High Level Design and Test Methodologies with VHDL

by

Ruomei Wang

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

Master of Science

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba

ISBN 0-612-16366-0

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

COPYRIGHT PERMISSION

HIGH LEVEL DESIGN AND TEST METHODOLOGIES WITH VHDL

BY

RUOMEI WANG

A Thesis/Practicum submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or other means, in whole or in part, at the request of other institutions or individuals for the purpose of scholarly research.

ABSTRACT

The growing complexity and density of integrated circuits requires that good design and test methodologies be utilized throughout the design process. This thesis presents high level design methodologies typically involved in the design process, from behavioral to gate-level structural descriptions. Test and design for testability techniques are reviewed, and experience with the Synopsys test synthesis tools is presented. The diverse styles and advantages of testbenches are also discussed. A novel approach for doing testbench simulation on RTL level and gate-level circuits and comparison of the outputs from both versions of the design is demonstrated. As a real design, an elevator controller is developed utilizing these methodologies, and the simulation results are presented. The Synopsys CAD tools are involved in each design and test stage, which greatly relieves the designer from tedious and error prone tasks.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Dr. Robert McLeod for his intellectual advice throughout my study at the University of Manitoba. Even though he is very busy at the University and TRLabs, he still made time to support and guide my study. I would also like to thank some other students in the VLSI group, Guy Jonatschick and Budi Rahardjo for their helpful discussions and suggestions. Special thanks go to Ken Ferens for his invaluable assistance. Also, Kevin Zong's encouragement is gratefully acknowledged.

Heartfelt appreciation goes to my parents. Even though they are very far away from me, their understanding and love made me overcome all the difficulties.

CMC and Micronet who provided financial support are highly appreciated.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 Motivation

Conventional electronic design begins by manually entering the logic diagram, followed by timing verification at the gate-level. Only after this is done can comprehensive functional verification be performed. As a result, any functional specification errors or deficiencies will not be detected and corrected until a later stage in the design process. This means that any modification in the logic diagram requires long iteration loops. This iterative design process takes too long and results in low productivity. Because of the fast growth of VLSI designs, there is a need to improve designer's productivity and shorten time-to-market.

High level top-down design can significantly increase the designer's productivity. High level design flow methodologies are currently an active area of research. CAD and CASE tools play important roles in the design flow: synthesizing behavioral code to gate-level implementations, high-level simulation, gate-level simulation, insertion of scan chains, generation of test patterns, and board-level simulation. Automation of these tasks saves designer's time and energy. CAD and CASE tools continuously need to be improved to meet the increasing demands of IC designers.

According to many designers' experience, the majority of time in designing a system is spent on capturing and verifying the functional design. CAD tools can carry out most of the remaining tasks. Thus the design methodology focuses on capturing and verifying the design. Time is spent on writing behavioral code and a good testbench, modifying the behavioral code, until the behavior is satisfactory. As a result, verification at a high level has significant meaning. Functional testbench generation and application methodologies need to be developed for high level verification.

1.2 Levels of Abstraction

Any complex system may be broken down into gates and memory elements by successively subdividing the system in a hierarchical manner. This subdivision may be done manually or by automated techniques. To do this, integrated circuits may be described at different abstraction levels. There are three design domains that represent a system: behavioral, structural, and physical domains. Behavioral representation specifies what a system does, the structural domain specifies how entities are connected together, and the physical domain binds the system in space or to silicon. Each design domain may be specified at a variety of abstraction levels. From highest to lowest, these include: system, RTL (Register-Transfer-Level), logic and circuit level. The Y-chart [20] of Figure 1.1 illustrates the tripartite representation of design.

Design Strategies

A good VLSI design system should be capable of providing consistent descriptions in all three domains and at all abstraction levels. The design parameters may be described in terms of performance, size of die, time to design, ease of test generation and testability. The design process involves trade-offs to achieve good results for all of the parameters. There are some

classical approaches to reducing the complexity of IC design including: hierarchy, regularity, modularity, and locality.

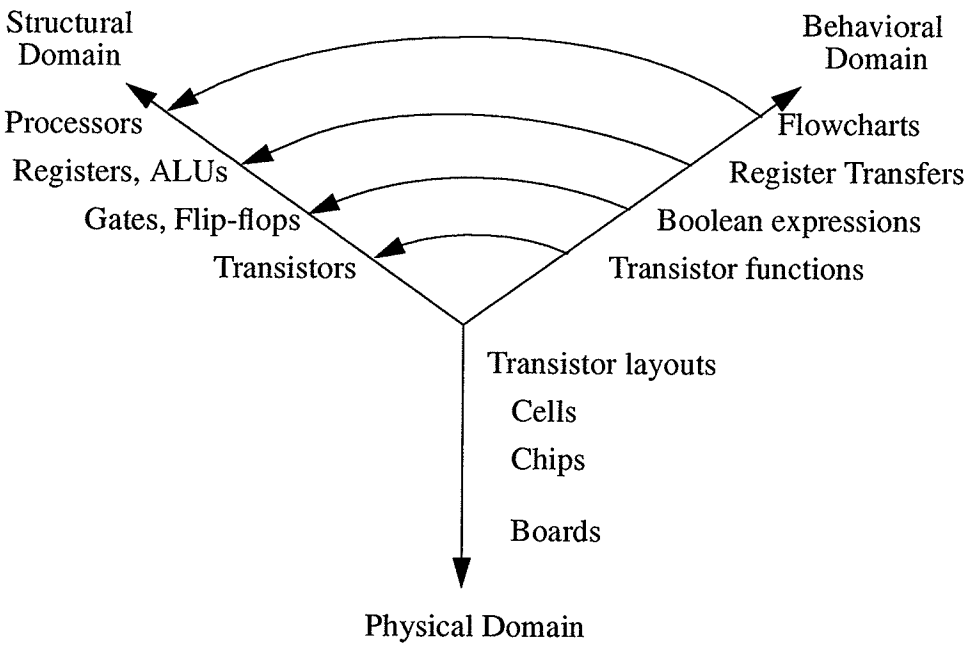


Fig. 1.1: The Y-chart

1.3 Introduction to High Level Design Flow

Figure 1.2 shows a high level design methodology [28]. In the Design Entry stage, design description consists of schematic diagrams and Register-Transfer-Level HDL code (such as Verilog or VHDL) which define the behavior of the designs. The validation phase is used for simulating the source code and making sure the RTL code behaves in the way it is intended to. This phase involves interactive simulation and source code debugging. The next phase is implementation where logic and test synthesis tools are used to translate RTL code to a gate-level schematic (or netlist). Test synthesis performs scan insertion in addition to logic synthesis. In the verification phase, the HDL testbench is designed for testing against the RTL description, and also against the gate-level implementation. The HDL testbench and test vectors are used to automatically exercise the circuit during simulation. In this phase, a wide

range of struc-

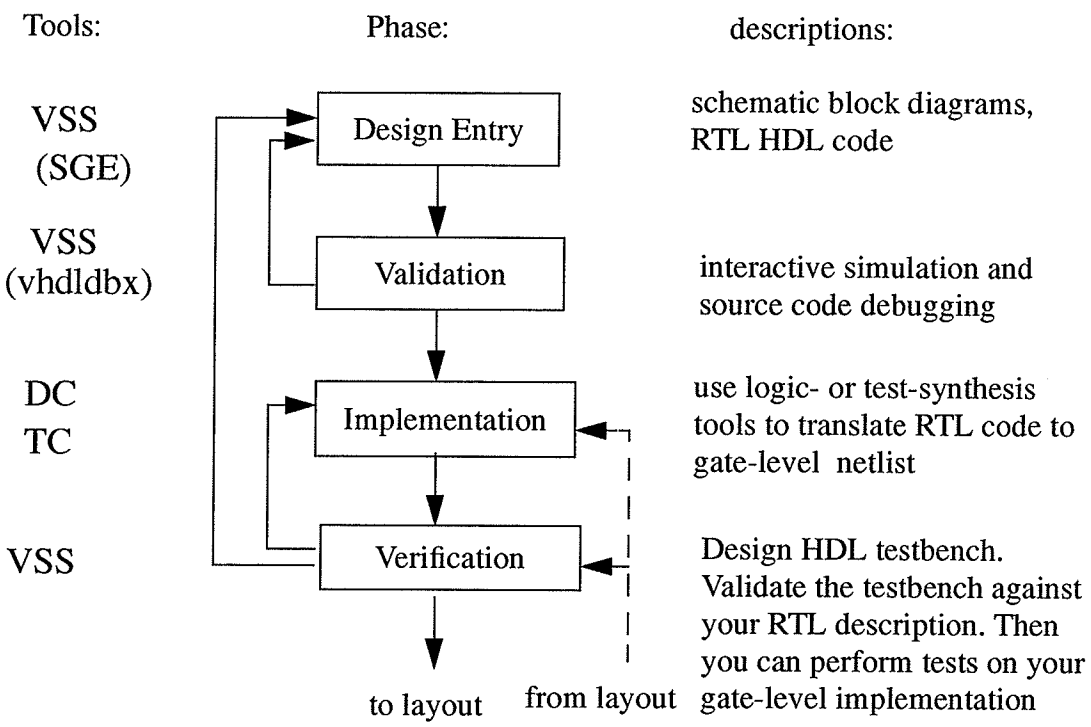


Fig. 1.2: High Level Design Flow

tural and behavioral alternatives can be explored, and the best trade-offs among speed, size, and testability are chosen. If there are malfunctions or errors detected in the verification phase, the design is returned to the design entry phase, with subsequent modification of the code or gate-level structure. The Synopsys tools help in each phase and are indicated in the design flow of Figure 1.2. *VSS* is the VHDL System Simulator, *DC* is the Design Compiler, *SGE* is the Synopsys Graphical Environment, and *TC* is the Test Compiler.

1.4 Introduction to HDL Testbenches and Their Applications

This thesis proposed a complete scheme for HDL testbench methodologies. The scheme is shown in Figure 1.3.

HDL testbenches basically divide into two types: structural and behavioral testbenches.

A structural testbench is generated by an ATPG (Automatic Test Pattern Generator) using DFT (Design-for-Testability) techniques developed from the structural model (gate-level schematic or netlist). This is simulated on the gate-level design, detecting gate-level faults, and generating a fault coverage report. This work can be done using CAD tools such as the Synopsys Test Compiler and TestSim. A complete set of test vectors is generated by the Synopsys Test Compiler. This set of test vectors can be simulated at the module, chip and system level.

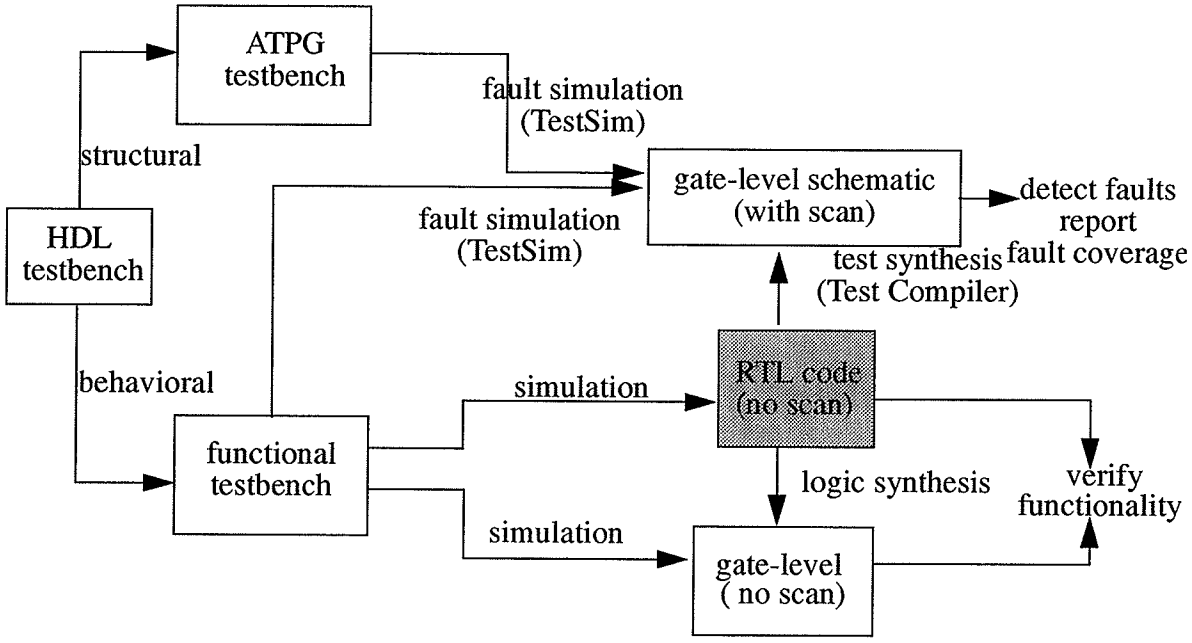


Fig. 1.3: HDL Testbenches and Their Applications

A behavioral testbench is manually generated. It is developed from the behavioral model (RTL code, State Machine, BDD, Petri-Nets, etc.), and is used to verify the design's compliance to its functionality, i.e. to check its behavior. A behavioral testbench is also called a functional testbench. If there are Hard-To-Detect (HTD) faults in the gate-level design,

behavioral testbenches can help detect them. Both ATPG and functional test vectors are used for gate-level simulation (this is called a multi-pass, multi-style test process, which is a feature of the Synopsys test synthesis tools). In this way, higher fault coverage is achieved. Before doing test synthesis, a functional testbench (without scan) is written for the RTL HDL code to verify its functionality. It can also be used at the gate-level, which is one of the HDL testbench's advantages, mixed level simulation. Synopsys SGE can generate the functional HDL testbench template, while the user writes the behavioral part by hand.

