
input A,B,C;
output OUT;
wire w1;

and(1,1) g1 (w1,A,B);
nor(1,1) g2 (OUT,w1,C);

ENDCIRCUIT

************
CCT AO7 (A,B,C,OUT)
input A,B,C;
output OUT;
wire w1;

or(1,1) g1 (w1,A,B);
nand(1,1) g2 (OUT,w1,C);

ENDCIRCUIT

************
CCT EO1 (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1,w2;

and(1,1) g1 (w1,A,B);
```

```
nor(1,1) g2 (w2,C,D);
nor(1,1) g3 (OUT,w1,w2);

ENDCIRCUIT

************
CCT EON1 (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1,w2;

or(1,1) g1 (w1,A,B);
nand(1,1) g2 (w2,C,D);
nand(1,1) g3 (OUT,w1,w2);

ENDCIRCUIT

************
CCT LD2 (D,G,Q,QN)
input D,G;
output Q,QN;
wire w1,w2,w3;

and (1,1) g1 (w1,D,G);
and (1,1) g2 (w3,w2,Q);
not (1,1) g3 (w2,G);
not (1,1) g4 (QN,Q);
or (1,1) g5 (Q,w1,w3);

ENDCIRCUIT

************
CCT LD1 (D,G,Q,QN);

input D,G;
output Q,QN;
wire w1,w2,w3,w4;

NAND (1,1)
        g1 (w1, w4, w2),
        g2 (w2, G, w1),
        g3 (w3, w4, G, w2),
        g4 (w4, D, w3),
        g5 (Q, w2, QN),
        g6 (QN, w3, Q);

ENDCIRCUIT
```

BIG MODELS

FOUR FUNCTION ALU

VHDL Behavioral Description of Four Function ALU

```
library SYNOPSYS;
use synopsys.bv_arithmetic.all;

entity ALU is
port(A,B: in BIT_VECTOR(3 downto 0);
    FSEL:in BIT_VECTOR(1 downto 0);
    F:out BIT_VECTOR(3 downto 0));
end ALU;

architecture BEH of ALU is

begin

process(A,B,FSEL)
begin

s1: case FSEL is

s2:  when "00" => F <=A;
s3:  when "01" => F <=not(A);
s4:  when "10" => F <=A + B;
s5:  when "11" => F <=A and B;

end case;
end process;

end BEH;
```

Prolog Representation of Four Function ALU

```
modelspec :-

assert(fileprefix("alu")),
assert(modelname("ALU")),

assert(datatype(a,bv)), assert(bvlength(a,4)), assert(inputpin(a)),
assert(datatype(b,bv)), assert(bvlength(b,4)), assert(inputpin(b)),
assert(datatype(fsel,bv)), assert(bvlength(fsel,2)), assert(inputpin(fsel)),
assert(datatype(f,bv)), assert(bvlength(f,4)), assert(outputpin(f)),

assert(statementtype(s1,case)),
assert(controlexpression(s1,[obj,fsel])),
assert(subordinaterange(s1,[[bv,"00"]],[s2])),
assert(subordinaterange(s1,[[bv,"01"]],[s3])),
assert(subordinaterange(s1,[[bv,"10"]],[s4])),
assert(subordinaterange(s1,[[bv,"11"]],[s5])),

assert(statementtype(s2,assignment)),
assert(destinationobject(s2,f)),
assert(sourceexpression(s2,[obj,a])),

assert(statementtype(s3,assignment)),
assert(destinationobject(s3,f)),
assert(sourceexpression(s3,[bvnot,[obj,a]])),

assert(statementtype(s4,assignment)),
assert(destinationobject(s4,f)),
assert(sourceexpression(s4,[bvadd,[obj,a],[obj,b]])),

assert(statementtype(s5,assignment)),
assert(destinationobject(s5,f)),
assert(sourceexpression(s5,[bvand, [obj,a], [obj,b]])).
```

Gate Level GHDL Description of Four Function ALU

```
CCT ALU (a[0:3],b[0:3],fsel[0:1],f[0:3])
WIRE w1,w2,w3,w4,w5,w6,w7,w8;

ADDER ADDER1
          (a[0:3],b[0:3],t[0:3],CO);

and (1,1) g1 (w1,a[0],b[0]);
and (1,1) g2 (w2,a[1],b[1]);
and (1,1) g3 (w3,a[2],b[2]);
and (1,1) g4 (w4,a[3],b[3]);

not(1,1) g5 (w5,a[0]);
not(1,1) g6 (w6,a[1]);
not(1,1) g7 (w7,a[2]);
not(1,1) g8 (w8,a[3]);


MX MX1
     (a[0:3],w5,w6,w7,w8,t[0:3],w1,w2,w3,w4,fsel[0:1],f[0:3]);

ENDCIRCUIT

CCT ADDER (a[0:3],b[0:3],c[0:3],CO)
input a[0:3],b[0:3];
wire w1,w2,w3;

FA

     FA1 (a[0],b[0],0,c[0],w1),
     FA2 (a[1],b[1],w1,c[1],w2),
     FA3 (a[2],b[2],w2,c[2],w3),
     FA4 (a[3],b[3],w3,c[3],CO);

ENDCIRCUIT


CCT FA (A,B,CIN,S,COUT)
input A,B,CIN;
wire w1,w2,w3,w4;
xor(1,1) g1 (w1,A,B);
xor(1,1) g2 (S,w1,CIN);
and(1,1) g3(w2,A,B);
and (1,1) g4(w3,A,CIN);
and(1,1) g5(w4,B,CIN);
or(1,1) g6(COUT,w2,w3,w4);
```

```
ENDCIRCUIT

CCT MX (in0[0:3],in1[0:3],in2[0:3],in3[0:3],sel[0:1],o[0:3])
input in0[0:3],in1[0:3],in2[0:3],in3[0:3],sel[0:1];

MUX
    MUX1 (in0[0],in1[0],in2[0],in3[0],sel[0:1],o[0]),
    MUX2 (in0[1],in1[1],in2[1],in3[1],sel[0:1],o[1]),
    MUX3 (in0[2],in1[2],in2[2],in3[2],sel[0:1],o[2]),
    MUX4 (in0[3],in1[3],in2[3],in3[3],sel[0:1],o[3]);


ENDCIRCUIT

CCT MUX (in0,in1,in2,in3,sel[0:1],o)
input  in0,in1,in2,in3,sel[0:1];
wire w1,w2,w3,w4,w5,w6;

not(1,1) g1 (w1,sel[0]);
not(1,1) g2 (w2,sel[1]);

and (1,1) g3 (w3,w1,w2,in0);
and (1,1) g4 (w4,w1,sel[1],in1);
and (1,1) g5 (w5,sel[0],w2,in2);
and (1,1) g6 (w6,sel[0],sel[1],in3);

or(1,1) g7 (o,w3,w4,w5,w6);


ENDCIRCUIT
```

LINEAR FEEDBACK SHIFT REGISTER (LFSR)

VHDL Behavioral Description of LFSR