1.5 Organization of the Thesis

In Chapter 2, test and design for testability techniques are reviewed. Experiences using the Synopsys test synthesis tools are then presented.

Chapter 3 discusses all the methodologies involved in the entire high level design process, such as functional partitioning, schematic generation, VHDL design entry, synthesis and optimization.

In Chapter 4, functional testbench styles and generation methodologies are presented, and how such a testbench works on the RTL and gate level descriptions is demonstrated. A couple of behavioral test pattern generation algorithms are also given. Testbench simulation results for the elevator controller design are also demonstrated in this chapter.

Chapter 5 presents a summary and general conclusions, as well as future work.

Design schematics, behavioral descriptions, source code and experimental results using test synthesis tools are shown in APPENDIX A and APPENDIX B.

CHAPTER 2

TEST AND DESIGN FOR TESTABILITY OVERVIEW

2.1 Introduction

The purpose of testing is to detect faults and locate the faults in the design. As such, testing is an important issue in VLSI design. In this chapter, some fundamental concepts in testing, such as ATPG, testability, DFT, and scan-based design will be presented. Testing techniques have been developed to greatly improve test efficiency and flexibility. These techniques will be discussed. The test synthesis approach is presented. The results for several examples using these test methodologies and CAD tools will be given. Experience with physical testing of a chip (DM74181) using the VXI bus test system and IC test head is also presented.

2.2 Testing Overview

Tests

Design quality implies three aspects: usability, reliability and efficiency [20]. Usability refers to the functionality of design and what it can do. Reliability means how long the design functionality can last and efficiency asks the question "Is it worthwhile using it?". There are always some faults that occur in a design, due to the VLSI processing, packaging and assembly, and/or specification ambiguity. Testing helps detect malfunctions and errors

in the system thus ensuring that it is fault free. Therefore, testing ensures a high quality design.

Tests may be divided into two categories: functional tests and manufacturing tests [20]. A functionality test verifies that the system performs as intended. It is used in the early design stages. In a high level design cycle, this is called validation. It proves that the design performs according to some specifications. The specification might be a verbal description, a plain-language textual specification, or a high-level computer language such as C, FORTRAN, VHDL, SDL, tables of inputs and required outputs, or high level languages such as statecharts. Functional test pattern generation methodologies are discussed in Chapter 4. In this chapter, test refers to manufacturing test.

A manufacturing test verifies internal gates, detects manufacturing defects which might come from fabrication or ware out. Normally, three kinds of fault models are defined: the stuck-at fault, stuck-open fault and line-delay fault. The stuck-at fault is the most common in industry and is the only gate-level fault model examined in this thesis.

Testability

The quality of testing is evaluated by testability. According to R.G. Bennetts [27], "Testability is the ability to generate, evaluate, and apply tests to satisfy a number of predefined test objectives (e.g. fault coverage, fault isolation, runtime, time-to-profit) subject to the two fundamental constraints of time and money".

There are two attributes used to evaluate testability. They are controllability and observability. Controllability is the ability to drive an internal node to a specified logic value by setting primary inputs to specific values. Observability is the ability to predict the response of an

internal node and propagate the response to primary outputs [28]. A design is testable when it can be put into a known initial state, and the internal nodes are controllable and observable.

2.3 Design-for-Testability (DFT)

Traditionally, design and test were separate processes. In today's design flow, test and design are merged early in the design cycle. This approach improves controllability and observability. The incorporation of test hardware into a circuit design is called Design-for-Testability.

Generally speaking, there are two approaches to DFT: Ad-Hoc and Structured. Ad-Hoc DFT does not make big changes to the design. It includes: minimizing redundant logic, minimizing asynchronous logic, isolating clocks from the logic, and adding internal control and observation points. Structured DFT is a more systematic and automated approach to enhance design testability and thus is a more popular one. The purpose of structured DFT is to increase the controllability and observability of a circuit. There are various methods to reach this goal. Scan design is the most common method. It changes the internal sequential circuitry of the design. Another method is boundary scan which modifies the chip circuitry and increases board level testability. DFT techniques lower test cost, improve product quality, and reduce time-to-market.

2.4 Scan Design Methodology

There are two main types of scan circuitry: internal scan and boundary scan. Internal scan (also referred as scan design) can be full scan or partial scan. Internal scan modifies circuitry

within the original design itself and increases chip level testability. Boundary scan adds circuitry around the periphery of the design and increases board level testability.

2.4.1 Internal Scan (Scan Design)

A sequential circuit is much more difficult to test, compared with a combinational circuit. Scan design makes a sequential circuit behave like a combinational circuit when the design is in test mode. To achieve this goal, sequential elements are replaced by scannable sequential cell (scan cells), then scan cells are stitches together into scan chains. These serially-connected scan cells can shift data in and out during the test process [28].

Scan design methodology is depicted in Figure 2.1. The original design contains combinational and sequential portions. Before scan circuitry is added, the design has three inputs: A, B, and C, and two outputs, OUT1, OUT2. This "before scan" circuit is difficult to test because it is hard to initialize the design to a known state. Thus it is difficult to control and observe its behavior by using the primary inputs and outputs. After adding scan circuitry (shown by -----lines), there are two additional inputs, sc_in (scan input) and sc_en (scan enable), and one additional output, sc_out (scan output). The original sequential elements are replaced by scan elements, to allow data to be read in from sc_in and shifted out to sc_out when sc_en line is enabled. The se_out line from each flip-flop feeds into the next flip-flop's se_in, and sc_en is connected in parallel to each sequential element. This permits serial I/O to the scan elements which requires minimal pin out overhead.

2.4.2 Full Scan and Partial Scan

In full scan design, all storage elements are replaced by their scannable equivalents and stitched into scan chains. Whereas in partial scan design, only the storage elements of interest are replaced by their scannable equivalents and stitched into scan chains [28].

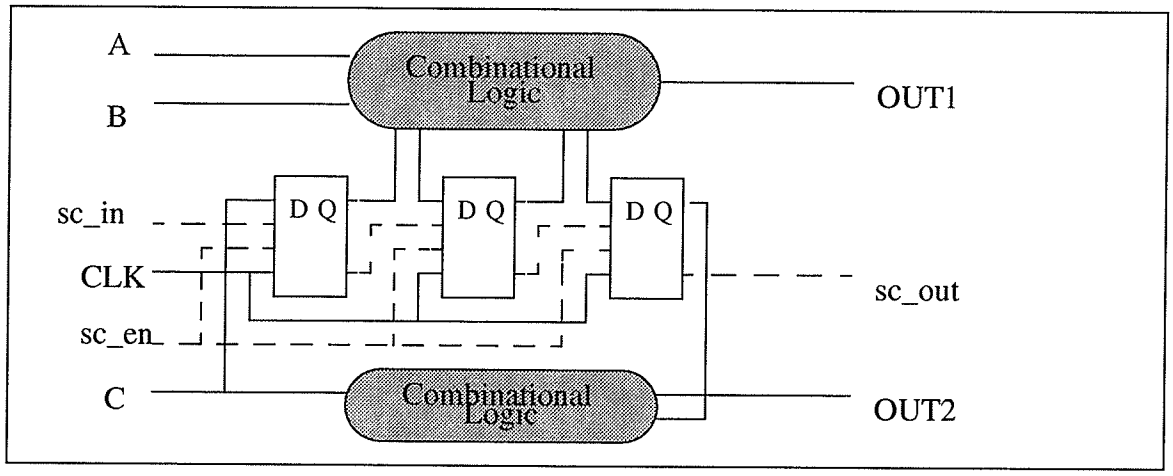


Fig. 2.1: Design Before and After Adding Scan Circuitry

Figure 2.2 gives a symbolic representation of a full scan and partial scan design. The rounded boxes represent combinational portions. Rectangular boxes represent sequential elements and black rectangles are sequential cells that have been converted to scan cells. The line connecting them is the scan chain.

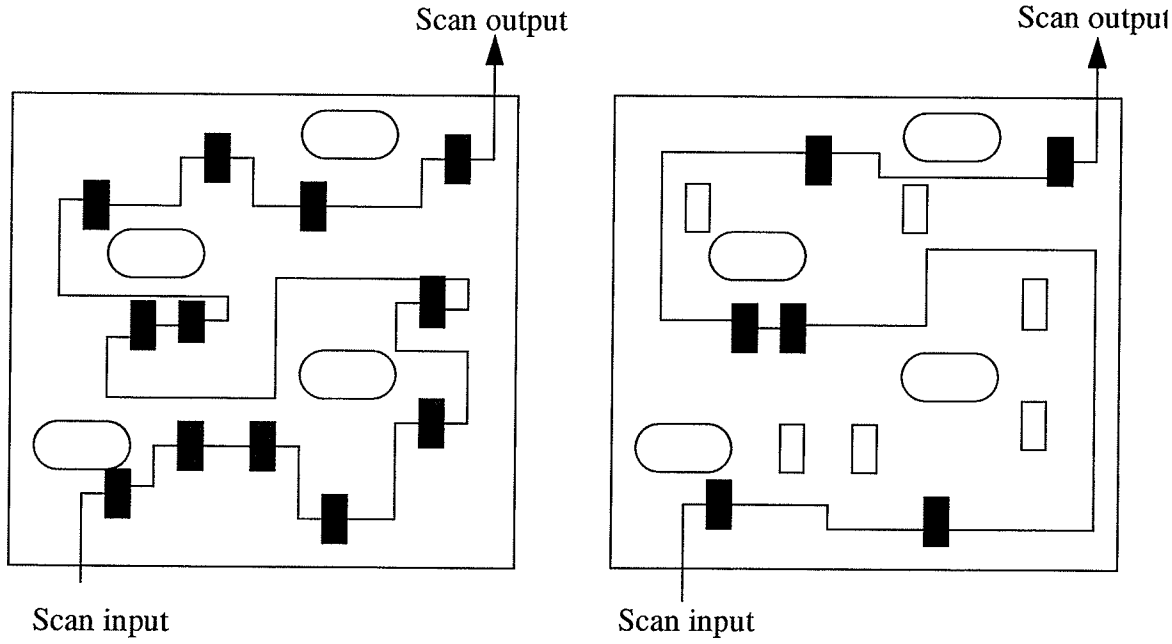


Fig. 2.2: Full Scan and Partial Scan Symbolic Representation

There are trade-offs among full scan, partial scan and non-scan designs as shown in Figure 2.3. Full scan requires a significant amount of area overhead because all the sequential cells are replaced by scannable cells. However it requires less test generation effort and since the scan insertion process is automated, it ensures high efficiency, high fault coverage and high quality. Partial scan is more flexible with respect to area overhead and fault coverage. Partial and non-scan designs require much less area overhead but far more test generation effort.

Full scan employs a combinational ATPG algorithm, while partial scan utilizes sequential ATPG. The Synopsys Test Compiler only has combinational ATPG algorithm, while Test Compiler Plus has both combinational and sequential ATPG algorithms [28].

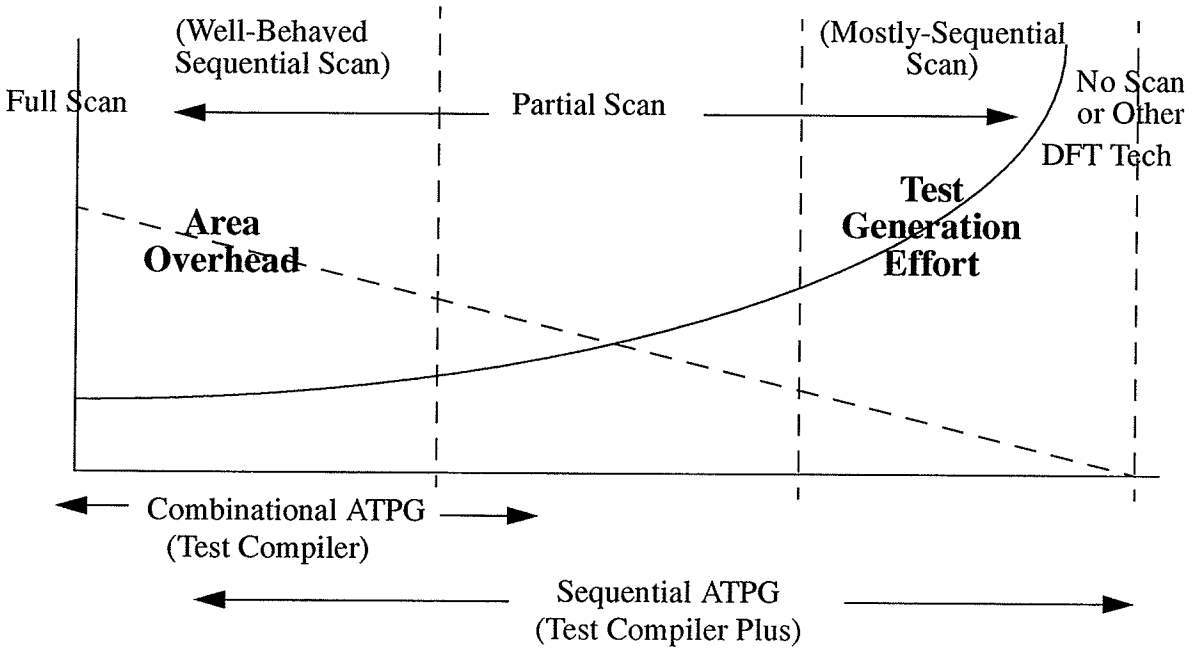


Fig. 2.3: Full, Partial, and Non-Scan Trade-offs

There are some limitations to the scan design technique. For example, scan design techniques do not work well with certain circuit structures, including:

- Large, nonscan macro functions (e.g., microprocessor cores).
- Compiled cells (e.g., RAMs and ALUs).
- Analog circuitry.

2.4.3 Boundary Scan

Boundary scan improves board level testing by providing an interface through which data can shift between circuits on the board, thereby, making each net on a board accessible. Boundary scan is also referred to as JTAG (Joint Test Action Group) and IEEE standard 1149.1 testing. Figure 2.4 shows how boundary scan works.

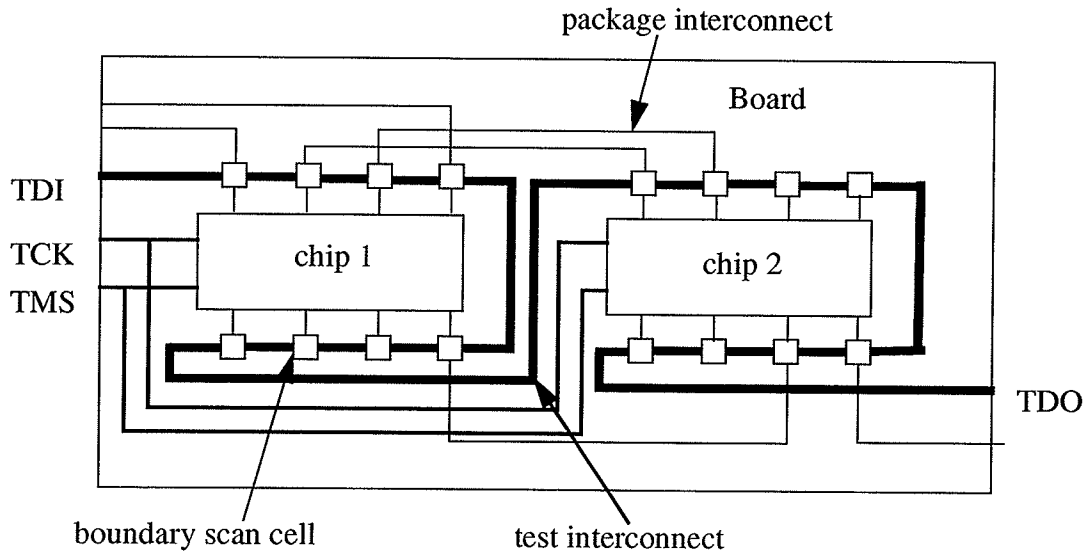


Fig. 2.4: Boundary Scan

The input and output ports of each chip are stitched together into a scan path [13]. The scan path begins at TDI (test data in) and ends at TDO (test data out). The TDO line of one chip is connected to TDI of the next chip. TCK (test clock) and TMS (test mode select) are inputs connected to every boundary scan device. The scan path connects all the boundary scan devices on the board. This configuration allows board interconnection test, snapshots of nor-

mal system data, and testing of individual chips.

The IEEE standard 1149.1 Boundary Scan interface consists of:

- TAP (Test Access Port): At least four signals TMS, TCK, TDI, TDO compose the test bus, TRST (asynchronous test reset signal) is optional.
- TAP Controller: Controls the operation of the instruction and test data registers.
- Instruction Register: At least two bits long, it decodes at least three instructions which are BYPASS (allows testing a specific chip in the scan chain without shifting through the SR stages in all chips), EXTEST (allows testing of off-chip circuitry), SAMPLE/ PRELOAD (places the boundary-scan register in the DR chain, and samples or preloads the chip I/Os).
- Boundary Scan Register: A shift register consisting of the connection of boundary scan cells that can parallel/serially load and unload the input and output data of the circuit.
- Bypass register: shortens the serial path between TDI and TDO when there is no requirement to test the current device. In this manner data bypasses the current chip and are shifted to other chips more directly.
- Test Data Register: includes the optional device identification register and data-specific registers.

2.5 Design Flow Using the Synopsys Test Synthesis Tools

2.5.1 Test Synthesis

Test Synthesis is a technique which incorporates test and synthesis into one [14]. Traditionally, DFT circuitry is added manually after logic synthesis, however this is very time-consuming. Test synthesis approaches shorten design time and are therefore an increasingly

important part of circuit design.

There are two classes of test synthesis tools available: gate-level and RTL test synthesis. In gate-level test synthesis, the gate-level model is used to drive the test synthesis process. First, logic synthesis of RTL code is performed, then test circuitry is added and test vectors are generated for the synthesized model. In RTL synthesis, designers add the test circuitry in the RTL code, then conduct logic synthesis for this RTL code with the test circuitry. Gate-level test synthesis is more mature and more suitable for full scan design. The Synopsys Test Compiler family and Mentor Graphics FastScan use gate-level test synthesis techniques. RTL test synthesis is newer and more suitable for a boundary scan methodology.

2.5.2 Design Flow Using the Synopsys Test Compiler Tool

A typical DFT design flow using the Synopsys test compiler is described below. The features of the Test Compiler tools are also illustrated within the flow. The Test Compiler is used in three hierarchical stages: module level, chip level, and board level. A complex design is partitioned into subdesigns which are less complex, subdesigns are at the module level. At this level, scan compliance is analyzed, fault coverage is estimated and testability problems are identified and solved. These activities are called testability analysis. By doing testability analysis at the module level, problems are corrected early in the design cycle and time-to-market is shortened. At the chip level, testability analysis on the top level design is done as was done for the module level, this ensures all the test violations are solved, then the scan structure is inserted into the design. If boundary scan is not to be added to the design, final test vector sets are generated and formatted. However, if boundary scan is to be added, the process moves to the board level test process, in which JTAG is added, and the final test vectors are generated and formatted for a specific technology.

Referring to Figure 2.5, the methodology for using the Test Compiler on subdesigns is [29]:

1. Choose scan styles. If the design is purely combinational, define combinational as the scan style. If the design is sequential, select one of the styles the Test Compiler supports, which are Multi flip-flop, Clock Scan, LSSD and Auxiliary Clock LSSD. The Test Compiler requires that the same scan style be used on the entire chip.
2. Optimize the subdesign by using the Design Compiler.
3. Check design rules. For combinational designs, the Test Compiler checks combinational feedback loops. For sequential designs, it analyzes the design for compliance with the scan design rules associated with the selected scan style, and reports any violations.
4. Estimate fault coverage. Design rule violations cause significantly lower fault coverage. Statistical ATPG can quickly estimate the fault coverage.
5. Fix testability problems. If the fault coverage estimated is not acceptable, the designer needs to fix the testability problems which were detected in step 2. The best method is selected to fix the problem. The target for modification might be the RTL description or gate-level netlist. After the design modification, the cycle from step 1 to step 5 is repeated until an acceptable fault coverage result is achieved.

After module level testability analysis is done for all the subdesigns, the next step should be chip level testability design. The methodology using the Test Compiler at this level is:

1. The same testability analysis is done on the top design as it is done on module level, this ensures that all the violations have been identified and resolved. 100% fault coverage estimated on each module does not ensure 100% fault coverage at the chip level.
2. Insert scan circuitry. Although the scan logic is inserted at the top level, the Test Compiler can automatically insert the scan test structure through the whole hierarchy. Single or multiple scan chains, full or partial scan style can be chosen. The Test Compiler supports full scan design. The Test Compiler Plus supports partial scan design, by selecting a percentage of

sequential cells to scan according to performance, area, and testability constraints.