```
entity LFSR is
port (clk: in BIT;
     a1,a2,a3,a4: inout BIT);
end LFSR;

architecture BEH of LFSR is
signal b1:bit;

process(reset)

s1: if reset = 1 then
s2: a1 <= '0';
s3: a2 <='0';
s4: a3 <= '0';
s5: a4 <= '1';

end process

process(clk)


s6: if (clk='1' and (not clk'stable) and reset='0')

s7:  a1 <= a4;
s8:  a2 <= a1 xor a4;
s9:  a3 <= a2 xor a4;
s10:  a4 <= a3 xor a4;

end if;

end process;

end BEH;
```

Prolog Representation of LFSR

```
modelspec :-

assert(fileprefix("lfsr")),
assert(modelname("LFSR")),

assert(datatype(clk,bit)),   assert(inputpin(clk)),
assert(datatype(reset,bit)),  assert(inputpin(reset)),
assert(datatype(a1,bit)),  assert(outputpin(a1)),
assert(datatype(a2,bit)),  assert(outputpin(a2)),
assert(datatype(a3,bit)),  assert(outputpin(a3)),
assert(datatype(a4,bit)),  assert(outputpin(a4)),


assert(statementtype(s1,if)),
assert(controlexpression(s1,[biteqv,[obj,reset],[lit,[bit,"1"]]])),
assert(subordinaterange(s1,then,[s2,s3,s4,s5])),
assert(subordinaterange(s1,else,[])),


assert(statementtype(s2,assignment)),
assert(destinationobject(s2,a1)),
assert(sourceexpression(s2,[lit,[bit,"0"]])),
assert(edge_response(s2,true)),


assert(statementtype(s3,assignment)),
assert(destinationobject(s3,a2)),
assert(sourceexpression(s3,[lit,[bit,"0"]])),
assert(edge_response(s3,true)),

assert(statementtype(s4,assignment)),
assert(destinationobject(s4,a3)),
assert(sourceexpression(s4,[lit,[bit,"0"]])),
assert(edge_response(s4,true)),

assert(statementtype(s5,assignment)),
assert(destinationobject(s5,a4)),
assert(sourceexpression(s5,[lit,[bit,"1"]])),
assert(edge_response(s5,true)),

assert(statementtype(s6,if)),
assert(controlexpression(s6,[bitand,[biteqv,[obj,clk],[lit,[bit,"R"]]],[biteqv,[obj,reset],
[lit,[bit,"0"]]]])),
assert(subordinaterange(s6,then,[s7,s8,s9,s10])),
assert(subordinaterange(s6,else,[])),
```

```
assert(statementtype(s7,assignment)),
assert(destinationobject(s7,a1)),
assert(sourceexpression(s7,[obj,a4])),
assert(edge_response(s7,true)),


assert(statementtype(s8,assignment)),
assert(destinationobject(s8,a2)),
assert(sourceexpression(s8,[bitxor,[obj,a1],[obj,a4]])),
assert(edge_response(s8,true)),

assert(statementtype(s9,assignment)),
assert(destinationobject(s9,a3)),
assert(sourceexpression(s9,[bitxor,[obj,a2],[obj,a4]])),
assert(edge_response(s9,true)),

assert(statementtype(s10,assignment)),
assert(destinationobject(s10,a4)),
assert(sourceexpression(s10,[bitxor,[obj,a3],[obj,a4]])),
assert(edge_response(s10,true)).
```

Gate Level GHDL Description of LFSR

```
CCT LFSR (a1,a2,a3,a4,reset,clk)

input reset, clk;
output a1,a2,a3,a4;

wire w1,w2,w3,w4;

DFF
      DFF1 (a1,a4,clk,w1,1),
      DFF2 (a2,w2,clk,w1,1),
      DFF3 (a3,w3,clk,w1,1),
      DFF4 (a4,w4,clk,1,w1);

xor (1,1)
      g2 (w2,a1,a4),
      g3 (w3,a2,a4),
      g4 (w4,a3,a4);


not (1,1) g1 (w1,reset);

ENDCIRCUIT

CCT DFF (Q,D,CLK,CLR,PRESET);

input D,CLK,CLR,PRESET;
output Q;

wire w1,w2,w3,w4,w5;

NAND (1,1)
            g1 (w1, PRESET, w4, w2),
            g2 (w2, CLR, CLK, w1),
            g3 (w3, w4, CLK, w2),
            g4 (w4, D, CLR, w3),
            g5 (Q, w2, PRESET, w5),
            g6 (w5, CLR, w3, Q);

ENDCIRCUIT
```

CONTROLLED COUNTER

VHDL Behavioral Description of Controlled Counter

```
  entity CONCTR is
        port(CLK,STRB : in BIT;
        CON : in BIT_VECTOR(1  downto 0);
        DATA : in BIT_VECTOR(1 downto 0);
        COUNT : out BIT_VECTOR(1 downto 0)) is
  end CONCTR;


  architecture ARCH of CONCTR is

    signal
      LIM : BIT_VECTOR(1 downto 0);
      CONSIG : BIT_VECTOR(3 downto 0);

begin

 s01: DECODE: process (STRB)
    begin
 s02:   if STRB='1' and not STRB'stable  then
 s03:      case INTVAL(CON) is
           when 0 =>
 s04:          CONSIG <= "1000";
           when 1 =>
 s05:          CONSIG <= "0100";
           when 2 =>
 s06:          CONSIG <= "0010";
           when 3 =>
 s07:          CONSIG <= "0001";
         end case;
      end if;
    end process DECODE;

 s08: LOAD_LIMIT: process (STRB)
    begin
 s09:   if (STRB='0' and not STRB'stable and CONSIG(2)='1') then
 s10:      LIM <= DATA;
      end if;
    end process LOAD_LIMIT;
```

```
s11: CLEAR_CTR:  process (CONSIG(3))
    begin
 s12:   if CONSIG(3)='1' then
 s13:      COUNT <= "00";
      end if;
    end process CLEAR_CTR;

 s14: CNT_UP: process (CLK)
    begin
 s15:   if (CLK='1' and not CLK'stable and CONSIG(1)='1'
         and  not(count = lim) then
 s16:           COUNT <= ADD(COUNT,"01");
      endif;
    end process CNT_UP;

 s17: CNT_DOWN: process (CLK)
    begin
 s18:   if (CLK='1' and not CLK'stable and CONSIG(0)='1'
         and not(count = lim) then
 s19:           COUNT <= SUB(COUNT,"01");
      endif;
    end process CNT_DOWN;

 end ARCH;
```

Prolog Representation of Controlled Counter

```
modelspec :-

assert(fileprefix("conctr")),
assert(modelname("CONCTR")),

assert(datatype(clk,bit)), assert(inputpin(clk)),
assert(datatype(strb,bit)), assert(inputpin(strb)),
assert(datatype(con,bv)), assert(bvlength(con,2)), assert(inputpin(con)),
assert(datatype(data,bv)), assert(bvlength(data,2)), assert(inputpin(data)),
assert(datatype(count,bv)), assert(bvlength(count,2)), assert(outputpin(count)),
assert(datatype(lim,bv)), assert(bvlength(lim,2)),
assert(datatype(consig,bv)), assert(bvlength(consig,4)),

assert(statementtype(s2,if)),
assert(controlexpression(s2,[biteqv,[obj,strb],[lit,[bit,"R"]]])),
assert(subordinaterange(s2,then,[s3])),
assert(subordinaterange(s2,else,[])),

assert(statementtype(s3,case)),
assert(controlexpression(s3,[obj,con])),
assert(subordinaterange(s3,[[bv,"00"]],[s4])),
assert(subordinaterange(s3,[[bv,"01"]],[s5])),
assert(subordinaterange(s3,[[bv,"10"]],[s6])),
assert(subordinaterange(s3,[[bv,"11"]],[s7])),

assert(statementtype(s4,assignment)),
assert(destinationobject(s4,consig)),
assert(sourceexpression(s4,[lit,[bv,"1000"]])),

assert(statementtype(s5,assignment)),
assert(destinationobject(s5,consig)),
assert(sourceexpression(s5,[lit,[bv,"0100"]])),

assert(statementtype(s6,assignment)),
assert(destinationobject(s6,consig)),
assert(sourceexpression(s6,[lit,[bv,"0010"]])),

assert(statementtype(s7,assignment)),
assert(destinationobject(s7,consig)),
assert(sourceexpression(s7,[lit,[bv,"0001"]])),

assert(statementtype(s9,if)),
assert(controlexpression(s9,[bitand,[biteqv,[obj,strb],[lit,[bit,"F"]]],
        [bveq,[[bvsubv,2],[obj,consig]],[lit,[bv,"1"]]]])),
assert(subordinaterange(s9,then,[s10])),
```