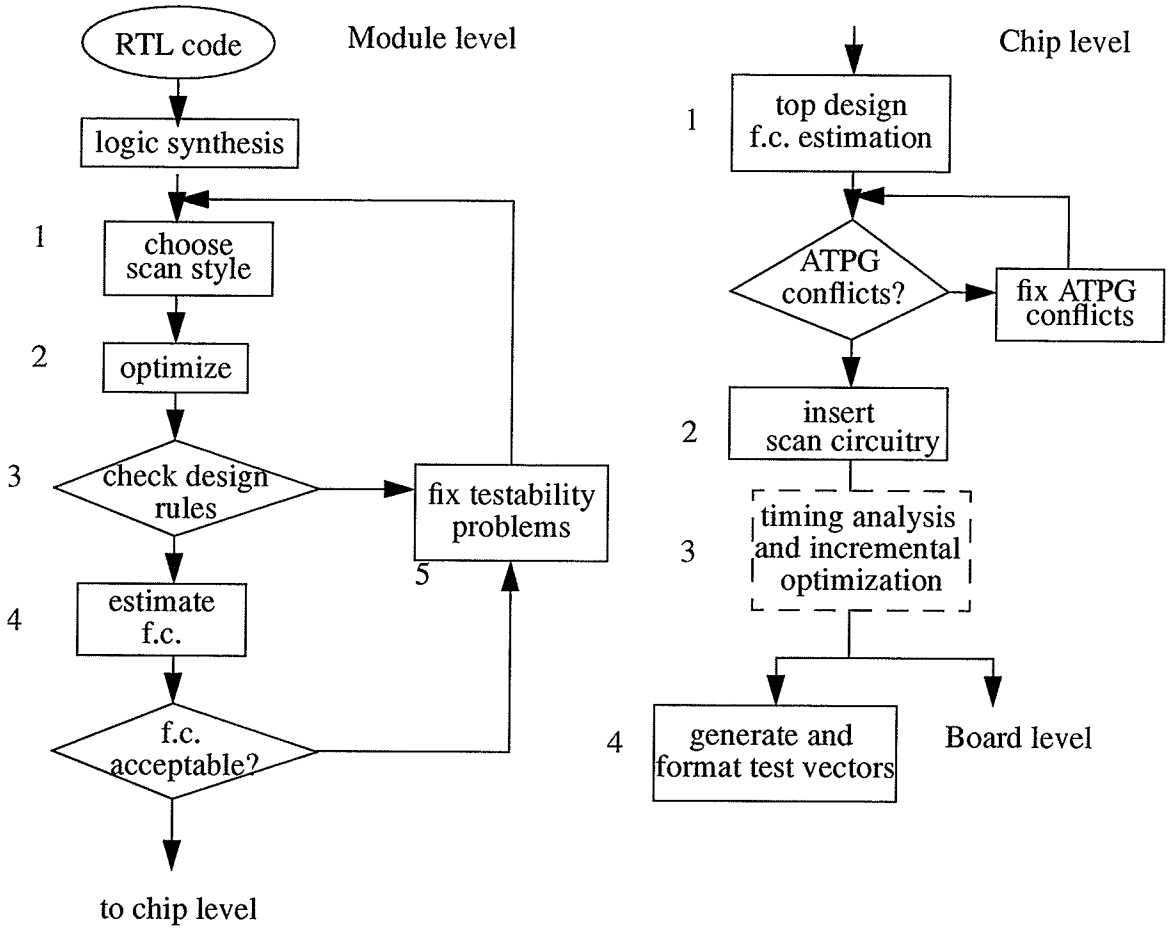


Fig. 2.5: Module and Chip Level Testability Analysis and Design

3. If necessary, perform timing analysis and incremental optimization.

4. Generate and format manufacturing test patterns. An ATPG is used to generate the final set of manufacturing test patterns which is targeted to a specific semiconductor vendor (for example, FPGA, BiCMOS, i.e.). After the generation, you can format the patterns to one of the following styles: Synopsys generic format, TDS ASCII format, WGL format, Verilog format, or VHDL format. If you want to do board level testability design, test patterns can be generated and formatted later.

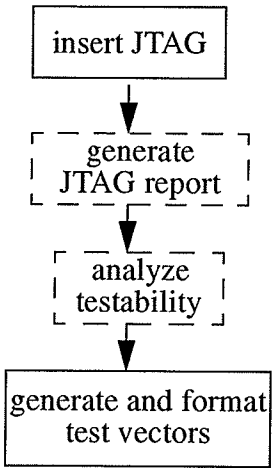


Fig. 2.6: Board Level Testability Design

The fault coverage used by the Test Compiler is defined as follows. It is used to calculate both incremental and cumulative fault coverage [28].

$$\frac{\text{faults detected}}{\text{total faults - tied - redundant}} \times 100\%$$

A tied fault cannot be tested because it is connected to logic 1 (tied high) or logic 0 (tied low). A redundant fault cannot be tested because the overall static behavior of the circuit is independent of the values at this node. Test compiler and Test Compiler Plus use a combinational ATPG algorithm for full-Scan designs; Test Compiler Plus uses a Sequential ATPG algorithm for partial-scan designs. Combinational ATPG uses random and deterministic techniques to generate test patterns for stuck-at faults. Sequential ATPG uses deterministic techniques to generate patterns for stuck-at faults.

After the chip level testability design is completed, a core design is ready for board level boundary scan insertion. This is shown in Figure 2.6. The Test Compiler does not support boundary scan synthesis on designs without an I/O pad ring. It synthesizes the 1149.1 JTAG

logic around the core logic design. After the insertion, a boundary scan report is generated and testability is analyzed. Finally, test patterns are generated for the entire design with JTAG circuitry included. The Test Compiler does not generate test patterns to test boundary scan logic, so the fault coverage of this test set is low. When the fault coverage is checked, the fault coverage for the core design will be close to the fault coverage obtained for the chip level testability design before the JTAG was inserted.

After the DFT and ATPG is done for the design using the Test Compiler, fault simulation using TestSim may be performed. Preparation for the fault simulation includes generating a TestSim library, generating a TestSim model for the design, and starting the fault simulation. TestSim can simulate both the patterns generated by Test Compiler ATPG and functional vectors generated manually. TestSim fault simulates ATPG vectors either in serial mode (for multi flip-flop designs only), or in parallel mode. TestSim does not support fault simulation for LSSD designs. The fault coverage results reported by TestSim are greater than or equal to those reported by Test Compiler. This is because Test Compiler treats some cells as black boxes due to design rule violations, but these cells are not black boxes for TestSim. The functional vectors which TestSim can use must meet two requirements. One is that the vector file must be one of the formats accepted by TestSim which are TSSI TDS, ASCII, TSSI WGL, or WIF, the other is that the vectors must be cycle-based. A set of vectors is cycle-based when this sequence of events is of the same period (or cycle) throughout the vector set.

TestSim automatically disables the scan functionality at the start of the fault simulation, so the verification result of the pre-scan logic simulation can be used on the scannable design.

The Multi-Style, Multi-Pass capability of the Test Manager enhances fault coverage [28]. This is shown in Figure 2.8.

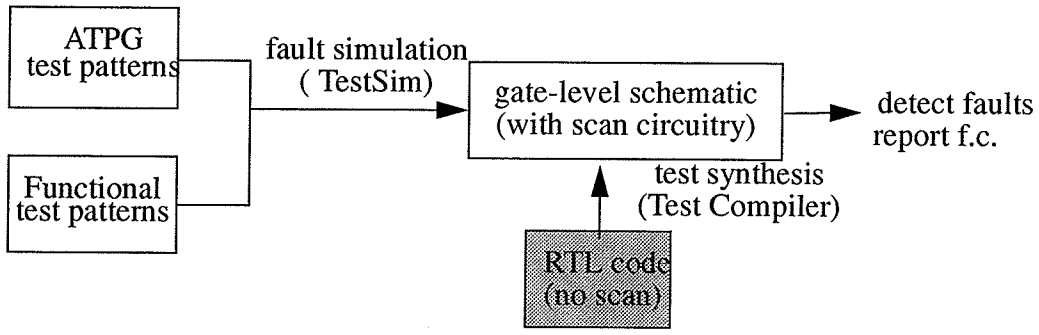


Fig. 2.7: Fault Simulation Strategies

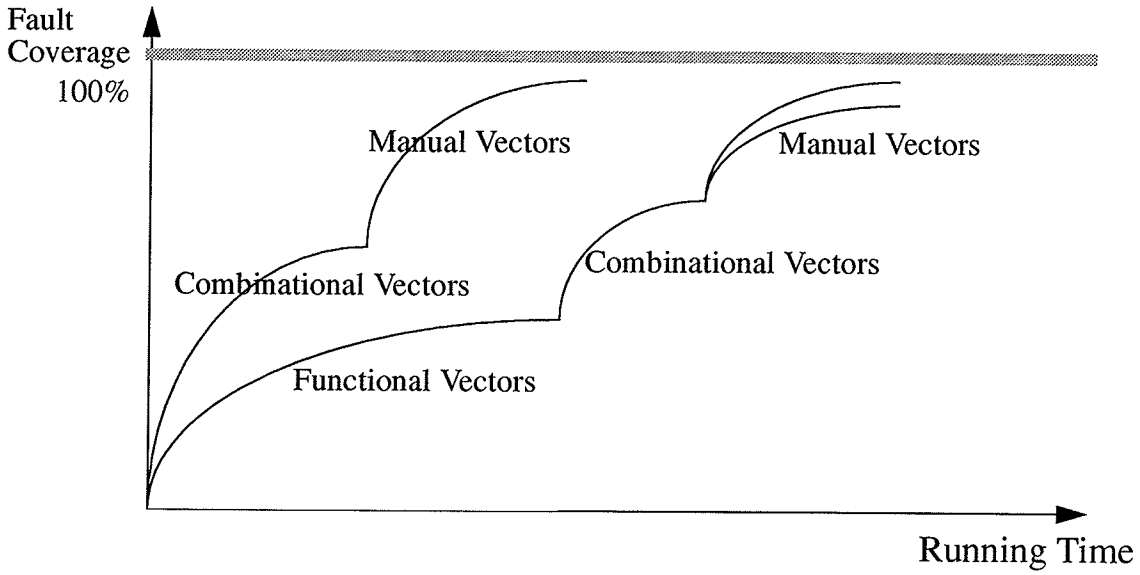


Fig. 2.8: Multi-style, Multi-pass Capability

2.6 Experience With the Synopsys Test Synthesis Tools

Testability Analysis and Design Results

Testability analysis and design results for five examples are reported here. A brief description and test results are given below. Time reported is on a SUN/SPARC5 with 64 MB of

memory. The designs are denoted *rom* (Read-Only-Memory), *tlc* (traffic light controller), *wfg* (waveform generator), *vend* (vending machine), and *counts* (count zeros circuit). Test synthesis was used both for combinational and sequential circuits, with a scannable cell number up to 49. Fault coverage of 100% is achieved for all of the circuits, with run time being very short. The number of scannable cells is the same as the number of clocks per pattern. The Synopsys test synthesis system has proven to be quite successful.

Table 1: Experiment results using the Synopsys Test Compiler

Circuit name	# of faults (non-collapsed)	# of faults (collapsed)	cpu time (sec)	f.c (%)	# of test patterns	# of compacted patterns	# of scannable squential cells	# of clocks per pattern
rom	47	27	0.12	100	4	4	0	
counts	444	267	0.29	100	34	29	11	11
wfg	641	375	0.85	100	43	35	11	12
vend	388	261	0.29	100	26	25	3	3
tlc	2143	2093	3.12	100	90	67	49	49

2.7 Utilizing ATPG Test Vectors

ATPG test patterns generated by the Synopsys Test Compiler are developed from the gate-level model. These are used for detecting stuck-at faults and reporting fault coverage results. In addition to this original purpose, the author developed two new methods to use this ATPG in which ATPG works as the normal functional VHDL testbench, simulating at the RTL and gate-level of the design. The simulation results are shown as waveforms for all the input and output signals, thus giving us a quick idea about how the signals look. If it is a sequential circuit, the signals also include the clock, test_sc, test_se and test_so signals. All the incorrect responses can be detected and the error message will be shown on the screen. ATPG test

patterns used for this purpose must be formatted in VHDL.

ATPG Test Vectors Simulated at the RTL Level

The first method developed for ATPG simulation at the RTL level is illustrated in Figure 2.9.

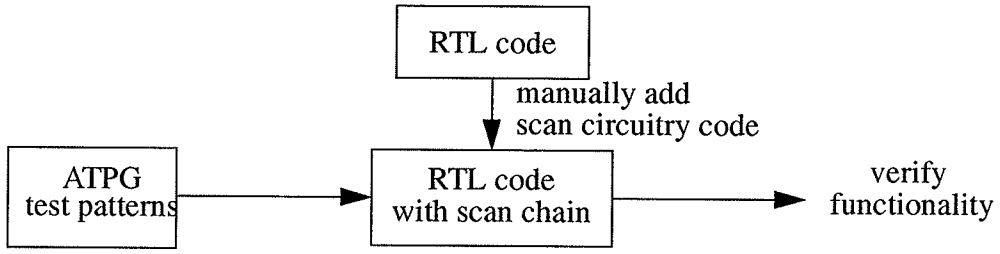


Fig. 2.9: ATPG Vectors Simulated at the RTL Level

Originally, in the ATPG testbench, the target model is the gate-level netlist synthesized and optimized from the RTL code, with scan circuitry inserted if it is a sequential circuit. But if add scan circuitry code into the original RTL code by hand, then replace the target model with this RTL code, and conduct simulation with the ATPG test patterns, the ATPG test patterns can be used as a functional testbench. Waveforms for the simulation results can be obtained and the functionality of the circuit can be verified. In this waveform representation, each input and output signal is clearly shown.

The VHDL configuration file feature enables the user to change the target model. For the original ATPG test patterns in VHDL format, its configuration file is as follows.

```
CONFIGURATION  TOP_ctl  OF TOP_tb  IS
 FOR testbench
      FOR U0: TOP USE ENTITY WORK.TOP (SYN_SCHEMATIC);
     END FOR:
 END FOR;
END TOP_ctl;;
```

It can be changed into the one as follows:

```
CONFIGURATION  TOP_ctl  OF TOP_tb  IS
 FOR testbench
      FOR U0: TOP  USE ENTITY WORK.TOP ( SCHEMATIC );
     END FOR:
 END FOR;
END TOP_ctl;
```

A simple example is used to present this methodology. The circuit ANDGATE is shown in Figure 2.10. After scan insertion, the circuit has three additional signals *test_si*, *test_se*, and *test_so*. As before, *test_si* is the scan input signal, *test_se* is the scan enable signal, and *test_so* is the scan output signal.

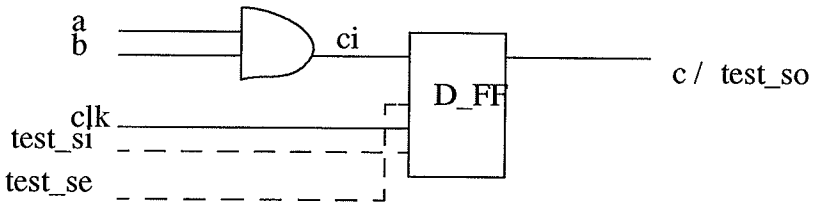


Fig. 2.10: Two-input And Gate with Registered Output (circuit ANDGATE)

The code for the register part before and after scan structure insertion is as follows. When test_se is active, the circuit is in test mode, it shifts test_in to test_out. When test_se is inactive, the circuit works in normal mode.

```
entity REG is
        port ( CI: in std_logic;
               CLK: in std_logic;
               C: out std_logic);
end REG;

architecture BEHAVIORAL of REG is
     begin
process( clk)
begin
if (clk'event and clk='1') then
c<=ci;
end if;
end process;

end BEHAVIORAL;
```

(1)

```
entity REG is
        port ( CI: in std_logic;
               CLK: in std_logic;
               test_si: in std_logic;
               test_se: in std_logic;
               test_so: out std_logic;
               C: out std_logic);
end REG;

architecture BEHAVIORAL of  REG is
     begin
process( clk)
begin
if (clk'event and clk='1') then
 if (test_se='1') then
    test_so<=test_si;
 else
   c<= ci;
 end if;
end if;
end process;

end BEHAVIORAL;
```

(2)

Fig. 2.11: VHDL Code for the Register. (1) Before, and (2) After Test Structure Insertion

Similarly, for the top level code, signals test_si and test_se are added as inputs, and test_so is added as an output in the entity port statement, and the same is done to the REG component port statement and the REG instantiation statement.

This technique is suitable for combinational circuits and simple sequential circuits. For combinational circuits, the designer only needs to modify the configuration file, and no scan insertion is needed. Manual scan structure insertion at the RTL level for complex sequential circuits may be complicated.

ATPG Test Vectors Simulated at the Gate-level

ATPG test vectors can also be used to stimulate the gate-level version of the design, For the sequential circuit, this gate-level version has scan circuitry added. Figure 2.12 shows how to get a gate-level VHDL description for the design. Test Synthesis transfers the RTL code to a Synopsys Database file, which is a gate-level netlist with scan circuitry inserted. The Synopsys SGE tool translates the database file into SGE graphics symbolic and schematic files. The Synopsys SGE tool then translates these graphics files into VHDL. Details about these processes will be discussed in Chapter 3. The gate-level VHDL description is the UUT (Unit Under Test) which will be instantiated in the ATPG testbench.

Modification of the configuration file should be done to make sure the UUT is a gate-level description. *FOR U0: TOP USE ENTITY WORK. TOP (SYN_SCHEMATIC)* should be replaced by *FOR U0: TOP USE ENTITY WORK. TOP (SCHEMATIC).*

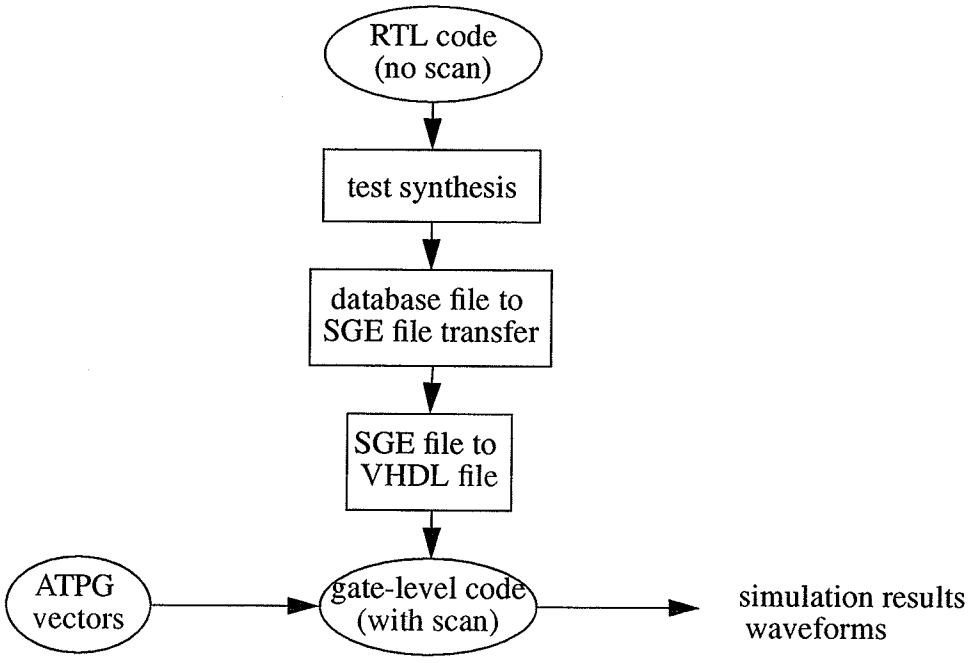


Fig. 2.12: ATPG Vectors Simulated at the Gate-Level

2.8 IC Testing Using VXI and the IC Test Head

VXIbus Test System

VXI stands for VMEbus eXtensions for Instrumentation. VMEbus is a standard protocol (IEEE STD 1014) for interconnecting digital subsystems. The VXIbus has additional connections for a variety of trigger and clock signals. By using the VXIbus, an open hardware interface between the test instruments of a testing system is defined. Greater flexibility and higher speed communication between instruments becomes possible.

The VXI test system consists of one hardware platform, the HP 75000 Model D20 [26], and a number of software platforms that can be used with this hardware. These software platforms include: HP E1496, SCPI, SICL and HP VEE.

The HP E1496 software was used in the experiment. It consists of a vector spreadsheet and a timing cycle spreadsheet. The user can define test patterns, pin groups and timing cycles in the spreadsheets. Test patterns can be stored in a PCF (Pattern Capture Format) file and imported to the spreadsheet or put into the spreadsheet manually. After all the definitions, test data can be sent to the DUT (Design Under Test), through the I/O pattern pod or timing control pod. The PCF file can be generated in an automated way.

ATPG test sets are generated by the Test Compiler and formatted into WGL (Waveform Generation Language) style, then modified into PCF format. This is the automated way the author developed to get the PCF file.

The VXI system has the following features: it is standard, flexible, and upgradable. Figure 2.13 illustrates the IC Test Head and its interface to the VXI test system.

IC Test Head

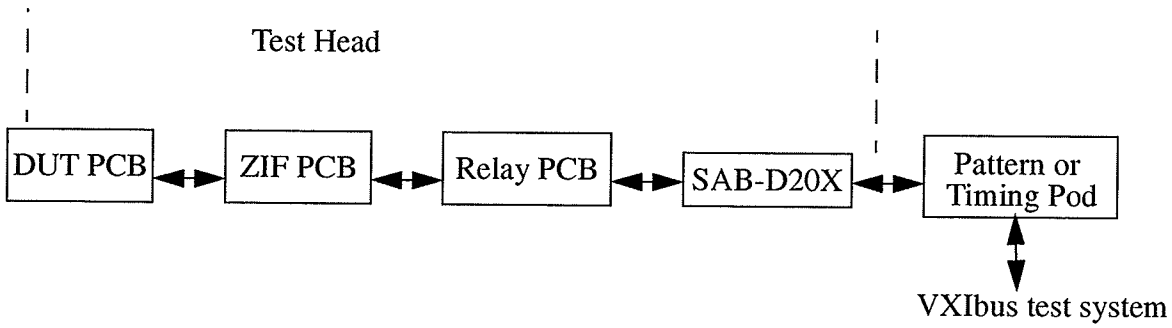


Fig. 2.13: Signal Routing and Distribution

The Test Head and DUT board are the hub of any IC test system. The CMC TH1000 mixed signal IC Test Head [25] is a low cost test fixture designed to test mixed signal integrated circuits. External signals are introduced through SABs (signal adaptor boards), which can carry digital and/or analog signals for testing.

This section is included to illustrate a physical test environment that could be used for manufacturing test within the test framework developed in thesis.

Because of the faults in relay boards and lacking of SABs, the author did not finish the testing of the ALU.

2.9 Summary

This Chapter reviewed test and Design-for-Testability techniques including full scan, partial scan and boundary scan approaches. As a more and more important technique for testability design, test synthesis was also discussed. Design flows using the Synopsys test synthesis tools on module, chip and board level were presented, experimental results were also

shown. The author developed two novel approaches for utilizing the test vectors generated by the Synopsys ATPG, one for RTL simulation, the other for gate-level simulation. Finally, Component overview for physical testing of a chip using VXIbus test system and CMC IC test head was presented.

CHAPTER 3

HIGH LEVEL DESIGN METHODOLOGIES

3.1 Introduction

This chapter will describe how a design can be developed, from a behavioral description down to a gate-level structural description. High level design methodologies and the Synopsys tools are involved in each stage, including functional partitioning, state machine to RTL code transfer, schematic and VHDL design entry, logic synthesis and optimization, database file to SGE file transfer and SGE file to VHDL model transfer. The features of Synopsys SGE, VHDL Compiler and Design Compiler tool will be presented. VHDL plays an important role in the high level design process.

3.2 The High Level Design Process

Figure 3.1 is a traditional flow producing a chip from a behavioral description. The behavioral description could be natural language (e.g. English), an executable programming language (e.g. C) or some other behavioral model (e.g. Finite State Machine, Petri Net, etc.). Simulation at this stage is for identifying malfunctions, weakness or ambiguities in the behavioral description. The behavioral description can be translated into RTL code, manually or mechanically (by automation tools). The RTL model is verified with a testbench. The

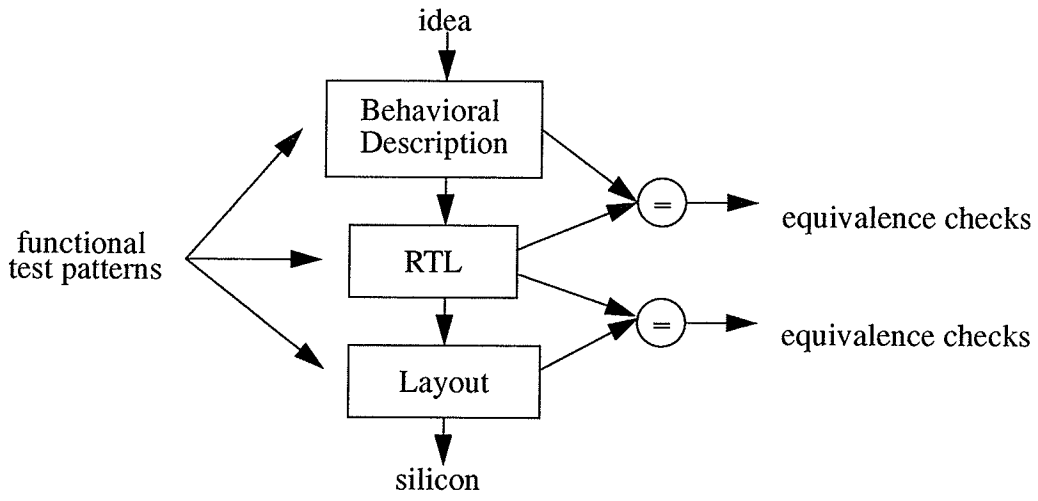


Fig. 3.1: High Level Design Flow

behavioral and RTL models can be fed with the same functional test patterns and the responses must be the same. The RTL code may be then synthesized into a gate-level form and finally transformed to layout. This could also be done by hand or automatically. The functionality of different descriptions must be proved isomorphic. This is done by applying a stimulus to different abstraction levels and comparing their results.