```
assert(subordinaterange(s9,else,[])),

assert(statementtype(s10,assignment)),
assert(destinationobject(s10,lim)),
assert(sourceexpression(s10,[obj,data])),

assert(statementtype(s12,if)),
assert(controlexpression(s12,[bveq,[[bvsubv,3],[obj,consig]],[lit,[bv,"1"]]])),
assert(subordinaterange(s12,then,[s13])),
assert(subordinaterange(s12,else,[])),

assert(statementtype(s13,assignment)),
assert(destinationobject(s13,count)),
assert(sourceexpression(s13,[lit,[bv,"00"]])),

assert(statementtype(s15,if)),
assert(controlexpression(s15,
        [bitand,[biteqv,[obj,clk],[lit,[bit,"R"]]],
              [bitand,[bveq,[[bvsubv,1],[obj,consig]],[lit,[bv,"1"]]],
                    [bitnot,[bveq,[obj,count],[obj,lim]]]]]])),
assert(subordinaterange(s15,then,[s16])),
assert(subordinaterange(s15,else,[])),

assert(statementtype(s16,assignment)),
assert(destinationobject(s16,count)),
assert(sourceexpression(s16,[bvadd,[obj,count],[lit,[bv,"01"]]])),

assert(statementtype(s18,if)),
assert(controlexpression(s18,
        [bitand,[biteqv,[obj,clk],[lit,[bit,"R"]]],
              [bitand,[bveq,[[bvsubv,0],[obj,consig]],[lit,[bv,"1"]]],
                    [bitnot,[bveq,[obj,count],[obj,lim]]]]]])),
assert(subordinaterange(s18,then,[s19])),
assert(subordinaterange(s18,else,[])),

assert(statementtype(s19,assignment)),
assert(destinationobject(s19,count)),
assert(sourceexpression(s19,[bvsub,[obj,count],[lit,[bv,"01"]]])).
```

Gate Level GHDL Description of Control Counter

```
CCT CONCTR (count[0:1],clk,strobe,data[0:1],con[0:1],mclear)

input clk,strobe, data[0:1], con[0:1], mclear;
wire count[0:1], bcon[0:1], consig[0:3], lim[0:1],w4,w5,w6,w7,en;
supply0 gnd;

COUNTER CON1 (count[0:1], clk,consig[3],en,consig[0]);

DECODE DEC1 (consig[0:3], bcon[0:1]);

DFFP FF1 (bcon[0],con[0],strobe,mclear),
       FF2 (bcon[1],con[1],strobe,mclear);

DFFN FF3 (lim[0],data[0],strb,consig[2],mclear),
        FF4 (lim[1],data[1],strb,consig[2],mclear);

and  (1,1) g1 (en,w6,w7);
xor (1,1) g2 (w4,lim[0],count[0]),
          g3 (w5,lim[1],count[1]);
or (1,1)   g4 (w7,consig[0],consig[1])
          g5 (w6,w4,w5);


ENDCIRCUIT

CCT COUNTER (count[0:1],clk,clear,en,updown)

input clk,clear,en,updown;
wire count[0:1],w1,w2,w3,w4;

xor (1,1)  g1 (w1,updown,count[1]),
          g2 (w3,w2,count[0]),
          g3 (w4,en,count[1]);
and(1,1)   g4 (w2,en,w1);

DFFP    FF1 (count[0],w3,clk,clear),
       FF2 (count[1],w4,clk,clear);

ENDCIRCUIT

CCT DECODE (consig[0:3],con[0:1])

input con[0:1];
wire ncon0,ncon1,consig[0:3];
```

```
not (1,1)
       g1 (ncon0, con[0]),
       g2 (ncon1, con[1]);
and (1,1)
          g3 (consig[0],con[0],con[1]),
         g4 (consig[1],con[0],ncon1),
         g5 (consig[2],con[1],ncon0),
         g6 (consig[3],ncon0,ncon1);

ENDCIRCUIT
```

32 FUNCTION ALU

VHDL Behavioral Description of 32 Function ALU

```
library SYNOPSYS;
use synopsys.bv_arithmetic.all;

entity FUNGEN is
port(A,B: in BIT_VECTOR(0 to 3);
    FSEL:in BIT_VECTOR(0 to 3);
    M:in BIT;
    F:out BIT_VECTOR(0 to 3));
end FUNGEN;

architecture BEH of FUNGEN is

begin

process(M,FSEL,A,B)

begin

s1: if M ='0' then


s2: case FSEL is
s3:      when "0000" => F <= A - "0001" ;

s4:      when "0001" => F <= (A and B) - "0001" ;

s5:      when "0010" => F <= (A and (not B)) - "0001" ;

s6:       when "0011" => F <= "0000" - "0001" ;

s7:      when "0100" => F <= A + (A or (not B)) ;

s8:      when "0101" => F <= (A and B) + (A or (not B)) ;

s9:      when "0110" => F <= A - B -"0001" ;

s10:      when "0111" => F <= (A or (not B)) ;

s11:      when "1000" => F <= A + (A or B) ;
```

```
s12:     when "1001" => F <= A + B ;

s13:     when "1010" => F <= (A and (not B)) + (A or B) ;

s14:     when "1011" => F <= (A or B)  ;

s15:     when "1100" => F <= A + A  ;

s16:     when "1101" => F <= (A and B) + A  ;

s17:      when "1110" => F <= (A and (not B)) + A  ;

s18:      when "1111" => F <= A ;

end case;

      end if;

s19: if M ='1' then

s20: case FSEL is
s21:      when "0000" => F <= not A;

s22:      when "0001" => F <= not (A and B);

s23:      when "0010" => F <= (not A) or B;

s24:      when "0011" => F <="0001";

s25:      when "0100" => F <= not (A or B);

s26:      when "0101" => F <= not B;

s27:      when "0110" => F <=not ( A xor B);

s28:      when "0111" => F <= A or (not B);

s29:      when "1000" => F <= (not A) and B;

s30:      when "1001" => F <=  A xor B;

s31:      when "1010" => F <= B;

s32:      when "1011" => F <= (A or B);

s33:      when "1100" => F <= "0000";
```

```
s34:      when "1101" => F <= (A and (not B));

s35:      when "1110" => F <= A and  B;

s36:      when "1111" => F <= A;

end case;
end if;


end process;
end BEH;
```

Prolog Representation of 32 Function ALU

```
modelspec :-

assert(fileprefix("fungen")),
assert(modelname("FUNGEN")),

assert(datatype(a,bv)), assert(bvlength(a,4)), assert(inputpin(a)),
assert(datatype(b,bv)), assert(bvlength(b,4)), assert(inputpin(b)),
assert(datatype(fsel,bv)), assert(bvlength(fsel,4)), assert(inputpin(fsel)),
assert(datatype(m,bit)),     assert(inputpin(m)),
assert(datatype(f,bv)), assert(bvlength(f,4)), assert(outputpin(f)),

assert(statementtype(s1,if)),
assert(controlexpression(s1,[biteqv,[obj,m],[lit,[bit,"0"]]])),
assert(subordinaterange(s1,then,[s2])),
assert(subordinaterange(s1,else,[])),

assert(statementtype(s2,case)),
assert(controlexpression(s2,[obj,fsel])),
assert(subordinaterange(s2,[[bv,"0000"]],[s3])),
assert(subordinaterange(s2,[[bv,"0001"]],[s4])),
assert(subordinaterange(s2,[[bv,"0010"]],[s5])),
assert(subordinaterange(s2,[[bv,"0011"]],[s6])),
assert(subordinaterange(s2,[[bv,"0100"]],[s7])),
assert(subordinaterange(s2,[[bv,"0101"]],[s8])),
assert(subordinaterange(s2,[[bv,"0110"]],[s9])),
assert(subordinaterange(s2,[[bv,"0111"]],[s10])),
assert(subordinaterange(s2,[[bv,"1000"]],[s11])),
assert(subordinaterange(s2,[[bv,"1001"]],[s12])),
assert(subordinaterange(s2,[[bv,"1010"]],[s13])),
assert(subordinaterange(s2,[[bv,"1011"]],[s14])),
assert(subordinaterange(s2,[[bv,"1100"]],[s15])),
```