With this design method, specification errors are identified in the early stages of the design process, and corrections are performed in shorter iteration cycles. The functional blocks are re-usable for other projects.

Methodologies for applying stimuli to RTL and layout simulation and for comparison of their results will be discussed in Chapter 4. Designers spend most of their time in the design cycle on high-level design and validation, including checking compliance to expected behavior and modifying the design as needed. As such, this is one of the most important aspects of the design process.

3.3 The VHDL Hardware Description Language

Because of the rapid growth of IC design complexity, hierarchical top-down design methods are utilized. An efficient approach is to incorporate a hierarchical hardware description language. VHDL (Very High Speed Integrated Circuit Hardware Description Language) is such a language and helps top-down design in two ways. First, as a specification tool, simulation of complex systems can begin before implementation details are realized. Second, VHDL facilitates the top-down design process where the higher level specification is developed, debugged and finally used to judge the correctness of the next lower level specification and implementation.

VHDL plays a significant role in high-level VLSI design. VHDL has been standardized as IEEE Standard 1076 and is accepted world-wide. It is a comprehensive language that makes dealing with design complexity easier. With VHDL, data and the design process are managed effectively. VHDL itself is not overly complex, but it is rich and powerful. The scope of VHDL includes descriptions from behavioral and architectural levels to the gate level. The language is hierarchical and mixed-level simulation is supported. VHDL covers a wide range of what other languages can not. Moreover, design automation, CAD tool development, logic synthesis, simulation and test pattern generation are made much easier using VHDL [30].

VHDL has the following features: a simulation modeling language, a design entry language, a test language, a netlist language and a standard language. In the design process, there are many points at which VHDL can be helpful: design specification, design capture and design simulation, design documentation, as an alternative to schematics, and as an alternative to a proprietary language.

By using VHDL, the designer's productivity can be dramatically improved, and he/she can take advantage of powerful tools for simulation and design verification. The reasons are: 1) High reuse of VHDL by building and using libraries of commonly-used VHDL modules, 2) With the rapid pace of development in electronic design automation (EDA) tools and target technologies, using VHDL can help you move into more advanced tools without having to re-enter your circuit descriptions. Unlike other programming languages, VHDL can describe concurrent (parallel) processes and their timing.

As a summary, the primary benefits of using VHDL are as follows:

- Makes the design specification more technology-independent:
  - Allows the use of multiple vendors
  - Avoids part obsolescence
  - Documents in a standard way
- Automates low-level design details:
  - Reduces design time
  - Gets the design to market quicker
  - Reduces design cost
  - Eliminates low-level errors
- Improves design quality:
  - Allows exploration of alternatives
  - Verifies the design function at a higher level
  - Verifies that the implementation matches the expected functionality
  - Promotes design component re-use and sharing

VHDL has three types of descriptions: behavioral, dataflow, and structural. In the behavioral description, the sequential VHDL process is used to express the algorithm or behavior of a design. The structural description describes the interconnection of the components of a

design, by instantiating components. The dataflow description is in between, and is an operation defined in terms of a collection of data transformations.

3.4 Functional Partitioning

In the example elevator controller system, the system is partitioned into three modules. Each module performs a specific and relatively independent function. The modules have signals interconnecting them, so they can communicate with each other. The three modules work together to carry out the functions of the whole system. The three modules are:

- door_control
- button_timing_light
- decision_make

The door_control module controls the movement of the elevator door when the elevator stops at a certain floor. The *button_timing_light* module performs whenever the user presses the button making a request to go to a certain floor, times each request, and controls the button light turning on and off. If a request is made, the button light is turned on. Once the request is met, the button light is turned off. The final module is *decision_make*. It makes the decision whether the elevator should go up or down. After the decision is made, the motor and arrow lights are given proper signals. If the elevator goes up, the motor drives the elevator up and the up arrow light is turned on. If it goes down, the motor drives the elevator down and the down arrow light is turned on. The decision is made based on comparing which request timing is the longest and which floor the elevator is currently on.

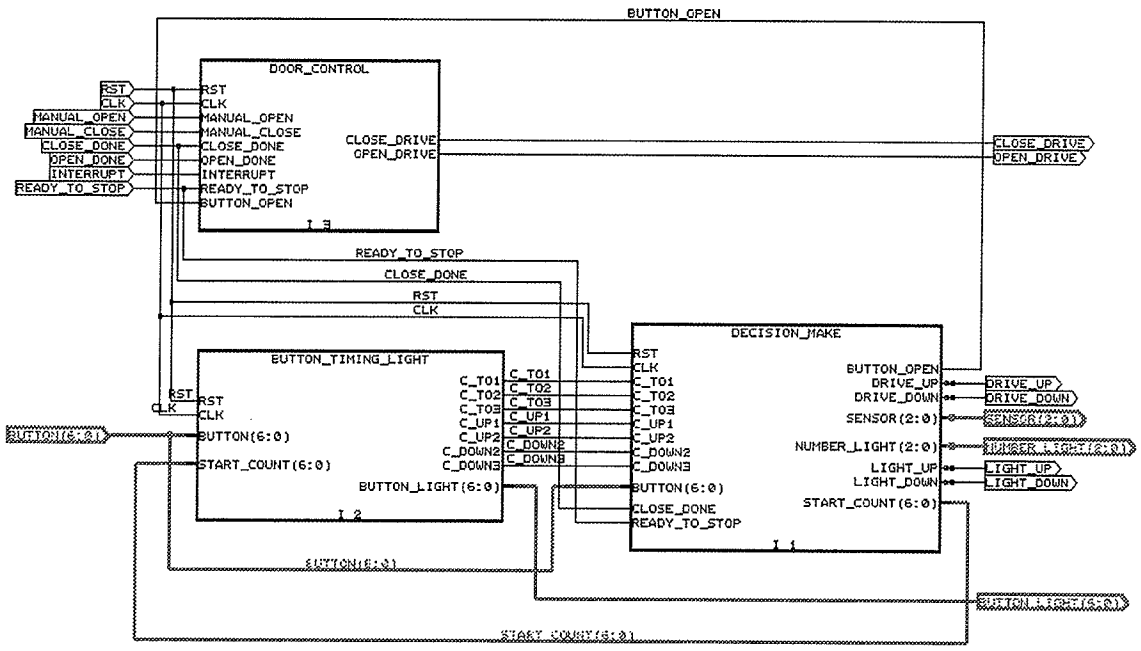


Fig. 3.2: Elevator Controller and its Submodules

3.5 Design Entry --- Schematic and VHDL

After the functional partitioning, the user can start entering a design either through schematic or VHDL code. Synopsys SGE (Synopsys Graphical Environment) is used for design entry and simulation of the design interactively. It consists of a schematic editor, a symbolic editor, a hierarchy navigator and a VHDL netlister. In the first two windows, the block diagram can be drawn and the signal names and types can be defined as shown in Figure 3.2. The VHDL netlister can generate the top level design structural VHDL, the top level design VHDL testbench template, and the bottom level module behavioral VHDL template. The user can use the text editor to manually enter behavioral code into the testbench and RTL level behavioral VHDL.

3.6 From Behavioral Description to RTL Code

The finite state machine is a very popular tool for defining the behavior of a system. It is a 5-tuple, F = (X, Q, Z, δ, ω), Where X is a finite set of inputs, Q is a finite set of states, Z is a finite set of outputs, δ is an output function, ω is the next state function.There are two kinds of finite state machines, one is the Mealy machine, another is the Moore machine. Mealy outputs are a function of state and input (Q x X-->Z), while Moore outputs are a function of state (Q-->Z) alone.

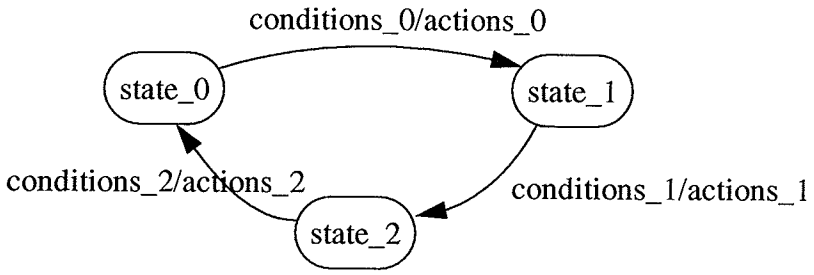


Fig. 3.3: Mealy State Transition Diagram

State Transition Diagrams and State Transition Matricies are two styles for representing state machines. The general structure of a Mealy State Transition Diagram is shown above. The diagram consists of states (circles) and transitions (arrows). Transitions are made up of input conditions that cause the transition and output actions that happen along with the state transitions.

The *door-control* module in the *elevator_controller* design is used to illustrate a realistic Mealy State Transition Diagram (see figure 3.4). *Door-control* controls the movement of the elevator door when the elevator stops at a certain floor. For the signal names, refer to Figure 3.2. There are four states for the door: *closed, opening, open, closing*. At any time, when *rst* is effective, regardless of what state the door is in, the door goes to the *closed* state. When the current state is *closed,* there is one possible next state: *opening*. When *ready_to_go* is

'1', *manual_open* is effective, someone presses the open door button, or a delay for 3 seconds occurs, the door will open. The door goes to *opening* state, and the output action is: *open_drive* is '1', *close_drive* is '0', which allows the motor to drive the door open. Other transitions are easy to understand in the same way. In Figure 3.4, all '1' values mean signal effective, '0' values mean ineffective.

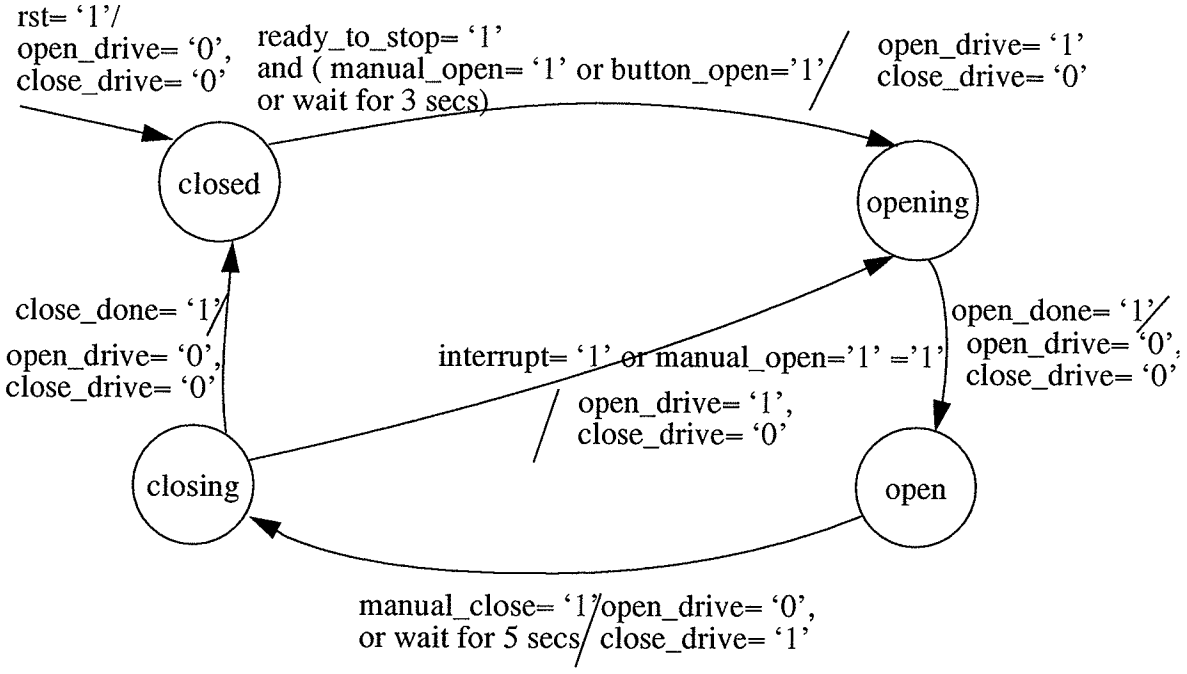


Fig. 3.4: State Machine for Door Control Module

A state machine may be coded using one or two case statements, but one case statement is preferable, because it is more concise and clear. Figure 3.5 shows a typical VHDL coding strategy for a state machine.