```
assert(subordinaterange(s2,[[bv,"1101"]],[s16])),
assert(subordinaterange(s2,[[bv,"1110"]],[s17])),
assert(subordinaterange(s2,[[bv,"1111"]],[s18])),


assert(statementtype(s3,assignment)),
assert(destinationobject(s3,f)),
assert(sourceexpression(s3,[bvsub,[obj,a],[lit,[bv,"0001"]]])),

assert(statementtype(s4,assignment)),
assert(destinationobject(s4,f)),
assert(sourceexpression(s4,[bvsub,[bvand,[obj,a],[obj,b]],[lit,[bv,"0001"]]])),

assert(statementtype(s5,assignment)),
assert(destinationobject(s5,f)),
assert(sourceexpression(s5,[bvsub,[bvand,[obj,a],[bvnot,[obj,b]]],[lit,[bv,"0001"]]])),

assert(statementtype(s6,assignment)),
assert(destinationobject(s6,f)),
assert(sourceexpression(s6,[bvsub,[lit,[bv,"0000"]],[lit,[bv,"0001"]]])),

assert(statementtype(s7,assignment)),
assert(destinationobject(s7,f)),
assert(sourceexpression(s7,[bvadd,[obj,a],[bvor,[obj,a],[bvnot,[obj,b]]]])),

assert(statementtype(s8,assignment)),
assert(destinationobject(s8,f)),
assert(sourceexpression(s8,[bvadd,[bvand,[obj,a],[obj,b]],[bvor,[obj,a],[bvnot,[obj,b]
]]])),


assert(statementtype(s9,assignment)),
assert(destinationobject(s9,f)),
assert(sourceexpression(s9,[bvsub,[obj,a],[bvadd,[obj,b],[lit,[bv,"0001"]]]])),

assert(statementtype(s10,assignment)),
assert(destinationobject(s10,f)),
assert(sourceexpression(s10,[bvor,[obj,a],[bvnot,[obj,b]]])),

assert(statementtype(s11,assignment)),
assert(destinationobject(s11,f)),
assert(sourceexpression(s11,[bvadd,[obj,a],[bvor,[obj,a],[obj,b]]])),

assert(statementtype(s12,assignment)),
assert(destinationobject(s12,f)),
assert(sourceexpression(s12,[bvadd,[obj,a],[obj,b]])),
```

```
assert(statementtype(s13,assignment)),
assert(destinationobject(s13,f)),
assert(sourceexpression(s13,[bvadd,[bvand,[obj,a],[bvnot,[obj,b]]],[bvor,[obj,a],[obj,
b]]])),

assert(statementtype(s14,assignment)),
assert(destinationobject(s14,f)),
assert(sourceexpression(s14,[bvor,[obj,a],[obj,b]])),


assert(statementtype(s15,assignment)),
assert(destinationobject(s15,f)),
assert(sourceexpression(s15,[bvadd,[obj,a],[obj,a]])),

assert(statementtype(s16,assignment)),
assert(destinationobject(s16,f)),
assert(sourceexpression(s16,[bvadd,[obj,a],[bvand,[obj,a],[obj,b]]])),

assert(statementtype(s17,assignment)),
assert(destinationobject(s17,f)),
assert(sourceexpression(s17,[bvadd,[obj,a],[bvand,[obj,a],[bvnot,[obj,b]]]])),
assert(statementtype(s18,assignment)),
assert(destinationobject(s18,f)),
assert(sourceexpression(s18,[obj,a])),

assert(statementtype(s19,if)),
assert(controlexpression(s19,[biteqv,[obj,m],[lit,[bit,"1"]]])),
assert(subordinaterange(s19,then,[s20])),
assert(subordinaterange(s19,else,[])),

assert(statementtype(s20,case)),
assert(controlexpression(s20,[obj,fsel])),
assert(subordinaterange(s20,[[bv,"0000"]],[s21])),
assert(subordinaterange(s20,[[bv,"0001"]],[s22])),
assert(subordinaterange(s20,[[bv,"0010"]],[s23])),
assert(subordinaterange(s20,[[bv,"0011"]],[s24])),
assert(subordinaterange(s20,[[bv,"0100"]],[s25])),
assert(subordinaterange(s20,[[bv,"0101"]],[s26])),
assert(subordinaterange(s20,[[bv,"0110"]],[s27])),
assert(subordinaterange(s20,[[bv,"0111"]],[s28])),
assert(subordinaterange(s20,[[bv,"1000"]],[s29])),
assert(subordinaterange(s20,[[bv,"1001"]],[s30])),
assert(subordinaterange(s20,[[bv,"1010"]],[s31])),
assert(subordinaterange(s20,[[bv,"1011"]],[s32])),
assert(subordinaterange(s20,[[bv,"1100"]],[s33])),
assert(subordinaterange(s20,[[bv,"1101"]],[s34])),
assert(subordinaterange(s20,[[bv,"1110"]],[s35])),
```

```
assert(subordinaterange(s20,[[bv,"1111"]],[s36])),


assert(statementtype(s21,assignment)),
assert(destinationobject(s21,f)),
assert(sourceexpression(s21,[bvnot,[obj,a]])),

assert(statementtype(s22,assignment)),
assert(destinationobject(s22,f)),
assert(sourceexpression(s22,[bvnot,[bvand,[obj,a],[obj,b]]])),

assert(statementtype(s23,assignment)),
assert(destinationobject(s23,f)),
assert(sourceexpression(s23,[bvor,[obj,b],[bvnot,[obj,a]]])),

assert(statementtype(s24,assignment)),
assert(destinationobject(s24,f)),
assert(sourceexpression(s24,[lit,[bv,"0001"]])),

assert(statementtype(s25,assignment)),
assert(destinationobject(s25,f)),
assert(sourceexpression(s25,[bvnot,[bvor,[obj,a],[obj,b]]])),

assert(statementtype(s26,assignment)),
assert(destinationobject(s26,f)),
assert(sourceexpression(s26,[bvnot,[obj,b]])),


assert(statementtype(s27,assignment)),
assert(destinationobject(s27,f)),
assert(sourceexpression(s27,[bvnot,[bvxor,[obj,a],[obj,b]]])),

assert(statementtype(s28,assignment)),
assert(destinationobject(s28,f)),
assert(sourceexpression(s28,[bvor,[obj,a],[bvnot,[obj,b]]])),

assert(statementtype(s29,assignment)),
assert(destinationobject(s29,f)),
assert(sourceexpression(s29,[bvand,[bvnot,[obj,a]],[obj,b]])),

assert(statementtype(s30,assignment)),
assert(destinationobject(s30,f)),
assert(sourceexpression(s30,[bvxor,[obj,a],[obj,b]])),

assert(statementtype(s31,assignment)),
assert(destinationobject(s31,f)),
assert(sourceexpression(s31,[obj,b])),
```

```
assert(statementtype(s32,assignment)),
assert(destinationobject(s32,f)),
assert(sourceexpression(s32,[bvor,[obj,a],[obj,b]])),

assert(statementtype(s33,assignment)),
assert(destinationobject(s33,f)),
assert(sourceexpression(s33,[lit,[bv,"0000"]])),

assert(statementtype(s34,assignment)),
assert(destinationobject(s34,f)),
assert(sourceexpression(s34,[bvand,[obj,a],[bvnot,[obj,b]]])),

assert(statementtype(s35,assignment)),
assert(destinationobject(s35,f)),
assert(sourceexpression(s35,[bvand,[obj,a],[obj,b]])),

assert(statementtype(s36,assignment)),
assert(destinationobject(s36,f)),
assert(sourceexpression(s36,[obj,a])).
```

Gate Leval GHDL description of 32 Function ALU
Generated using the Synopsys Synthesis Tool