Mentor Graphics SDS tools have State Transition Diagram (STD) and State Transition Matrix (STM) editors. The tools can generate VHDL code from STD or STM automatically and do animation in STM to simulate the behavior of the system. These tools were not used for the elevator controller design. They are mentioned here and have been previously used

by the author merely to illustrate that commercial high level graphical STD tools are available.

```
architecture State_Machine of module_A is
type state_type is ( state_1, state_2, state_3);
signal current_state, next_state:   state_type;
begin
process ( input-list, current_state)
begin
-----synchronous reset
if (rst='0') then
outputs and next state initialization
else
-------state transitions and output logic
case current_state is
when state_1 => if ( input condition is true)then
                    next_state <= state_2;
                    assign output;
                    end if;
...
end process;
process( clk)
begin
if ( clk'event and clk='1') then
current_state<=next_state;
end if;
end process;
end State_Machine;
```

Fig. 3.5: Typical VHDL Coding for a State Machine

3.7 From RTL Level to Gate Level

The process which transfers RTL level code to gate-level code is shown in Figure 3.6. The process includes three stages: logic synthesis and optimization, database file to SGE file transfer, and SGE file to VHDL file transfer. Each stage will be discussed below.

Logic synthesis and optimization:

Synthesis is a process that transforms a circuit defined at one level of abstraction into a lower level definition subject to contain constraints. Optimization modifies the structure of the

logic according to timing and area constraints which are defined by the user, and maps the netlist to a certain technology.

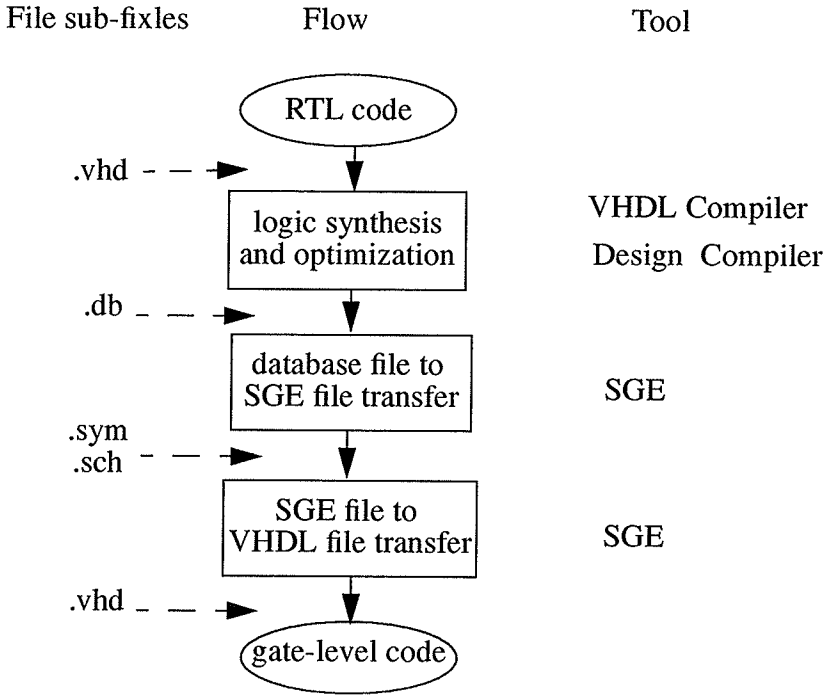


Fig. 3.6: From RTL Level to Gate-Level

The Synopsys VHDL Compiler and Design Compiler are synthesis tools that work together to carry out the logic synthesis and optimization tasks. The Synopsys VHDL Compiler tools optimize the VHDL design at the architecture level. The Design Compiler tools optimize the design at the gate-level. The VHDL Compiler accomplishes two things: converting VHDL to an internal format, and optimizing block level representation through different optimization methods. The Design Compiler reads the design in internal format from the VHDL

Compiler, then optimizes and maps the design's structure for a specific technology. The VHDL Compiler synthesizes VHDL according to a certain synthesis policy. A synthesis policy consists of three parts: design methodology, design style, and language constructs. The Design Compiler takes the internal database format generated by the VHDL Compiler, performs logic optimization for area and timing, reconstructs part or all of the design, and maps it to a target technology.

Accurate and achievable constraints must be set before a design is optimized. These constraints include: clock constraints (clock period and edge), time delay constraints (for rising edge and falling edge, the maximum rise and fall, the minimum rise and fall times), and area constraints. After constraints are set for a design and before a design is compiled, the *check_design* command is run to identify and help correct any problems in the design.

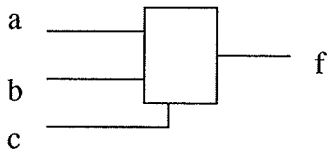
Database file to SGE file transfer:

After synthesis and optimization using the VHDL Compiler and the Design Compiler, the technology-dependent gate-level netlist of the design will be a Synopsys database file (.db). To transfer this file to SGE graphics, the technology library must be translated to VHDL. The symbols for the CLASS library should be added to the Symbol Library search path in the .synopsys_sge.setup file. Using the "*db2sge*" command or the SGE Hierarchy Navigator window menu "*setup>DA to SGE transfer*", the database file is transferred to SGE files, a schematic file (.sch) and a symbolic file (.sym).

SGE file to VHDL file transfer:

Finally, a new VHDL design description can be created, by using the Synopsys SGE window menu command "*code VHDL models*". This a structural VHDL design. The architec-

ture consists of instantiation statements for gate-level components (refer to Figure 3.8 which presents architecture level VHDL for the multiplexer of Figure 3.7). Library CLASS is the ASIC vendor technology for this design. Port maps in the component instantiation statements express network connectivity. The configuration statement (see Figure 3.9) refers to primitives from the library CLASS with architecture FTSM. At this stage, a gate-level VHDL file (.vhd) is created for the design. In Chapter 4, this gate-level VHDL file will be shown to be very useful when doing after-synthesis simulation or mixed level simulation for a design.



when c is '1', f is a;
when c is '0', f is b;

a, b, c, f are all one bit.

Fig. 3.7: Multiplexer

The RTL version for multiplexer is:

```
entity MUL is
port(a,b,c: in std_logic; f: out std_logic);
end MUL;

architecture BEHAVIORAL of MUL is

begin
process(a,b,c)
begin
if c='1' then
f<=a;
else
f<=b;
end if;
end process;

end BEHAVIORAL;
```

Fig. 3.8: RTL Level VHDL Description for Multiplexer

```
--- VHDL Model Created from SGE Schematic mul.sch -- Apr 26 14:13:36 1996

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
library CLASS;
use CLASS.components.all;

entity MUL is
    Port (      A : IN STD_LOGIC;
                B : IN STD_LOGIC;
                C : IN STD_LOGIC;
                F : OUT STD_LOGIC );
end MUL;

architecture SCHEMATIC of MUL is
  signal     n17 : std_logic;
  signal      f_DUMMY : std_logic;

begin

  f <= f_DUMMY;
  U8 : IVA
    Port Map ( A=>b, Z=>n17 );
  U7 : EON1
    Port Map ( A=>n17, B=>c, C=>a, D=>c, Z=>f_DUMMY );

end SCHEMATIC;

configuration CFG_MUL_SCHEMATIC of MUL is

  for SCHEMATIC
    for U8: IVA
      use entity CLASS.IVA(FTGS);
    end for;
    for U7: EON1
      use entity CLASS.EON1(FTGS);
    end for;
  end for;

end CFG_MUL_SCHEMATIC;
```

Fig. 3.9: Gate-level VHDL Description for Multiplexer

Some statements are non-synthesizable, such as: *wait for, assert, read, readline, while( ), loop*.

3.8 Summary

This chapter considers the issues of high level design methodologies, discusses how to develop a design from the initial specification down to gate-level implementation. The state machine is presented as a very useful behavior description tool. RTL VHDL code can be derived from the state machine model. Logic synthesis and optimization can be done automatically using CAD tools. A gate-level version of a VHDL file can be transferred from the RTL version of the VHDL file using the CAD tool Synopsys SGE. The VHDL feature demonstrated is that it works as both behavioural and implementation language. This is illustrated using an elevator controller design example.

CHAPTER 4

TESTBENCH GENERATION AND APPLICATION METHODOLOGIES

4.1 Introduction

In Chapter 3, it was mentioned that one of VHDL's benefits is that it is a test language. How VHDL works as a test language will be discussed in this chapter. This feature of VHDL is referred to as the VHDL testbench. Testbenches work for both manufacturing and functional verification purposes. The testbench that works for manufacturing verification purposes can be generated automatically by CAD tools, as was discussed in Chapter 2. This chapter focuses on functional testbench generation and application methodologies. In this chapter, testbench concepts, styles, generation methodologies, and how a testbench simulates at different levels of abstraction will be discussed. The testbench simulation results for the elevator controller design will also be presented.

4.2 VHDL Testbench

A testbench is a functional block, where the UUT (Unit Under Test) and reference models are embedded (it is also called a test bed). Responses from the reference model and from the UUT will be compared at certain clock periods. From this idea, a practical testbench scheme has been developed as illustrated in Figure 4.1 which shows the high level structure of the

testbench.

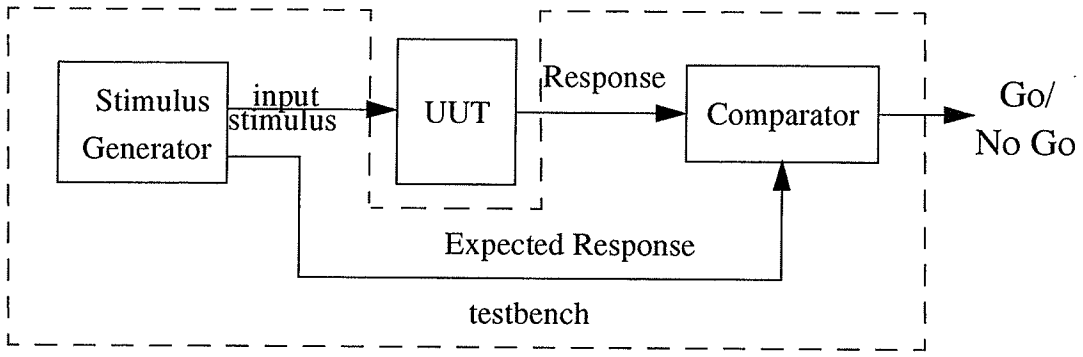


Fig. 4.1: Structure of A Testbench

The testbench consists of a Stimulus Generator block and a Comparator block. The design to be tested, is instantiated in the testbench as the unit under test (UUT). The Stimulus Generator block is responsible for stimulating the UUT and is the heart of the testbench. The Comparator block monitors device outputs and compares the expected outputs with the real outputs. If the real response does not match the expected response, an error message will be shown, and modification of the testbench, UUT, or test vectors is needed. The goal of a testbench is to thoroughly exercise the UUT by creating stimulus generators and comparators that exercise the UUT to verify its compliance to specification.

The testbench can be used to verify behavioral descriptions at the RTL level and to verify structural descriptions at the gate-level. The higher the abstraction level the testbench is generated in, the more convenient it is to control. The trend is to use a high level to generate test programs.