```
** GHDL file generated from EDIF by GENRAD Translator Version 0.8
**======================================================
**
** Translated on :          Mon Jan  4 18:43:20 1993
**
** EDIF Source filename :     SYNOPSYS_EDIF
** EDIF Version :           2 0 0
**
******************************************************************
** Circuit 'FUNGEN' translated from edif cell 'FUNGEN'
CIRCUIT BIN VERSION=NETLIST FUNGEN (M, A[0], A[1], A[2], A[3], B[0],
B[1], B[2], B[3], FSEL[0], FSEL[1], FSEL[2], FSEL[3], F[0], F[1], F[2],
F[3])

EO1 MINUS_174 (N709, N705, N709, N705, N635);
AO7 MINUS_255 (N693, N623, N685, N662);
AO2 MINUS_165 (N661, N683, N607, N680, N685);
AO2 MINUS_246 (N598, N662, N664, N666, N595);
AO2 MINUS_156 (N695, N678, A_2_, N582, N648);
AO2 MINUS_237 (A_2_, N603, N667, N681, N684);
AO2 MINUS_147 (N660, N619, N701, N602, N630);
NAND (1,1) MINUS_228 (N694, N610, N712);
EO1 MINUS_138 (N699, N695, N671, N695, N720);
AO7 MINUS_219 (N643, N713, N684, N639);
NOT (1,1) MINUS_129 (N633, N649);
NAND (1,1) MINUS_283 (N617, N610, N719);
AO1P MINUS_193 (N682, N571, N572, N573, N679);
NOR (1,1) MINUS_274 (N625, N626, N627);
AO6 MINUS_184 (N575, N661, N671, N660);
NAND (1,1) MINUS_265 (N710, N708, N625);
NAND (1,1) MINUS_175 (N699, N715, N629);
AO7 MINUS_256 (N691, N693, N696, N689);
NAND (1,1) MINUS_166 (N678, N583, N694);
NAND (1,1) MINUS_247 (N729, N648, N647);
AO2 MINUS_157 (B_2_, N672, N702, N674, N647);
NAND (1,1) MINUS_238 (N638, N600, N711);
AO2 MINUS_148 (N653, N590, N654, N691, N652);
EON1 MINUS_229 (N710, N594, N590, N594, N592);
AO2 MINUS_139 (N675, N621, N671, N602, N600);
AO7 MINUS_284 (N615, N713, N616, N573);
NOR (1,1) MINUS_200 (N715, N627, FSEL_3_);
NAND (1,1) MINUS_194 (N585, N714, N708);
NAND (1,1) MINUS_275 (N703, B_3_, A_3_);
```

```
NAND (1,1) MINUS_185 (N734, N612, N585);
NOT (1,1) MINUS_266 (N717, B_0_);
NOR (1,1) MINUS_176 (N716, N718, N720, N722, N723);
NOT (1,1) MINUS_257 (N698, B_2_);
NOT (1,1) MINUS_167 (N727, N724);
NAND (1,1) MINUS_248 (N591, N593, A_3_);
AND (1,1) MINUS_158 (N693, N604, N727, N589, N699);
AO3 MINUS_239 (N585, N607, N732, N730, N665);
AO2 MINUS_149 (N692, B_1_, N694, N606, N628);
NAND (1,1) MINUS_285 (N645, N719, N715);
NOT (1,1) MINUS_210 (N675, N645);
NOT (1,1) MINUS_120 (N603, N706);
NOR (1,1) MINUS_201 (N714, FSEL_2_, FSEL_3_);
AO2 MINUS_195 (N671, N673, N675, N677, N669);
NAND (1,1) MINUS_276 (F110_3_, N618, N620, N622, N624);
AO1P MINUS_186 (N657, N571, N658, N659, N656);
NOR (1,1) MINUS_267 (N708, N634, FSEL_1_);
NXOR (1,1) MINUS_177 (N657, N711,N600);
NAND (1,1) MINUS_258 (N683, N585, N692);
EON1 MINUS_168 (N640, B_3_, N692, B_3_, N622);
AO7 MINUS_249 (N693, N700, N646, N737);
AO2 MINUS_159 (N681, N683, N613, N680, N646);
AO2 MINUS_220 (N673, N678, A_0_, N582, N676);
NOR (1,1) MINUS_286 (N677, N691, B_0_);
NAND (1,1) MINUS_130 (N607, B_1_, N623);
NOT (1,1) MINUS_211 (N673, N614);
NOT (1,1) MINUS_121 (N599, N578);
NOT (1,1) MINUS_202 (N609, N607);
NAND (1,1) MINUS_196 (N608, B_0_, N691);
NOR (1,1) MINUS_277 (N702, N700, B_2_);
OR (1,1) MINUS_187 (N571, N675, N736, N671);
NOT (1,1) MINUS_268 (N586, N108);
AO7 MINUS_178 (N642, N643, N645, N641);
NOT (1,1) MINUS_259 (N623, A_1_);
NAND (1,1) MINUS_169 (N583, N610, N629);
NAND (1,1) MINUS_230 (N692, N715, N708);
EO1 MINUS_140 (N604, N702, N641, N702, N723);
NAND (1,1) MINUS_221 (N642, N715, N712);
NOT (1,1) MINUS_131 (N701, N697);
NAND (1,1) MINUS_212 (N697, N699, N642);
NOT (1,1) MINUS_122 (N627, FSEL_2_);
NOT (1,1) MINUS_203 (N731, N661);
NAND (1,1) MINUS_197 (N721, N719, N714);
EO1 MINUS_278 (N727, N729, N727, N729, N726);
NAND (1,1) MINUS_188 (N614, B_0_, A_0_);
NAND (1,1) MINUS_269 (N706, N629, N625);
XOR (1,1) MINUS_179 (N664, N662, N598);
```

```
AO4 MINUS_240 (N591, N690, N636, N726, N666);
NXOR (1,1) MINUS_150 (N688, N689, N650);
NAND (1,1) MINUS_231 (N590, N642, N587);
NOR (1,1) MINUS_141 (N711, N597, N705);
AO2 MINUS_222 (B_0_, N672, N677, N674, N670);
NAND (1,1) MINUS_132 (N661, N623, N606);
AO2 MINUS_213 (N692, B_0_, N694, N717, N584);
NOT (1,1) MINUS_123 (N626, FSEL_3_);
NOT (1,1) MINUS_204 (N597, N709);
NOR (1,1) MINUS_198 (N719, FSEL_0_, FSEL_1_);
NOR (1,1) MINUS_279 (N695, N698, N700);
NOR (1,1) MINUS_189 (N577, N579, N581, N584, N655);
NAND (1,1) MINUS_250 (N582, N585, N587, N589);
NAND (1,1) MINUS_160 (N651, N676, N670);
AO2 MINUS_241 (N731, N590, N654, N623, N730);
AO2 MINUS_151 (N679, N586, N577, N108, F110_0_);
NOR (1,1) MINUS_232 (N624, N736, N588, N592, N596);
AO2 MINUS_142 (N692, B_2_, N694, N698, N722);
NAND (1,1) MINUS_223 (N587, N714, N712);
NOT (1,1) MINUS_133 (N735, N639);
XOR (1,1)  MINUS_214 (N637, N690, N591);
NOT (1,1) MINUS_124 (N636, N737);
NOT (1,1) MINUS_205 (N707, N704);
NAND (1,1) MINUS_199 (N576, N691, N717);
NOT (1,1) MINUS_115 (N575, N612);
NAND (1,1) MINUS_260 (N680, N694, N587);
AO2 MINUS_170 (N675, N707, N605, N704, N618);
NOT (1,1) MINUS_251 (N640, N694);
AO2 MINUS_161 (N602, N678, A_1_, N582, N644);
AO1P MINUS_242 (N678, N738, N582, N633, N686);
EO1 MINUS_152 (N651, N724, N651, N724, N650);
AO2 MINUS_233 (N706, A_3_, N721, N580, N588);
AO2 MINUS_143 (N675, N702, N671, N695, N709);
AO2 MINUS_224 (A_0_, N603, N667, N576, N616);
NOR (1,1) MINUS_134 (N604, N605, N575);
XOR (1,1) MINUS_215 (N658, N666, N664);
NOT (1,1) MINUS_125 (N580, A_3_);
NOT (1,1) MINUS_206 (N738, N703);
NOT (1,1) MINUS_116 (N615, N608);
NOT (1,1) MINUS_270 (N634, FSEL_0_);
AO1P MINUS_180 (N635, N571, N637, N639, N632);
NAND (1,1) MINUS_261 (N713, N712, N625);
AND (1,1) MINUS_171 (N736, N719, N625);
NAND (1,1) MINUS_252 (N725, N644, N687);
NAND (1,1) MINUS_162 (N672, N612, N642);
AO2 MINUS_243 (A_1_, N603, N667, N661, N611);
EO1 MINUS_153 (N583, N677, N675, N677, N581);
```

```
AO3 MINUS_234 (N681, N587, N735, N733, N718);
AO2 MINUS_144 (N656, N586, N663, N108, F110_1_);
NAND (1,1) MINUS_225 (N681, N700, N698);
NOT (1,1) MINUS_135 (N643, N613);
XOR(1,1) MINUS_216 (N572, N595, N688);
NAND (1,1) MINUS_126 (N613, B_2_, N700);
NOT (1,1) MINUS_207 (N602, N619);
EO1 MINUS_280 (N724, N725, N724, N725, N598);
NOT (1,1) MINUS_117 (N654, N721);
NAND (1,1) MINUS_190 (N612, N610, N708);
NOR (1,1) MINUS_271 (N629, N631, N634)
EO1 MINUS_181 (N737, N726, N737, N726, N690);
NOT (1,1) MINUS_262 ( N606, B_1_);
AO2 MINUS_172 (N697, N738, N601, N703, N620);
NAND (1,1) MINUS_253 (N578, N580, B_3_);
AO2 MINUS_163 (B_1_, N672, N621, N674, N687);
NOR (1,1) MINUS_244 (N594, A_3_, B_3_);
AO2 MINUS_154 (B_3_, N672, N704, N674, N649);
AO2 MINUS_235 (N632, N586, N716, N108, F110_2_);
AO4 MINUS_145 (N645, N707, N617, N703, N705);
AO7 MINUS_226 (N594, N612, N617, N601);
NAND (1,1) MINUS_136 (N724, N721, N642);
EO1 MINUS_217 (N686, N727, N686, N727, N593);
NOT (1,1) MINUS_127 (N605, N583);
NOT (1,1) MINUS_208 (N671, N617);
NOR (1,1) MINUS_281 (N621, N623, B_1_);
NOT (1,1) MINUS_118 (N653, N576);
AO6 MINUS_191 (N575, N576, N671, N574);
NOT (1,1) MINUS_272 (N631, FSEL_1_);
NOR (1,1) MINUS_182 (N663, N665, N668, N628, N630);
NOR (1,1) MINUS_263 (N712, N631, FSEL_0_);
NAND (1,1) MINUS_173 (N589, N714, N629);
NOT (1,1) MINUS_254 (N700, A_2_);
NAND (1,1) MINUS_164 (N674, N699, N692);
AO3 MINUS_245 (N608, N585, N728, N652, N579);
AO2 MINUS_155 (N574, N614, N701, N673, N655);
AO2 MINUS_236 (N643, N734, N654, N700, N733);
EO1 MINUS_146 (N583, N621, N675, N621, N668);
AO2 MINUS_227 (N585, N599, N713, N578, N596);
NOT (1,1) MINUS_137 (N732, N659);
AO2 MINUS_218 (N608, N680, N576, N683, N696);
AO7 MINUS_128 (N609, N713, N611, N659);
NOT (1,1) MINUS_209 (N728, N573);
NAND (1,1) MINUS_282 (N619, B_1_, A_1_);
NOT (1,1) MINUS_119 (N667, N710);
NOR (1,1) MINUS_192 (N610, N626, FSEL_2_);
NOR (1,1) MINUS_273 (N704, N580, B_3_);
```

```
XOR (1,1) MINUS_183 (N682, N669, N638);
NOT (1,1) MINUS_264 (N691, A_0_);

INPUT M IS N108;
INPUT A[0] IS A_0_;
INPUT A[1] IS A_1_;
INPUT A[2] IS A_2_;
INPUT A[3] IS A_3_;
INPUT B[0] IS B_0_;
INPUT B[1] IS B_1_;
INPUT B[2] IS B_2_;
INPUT B[3] IS B_3_;
INPUT FSEL[0] IS FSEL_0_;
INPUT FSEL[1] IS FSEL_1_;
INPUT FSEL[2] IS FSEL_2_;
INPUT FSEL[3] IS FSEL_3_;
OUTPUT F[0] IS F110_0_;
OUTPUT F[1] IS F110_1_;
OUTPUT F[2] IS F110_2_;
OUTPUT F[3] IS F110_3_;
.
```

MICROPROCESSOR SLICE

VHDL Behavioral Description of Microprocessor Slice

```
library SYNOPSYS;
use synopsys.bv_arithmetic.all;

---------------------------------------------------------------------------

-- PROCESS MODEL OF MICROPROCESSOR SLICE

---------------------------------------------------------------------------

entity MSL is

      port ( IL : in BIT_VECTOR(2 downto 0);
            IH : in BIT_VECTOR(2 downto 0);
            A,B,D,Q : in BIT_VECTOR(3 downto 0);
            CI: in BIT;
            F: out BIT_VECTOR(3 downto 0));
end MSL;

architecture BEH of MSL is


            signal RE, S : BIT_VECTOR(3 downto 0);

begin


process (A,B,D,Q,IL,IH,CI,RE,S)

      begin


-- SELECT THE SOURCE OPERANDS FOR ALU. SELECTED OPERANDS ARE
"RE" AND "S".

s1:    case IL is

       when "000" =>
s2:                 RE <= A;
s3:                 S <= Q;
       when "001" =>
s4:                 RE <= A;
```

```
s5:                S <= B;
      when "010" =>
s6:                RE <= "0000";
s7:                S <= Q;
      when "011" =>
s8:               RE <= "0000";
s9:               S <= B;
     when "100" =>
s10:               RE <= "0000";
s11:                S <= A;
     when "101" =>
s12:                RE <= D;
s13:                S <= A;
     when "110" =>
s14:                RE <= D;
s15:                S <= Q;
     when "111" =>
s16:                 RE <= D;
s17:                 S <= "0000";

   end case;


-- SELECT FUNCTION TO BE PERFORMED BY THE ALU


 s18:    case IH is
      when "000" =>

 s19:                F <= RE + S + ("000" & CI);

      when "001" =>
 s20:                F <=  S - RE - "0001" + ("000" & CI);

      when "010" =>
 s21:               F <= RE - S - "0001" + ("000" & CI);

      when "011" =>

  s22:               F <= RE or S;

      when "100" =>

  s23:                F <= RE and S;

      when "101" =>
```

```
s24:                    F <= not(RE) and S;
        when "110" =>

s25:                    F <= RE xor S;

        when "111" =>

s26:                    F <= not(RE xor S);

      end case;


 end process;

end BEH;
```

Prolog Representation of Microprocessor Slice

```
modelspec :-

assert(fileprefix("msl")),
assert(modelname("MSL")),

assert(datatype(a,bv)), assert(bvlength(a,4)), assert(inputpin(a)),
assert(datatype(b,bv)), assert(bvlength(b,4)), assert(inputpin(b)),
assert(datatype(d,bv)), assert(bvlength(d,4)), assert(inputpin(d)),
assert(datatype(q,bv)), assert(bvlength(q,4)), assert(inputpin(q)),
assert(datatype(ci,bv)), assert(bvlength(ci,1)), assert(inputpin(ci)),
assert(datatype(il,bv)), assert(bvlength(il,3)), assert(inputpin(il)),
assert(datatype(ih,bv)), assert(bvlength(ih,3)), assert(inputpin(ih)),
assert(datatype(f,bv)), assert(bvlength(f,4)), assert(outputpin(f)),

assert(datatype(re,bv)), assert(bvlength(re,4)),
assert(datatype(s,bv)), assert(bvlength(s,4)),


assert(statementtype(s1,case)),
assert(controlexpression(s1,[obj,il])),
assert(subordinaterange(s1,[[bv,"000"]],[s2,s3])),
assert(subordinaterange(s1,[[bv,"001"]],[s4,s5])),
assert(subordinaterange(s1,[[bv,"010"]],[s6,s7])),
assert(subordinaterange(s1,[[bv,"011"]],[s8,s9])),
assert(subordinaterange(s1,[[bv,"100"]],[s10,s11])),
assert(subordinaterange(s1,[[bv,"101"]],[s12,s13])),
assert(subordinaterange(s1,[[bv,"110"]],[s14,s15])),
```

```
assert(subordinaterange(s1,[[bv,"111"]],[s16,s17])),

assert(statementtype(s2,assignment)),
assert(destinationobject(s2,re)),
assert(sourceexpression(s2,[obj,a])),

assert(statementtype(s3,assignment)),
assert(destinationobject(s3,s)),
assert(sourceexpression(s3,[obj,q])),

assert(statementtype(s4,assignment)),
assert(destinationobject(s4,re)),
assert(sourceexpression(s4,[obj,a])),

assert(statementtype(s5,assignment)),
assert(destinationobject(s5,s)),
assert(sourceexpression(s5,[obj,b])),

assert(statementtype(s6,assignment)),
assert(destinationobject(s6,re)),
assert(sourceexpression(s6,[lit,[bv,"0000"]])),

assert(statementtype(s7,assignment)),
assert(destinationobject(s7,s)),
assert(sourceexpression(s7,[obj,q])),

assert(statementtype(s8,assignment)),
assert(destinationobject(s8,re)),
assert(sourceexpression(s8,[lit,[bv,"0000"]])),

assert(statementtype(s9,assignment)),
assert(destinationobject(s9,s)),
assert(sourceexpression(s9,[obj,b])),

assert(statementtype(s10,assignment)),
assert(destinationobject(s10,re)),
assert(sourceexpression(s10,[lit,[bv,"0000"]])),

assert(statementtype(s11,assignment)),
assert(destinationobject(s11,s)),
assert(sourceexpression(s11,[obj,a])),

assert(statementtype(s12,assignment)),
assert(destinationobject(s12,re)),
assert(sourceexpression(s12,[obj,d])),

assert(statementtype(s13,assignment)),
```

```
assert(destinationobject(s13,s)),
assert(sourceexpression(s13,[obj,a])),

assert(statementtype(s14,assignment)),
assert(destinationobject(s14,re)),
assert(sourceexpression(s14,[obj,d])),

assert(statementtype(s15,assignment)),
assert(destinationobject(s15,s)),
assert(sourceexpression(s15,[obj,q])),

assert(statementtype(s16,assignment)),
assert(destinationobject(s16,re)),
assert(sourceexpression(s16,[obj,d])),

assert(statementtype(s17,assignment)),
assert(destinationobject(s17,s)),
assert(sourceexpression(s17,[lit,[bv,"0000"]])),

assert(statementtype(s18,case)),
assert(controlexpression(s18,[obj,ih])),
assert(subordinaterange(s18,[[bv,"000"]],[s19])),
assert(subordinaterange(s18,[[bv,"001"]],[s20])),
assert(subordinaterange(s18,[[bv,"010"]],[s21])),
assert(subordinaterange(s18,[[bv,"011"]],[s22])),
assert(subordinaterange(s18,[[bv,"100"]],[s23])),
assert(subordinaterange(s18,[[bv,"101"]],[s24])),
assert(subordinaterange(s18,[[bv,"110"]],[s25])),
assert(subordinaterange(s18,[[bv,"111"]],[s26])),

assert(statementtype(s19,assignment)),
assert(destinationobject(s19,f)),
assert(sourceexpression(s19,[bvadd,[bvadd,[obj,re],[obj,s]],[bvcat,[lit,[bv,"000"]],
[obj,ci]]])),

assert(statementtype(s20,assignment)),
assert(destinationobject(s20,f)),
assert(sourceexpression(s20,[bvadd,[bvsub,[obj,s],[bvadd,[obj,re],[lit,[bv,"0001"]]]],
[bvcat,[lit,[bv,"000"]],[obj,ci]]])),

assert(statementtype(s21,assignment)),
assert(destinationobject(s21,f)),
assert(sourceexpression(s21,[bvadd,[bvsub,[obj,re],[bvadd,[obj,s],[lit,[bv,"0001"]]]],
[bvcat,[lit,[bv,"000"]],[obj,ci]]])),

assert(statementtype(s22,assignment)),
assert(destinationobject(s22,f)),
```

```
assert(sourceexpression(s22,[bvor,[obj,re],[obj,s]])),

assert(statementtype(s23,assignment)),
assert(destinationobject(s23,f)),
assert(sourceexpression(s23,[bvand,[obj,re],[obj,s]])),

assert(statementtype(s24,assignment)),
assert(destinationobject(s24,f)),
assert(sourceexpression(s24,[bvand,[bvnot,[obj,re]],[obj,s]])),

assert(statementtype(s25,assignment)),
assert(destinationobject(s25,f)),
assert(sourceexpression(s25,[bvxor,[obj,re],[obj,s]])),

assert(statementtype(s26,assignment)),
assert(destinationobject(s26,f)),
assert(sourceexpression(s26,[bvnot,[bvxor,[obj,re],[obj,s]]])).
```

Gate Level GHDL Description of Microprocessor Slice

```
** GHDL file generated from EDIF by GNXOR(1,1)RAD Translator Version 0.8
**=============================================================
**
** Translated on :           Fri Dec 18 16:36:38 1992
**
** EDIF Source filename :      SYNOPSYS_EDIF
** EDIF Version :             2 0 0
**
*******************************************************************
** Circuit 'MSL' translated from edif cell 'MSL'
CCT BIN  MSL (CI, IL[2], IL[1], IL[0], IH[2],
IH[1], IH[0], A[3], A[2], A[1], A[0], B[3], B[2], B[1], B[0], D[3],
D[2], D[1], D[0], Q[3], Q[2], Q[1], Q[0], F[3], F[2], F[1], F[0])

NOT(1,1) PLUS_150 ( N321,CI);
NXOR(1,1) PLUS_92 ( N391,N333, N374);
AO2 PLUS_141 (N370, N332, N369, N412, N352);
AO3 PLUS_83 (N403, N348, N404, N406, F_1_);
AO2 PLUS_132 (N337, A_1_, D_1_, N353, N396);
OR(1,1) PLUS_74 (N318,N378, N320, N321);
NAND(1,1) PLUS_123 ( N370,N371, N348);
EO1 PLUS_65 ( N328, N330, N328, N330,N402);
NAND(1,1) PLUS_114 (N405,Q_0_, N356);
NOT(1,1) PLUS_56 (N408,N400);
AO2 PLUS_105 (A_0_, N341, B_0_, N339, N423);
XOR(1,1) PLUS_160 ( N374,N349, N415);
```

```
AO2 PLUS_93 (N334, N335, N336, N338, N333);
NAND(1,1) PLUS_84 ( N367,N368, N348);
AO2 PLUS_151 (N364, N392, N385, N355, N368);
AO2 PLUS_142 (N382, N389, N388, N373, N395);
EO1 PLUS_75 (N399, N323, N399, N323, N322);
AO4 PLUS_66 (N396, N397, N398, N403, N394);
NAND(1,1) PLUS_133 (N351, IL_2_, N340, IL_1_);
AO4 PLUS_124 (N346, N398, N397, N412, N415);
NOT(1,1) PLUS_57 ( N403, N420);
AO2 PLUS_115 (N337, A_2_, D_2_, N353, N354);
XOR(1,1) PLUS_106 ( N350, N386, N393);
AO2 PLUS_94 (A_3_, N341, B_3_, N339, N347);
AO2 PLUS_161 (N421, N365, N419, N396, N406);
NOR(1,1) PLUS_85 (N323,N390, N377);
AO2 PLUS_152 (N408, N390, N344, N416, N407);
NOR(1,1) PLUS_76 ( N337,IL_2_, IL_1_);
NOR(1,1) PLUS_143 ( N383,N375, IH_1_, N376);
EO1 PLUS_67 ( N394, N323, N394, N323,N326);
AO2 PLUS_134 (N417, N408, N372, N400, N413);
NOT(1,1) PLUS_58 ( N365, N396);
NAND(1,1) PLUS_125 (N420,N424, N361);
NAND(1,1) PLUS_116 ( N417,N418, N348);
NAND(1,1) PLUS_107 (N358, Q_2_, N356);
NOR(1,1) PLUS_95 (N381, N383, N385);
AO2 PLUS_86 (N387, N388, N389, N391, N386);
NOR(1,1) PLUS_77 ( N327, N325, N402);
AO2 PLUS_68 (N344, N345, N390, N392, N343);
NOT(1,1) PLUS_153 ( N375, IH_0_);
NOT(1,1) PLUS_144 (N388, N323);
NOT(1,1) PLUS_59 ( N344,N397);
AO2 PLUS_135 (N337, A_3_, D_3_, N353, N412);
NAND(1,1) PLUS_126 ( N362, IH_1_, IH_0_, IH_2_);
AO2 PLUS_117 (N337, A_0_, D_0_, N353, N400);
EO1 PLUS_96 ( N382, N327, N382, N327, N373);
NAND(1,1) PLUS_108 (N421, N422, N348);
NAND(1,1) PLUS_87 (N364, N366, N362);
NOR(1,1) PLUS_78 ( N319, N320, N321);
EO1 PLUS_69 ( N389, N325, N323, N325,N320);
NXOR(1,1) PLUS_154 ( N330, N363, N326);
AO2 PLUS_145 (N364, N331, N385, N346, N371);
AO4 PLUS_136 (N354, N398, N355, N397, N338);
NAND(1,1) PLUS_127 ( N331, N347, N414);
AO2 PLUS_97 (A_2_, N341, B_2_, N339, N425);
NAND(1,1) PLUS_88 (N398, N375, N376, IH_1_);
AO7 PLUS_118 (N342, N340, N351, N353);
XOR(1,1) PLUS_109 (N387, N380, N391);
AO3 PLUS_79 (N409, N348, N411, N413, F_0_);
```

```
NOT(1,1) PLUS_155 ( N376, IH_2_);
AO4 PLUS_146 (N363, N326, N328, N330, N335);
AO2 PLUS_137 (N364, N416, N385, N409, N418);
XOR(1,1) PLUS_98 ( N404, N378, N319);
NAND(1,1) PLUS_128 ( N361, Q_1_, N356);
NOR(1,1) PLUS_89 (N380, N382, N384);
AO2 PLUS_119 (N381, N416, N362, N409, N372);
NXOR(1,1) PLUS_156 ( N324, N407, N322);
NOT(1,1) PLUS_147 ( N390, N398);
XOR(1,1) PLUS_99 (N382, N335, N334);
EO1 PLUS_138 ( N402, N389, N323, N401,N378);
NOR(1,1) PLUS_129 (N377, IH_1_, IH_2_, N375);
NOT(1,1) PLUS_157 (N342, IL_2_);
AO2 PLUS_148 (N367, N345, N359, N354, N360);
NOR(1,1) PLUS_139 (N397, N377, N389);
AO3 PLUS_50 (N346, N348, N350, N352, F_3_);
XOR(1,1) PLUS_158 (N334, N338, N336);
AO4 PLUS_149 (N407, N322, N323, N324, N410);
OR(1,1) PLUS_100 (N366, IH_1_, IH_0_, N376);
NOT(1,1) PLUS_60 (N355, N392);
AO3 PLUS_51 (N355, N348, N357, N360, F_2_);
NOT(1,1) PLUS_159 (N340, IL_0_);
NAND(1,1) PLUS_110 (N416, N423, N405);
AO2 PLUS_101 (A_1_, N341, B_1_, N339, N424);
NOR(1,1) PLUS_70 (N341, N342, IL_1_);
NOT(1,1) PLUS_61 (N345, N354);
NOT(1,1) PLUS_52 (N346, N331);
AO7 PLUS_120 (N389, N388, N379, N411);
AO2 PLUS_80 (N381, N331, N362, N346, N369);
AO7 PLUS_111 (IL_0_, IL_2_, N351, N356);
EO1 PLUS_102 (N321, N320, N321, N320, N379);
NAND(1,1) PLUS_71 (N348, IH_0_, N376, IH_1_);
AO2 PLUS_62 (N331, N344, N390, N332, N329);
NOT(1,1) PLUS_53 (N332, N412);
AO2 PLUS_130 (N364, N420, N385, N403, N422);
EO1 PLUS_90 ( N323, N329, N323, N329, N349);
AO2 PLUS_121 (N365, N390, N344, N420, N363);
NOR(1,1) PLUS_81 (N389, IH_2_, IH_0_, IH_1_);
AO2 PLUS_112 (N381, N420, N362, N403, N419);
EO1 PLUS_72 ( N323, N324, N323, N324,N325);
EO1 PLUS_103 ( N343, N323, N343, N323,N336);
AO6 PLUS_63 (N402, N325, N327, N401);
NOT(1,1) PLUS_54 (N328, N410);
NOT(1,1) PLUS_140 (N384, N327);
AO2 PLUS_91 (N381, N392, N362, N355, N359);
NOR(1,1) PLUS_82 (N393, N395, N318);
NAND(1,1) PLUS_131 (N414, Q_3_, N356);
```

```
AND(1,1) PLUS_122 (N385, IH_1_, N375, IH_2_);
NOR(1,1) PLUS_73 (N339, N340, IL_2_);
AO4 PLUS_64 (N400, N397, N398, N409, N399);
NOR(1,1) PLUS_113 (N357, N395, N318);
NAND(1,1) PLUS_104 (N392, N425, N358);
NOT(1,1) PLUS_55 (N409, N416);

INPUT CI;
INPUT IL[2] IS IL_2_;
INPUT IL[1] IS IL_1_;
INPUT IL[0] IS IL_0_;
INPUT IH[2] IS IH_2_;
INPUT IH[1] IS IH_1_;
INPUT IH[0] IS IH_0_;
INPUT A[3] IS A_3_;
INPUT A[2] IS A_2_;
INPUT A[1] IS A_1_;
INPUT A[0] IS A_0_;
INPUT B[3] IS B_3_;
INPUT B[2] IS B_2_;
INPUT B[1] IS B_1_;
INPUT B[0] IS B_0_;
INPUT D[3] IS D_3_;
INPUT D[2] IS D_2_;
INPUT D[1] IS D_1_;
INPUT D[0] IS D_0_;
INPUT Q[3] IS Q_3_;
INPUT Q[2] IS Q_2_;
INPUT Q[1] IS Q_1_;
INPUT Q[0] IS Q_0_;
OUTPUT F[3] IS F_3_;
OUTPUT F[2] IS F_2_;
OUTPUT F[1] IS F_1_;
OUTPUT F[0] IS F_0_;

ENDCIRCUIT
```

Vita

Gunjeetsingh D. Baweja was born December 22, 1967. He graduated from Rosary High School in November 1985. He won the National Merit Scholarship for his performance at High School. He entered Maharaja Sayajirao University at Baroda, India, in January 1986, and received a Bachelor of Engineering degree in Electronics Engineering, graduating with *Distinction* in January 1990. He attended graduate school at Virginia Polytechnic Institute and State University (VPI & SU) from August 1991 to March 1993, receiving a Master's of Science degree in Electrical Engineering in March 1993. After graduating from VPI & SU, Gunjeet began working with Intel Corporation at Folsom, California.