VHDL based testbenches have the following characteristics:

- Development efficiency
- Reusability

- Inherent documentation
- Tool vendor independence
- Mixed level simulation - VHDL based testbenches allow simulation at all levels. From the behavioral level down to the gate level

A testbench consists of four major sections: library inclusions, entity declaration, test architecture and configuration statements. Libraries contain packages of shared subprograms. The testbench is the uppermost level of the hierarchy. All operations are performed inside the testbench, so it has no input and output signals in its entity. In the architecture declaration, there are component declarations (indicating which design is used as the UUT), signal declarations, and input stimulus generation and response checking processes. The components are instantiated in the configuration statement.

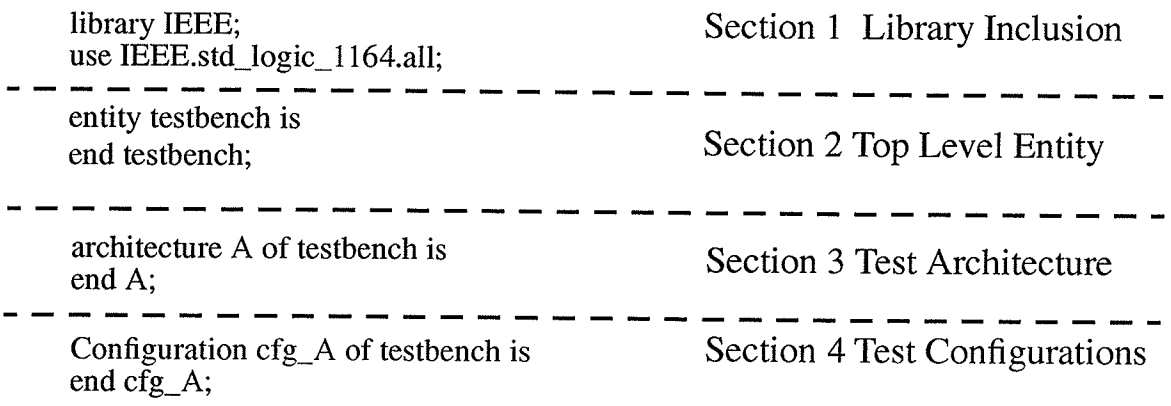


Fig. 4.2: Testbench Structural Contents

Functional behavior is more easily described, analyzed and debugged when working with high level descriptions of the design. The same applies to a testbench. Tests are more easily described, analyzed, and debugged when working with high level descriptions of the testbench. In this thesis, high level descriptions have the same meaning as functional and behavioral descriptions.

Developing HDL (Hardware Description Language) testbenches at high levels of abstraction is the current trend. It is desirable to raise the level of abstraction, allowing testbench reuse and achieving higher quality results. Post synthesis (gate-level) simulation represents a minimal percentage of overall simulation activity in an HLD-based methodology. RTL level simulation occupies the greater percentage of the simulation activities.

The values used for input stimulus and output comparison are called test vectors. Test vectors can be created and made available during simulation in many different ways. They may be used in various combinations. For example, you can generate stimulus with a VHDL process, and use some other method to check the circuit response.

4.3 Testbench Styles and Coding Strategies

There are different ways to do stimulus generation, application, and response checking. As a result, testbench design has numerous styles. Although it is not necessary to do comparisons of real responses with expected responses, it is recommended to do so. Generally speaking, testbenches have three major styles: Ad-Hoc, algorithmic, and vector files [30]. These styles are discussed below.

4.3.1 Ad-Hoc

An ad-hoc collection of stimulus cases are used to exercise the basic functions of the UUT. The ad-hoc style is used in the *door-control* testbench as shown is Figure 4.3. This example intends to check if the *manual_open* signal can drive the *open_drive* signal, and that *manual_close* can drive the *close_drive* signal. The simulation result is shown in section 4.8. Ad-Hoc was the major testbench style used for the *elevator_controller* design.

```
signal_stimulus: process
begin
ready_to_stop<='0';
manual_open<='0';
manual_close<='0';
close_done<='0';
open_done<='0';
interrupt<='0';
wait for 50 ns;
ready_to_stop<='1';
wait for 40 ns;
ready_to_stop<='0';
wait for 200 ns;
manual_open<='1';
wait for 40 ns;
manual_open<='0';
wait for 200 ns;
open_done<='1';
wait for 40 ns;
open_done<='0';
wait for 200 ns;
manual_close<='1';
wait for 40 ns;
manual_close<='0';
wait for 200 ns;
close_done<='1';
wait for 40 ns;
close_done<='0';
wait for 1500 ns;
end process;
```

Fig. 4.3: Ad-Hoc Test Vector Coding for door_control Module

4.3.2 Algorithmic

Another way to generate stimuli is to write an algorithm. For example, in Figure 4.4, a loop statement is used to generate an incremental signal value. This code was used in the functional verification of a 74181 ALU. The simulation results are shown in Appendix B.

```
A_stimulus: process
variable i: integer;
variable seed: std_logic_vector(3 downto 0);
begin
A<="0000";
seed:="0000";
wait for 20 ns;
for i in 1 to 7 loop
seed:=seed+"0010";
A<=seed;
wait for 20 ns;
end loop;
end process;
```

Fig. 4.4: Algorithmic Test Vector Coding for the ALU Testbench

4.3.3 Vector Files

A testbench process fetches test vectors (including input stimuli and expected responses) from a data file. Thus it reads the vectors as stimulus and also monitors whether the behavior of the UUT matches the expected response in the test vector file. These test vectors might be captured during an interactive simulation session or generated using an algorithmic approach.

Figure 4.6 shows one style of a testbench which obtains test vectors from an external data file. The design is a double *and* gate (*and_2*) as shown in Figure 4.5. In this case, two packages *textio* and *std_logic_textio* must be included in the library inclusion statement. The process begins with a set of variable declarations. The declaration of F tells the simulator the name of the test vector file, *and2.vec*. The *readline* statement allows one line read from the file F to be placed to the variable L. The *read* operations allow values from the line L to be "read" into a variable. The code for fetching and reading the values into variables is in the section labelled *get a vector.* Any difference between expected and real outputs are reported

by the *assert* statements.

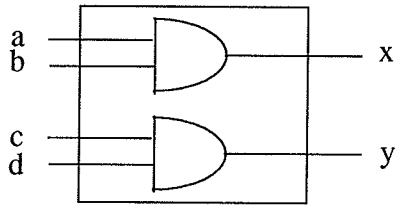


Fig. 4.5: Example: and_2

```
test: process
file F: text is in "and2.vec";
variable L: line;
variable a_in,b_in,c_in,d_in,x_out,y_out:std_logic;

begin
readline (F,L);
read (L,a_in);
read (L,b_in);
read (L,c_in);
read (L,d_in);
read (L,x_out);
read (L,y_out);

if endfile(F) then
wait;
end if;
wait for 400 ns;

a<=a_in;
b<=b_in;
c<=c_in;
d<=d_in;

wait for 100 ns;
assert x=x_out
report " Unexpected response on x";
assert y=y_out
report " Unexpected response on y";
wait for 400 ns;
end process;
```

get a vector

assign input values

check output values

Data file *and2.vec*:

```
000000
010100
111111
```

Fig. 4.6: Testbench Fetch Test Vectors From Data File and Data File

In the vector file, the storage order is a, b, c, d, x, y, i.e. a is the left most bit, and y is the right most bit. The simulation waveforms produced are illustrated in Figure 4.7.

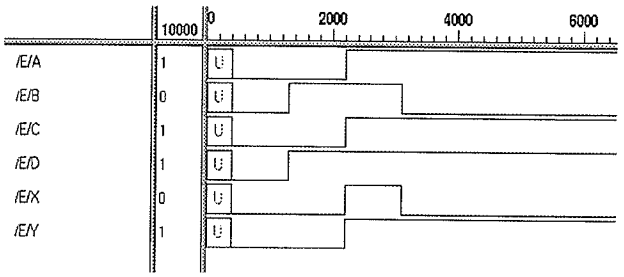


Fig. 4.7: Simulation Result for Circuit and_2

For a sequential circuit, the clock can be generated using an independent process like the one shown in Figure 4.8. In this case, after the input value assignment, the statements "*wait until clk'event and clk='1'*" or "*if (clk'event and clk='1')*" can be used to perform synchronization for output value checking. The vector file has no clock information in it.

```
clock_generation: process
begin
clk<='1';
wait for half_period;
clk<='0';
wait for half_period;
end process;
```

Fig. 4.8: Clock Generation Process

4.4 Testbenches for RTL and Schematic Model Equivalence Checking

After the RTL model is verified to be functionally correct, it can be synthesized and optimized into a schematic model (e.g., gate-level netlist). Does the optimized model still maintain the right functionality? In the following scheme, the same set of test vectors are applied to both the RTL and schematic models of the circuit, then the outputs from the two versions are compared. If they are equal, it means the optimized model has the right functionality.

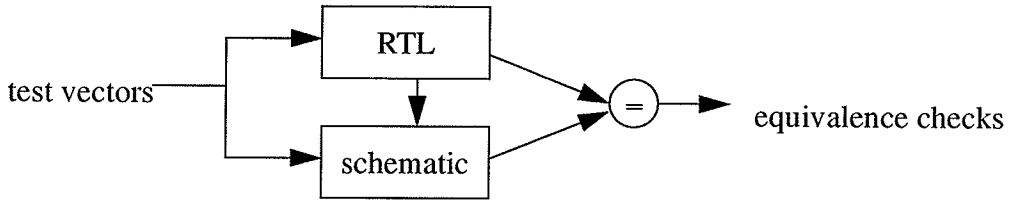


Fig. 4.9: Equivalence Checking for RTL and Schematic Models

There is more than one way to perform this task. One method is to plug the gate-level version of the design into the same testbench used for the RTL level functional verification and simulation. As long as no assertion errors are generated, it assures that the gate-level version of the design is error-free. Another method is to use the same set of test vectors (or same testbench) to stimulate both pre-synthesis and post-synthesis design, then save the waveform file (.ow) which contains simulation results for each of them, and finally compare the two waveform files, i.e., compare simulation results with the General Purpose Post Processor (GPP).

A novel approach has been developed by the author to realize this scheme. In this approach, a testbench is used to generate input stimuli, and compare the outputs from the RTL and the schematic models. Both RTL and schematic components are instantiated in this testbench. A small example shown in Figure 4.10 is used to illustrate this methodology.

The design has two inputs: i1 and i2, one output o. Their types are respectively i1_type, i2_type, o_type. The RTL version for the design is shown in Figure 4.11, and the schematic model is shown is Figure 4.12. The name DESIGN is given to the design. The RTL version and schematic version have the same entity DESIGN, but different architecture bodies, one is DESIGN (BEHAVIORAL), the other is DESIGN (SCHEMATIC).

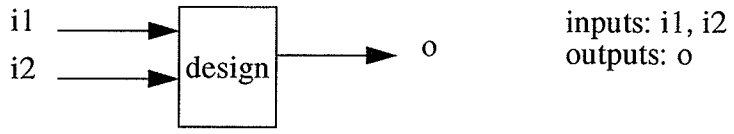


Fig. 4.10: Design Example

```
entity DESIGN is
port (i1: in i1_type; i2: in i2_type; o: out o1_type)
end DESIGN;

architecture BEHAVIORAL of DESIGN is
begin
...
end BEHAVIORAL;

configuration CFG_DESIGN_BEHAVIORAL of DESIGN
for BEHAVIORAL
end for;
end CFG_DESIGN_BEHAVIORAL;
```

Fig. 4.11: RTL Version (V1) for the Design

In Figure 4.13, the RTL version for the design is represented by V1 and the schematic version for the design is represented by V2. V1 combined with V2 are looked at as the UUT, so the UUT has two inputs i1, i2 and two outputs o_v1 and o_v2. The UUT is considered a new component whose name is DESIGN_U. V1 and V2 are two components of the UUT, which is a design with a mixed level description. This is realized by using the port map statement (component DESIGN is used, V1 and V2 have the same inputs and different outputs o_v1

and o_v2), and component instantiation configuration file (for component V1, use the RTL

```
entity DESIGN is
port (i1: in i1_type; i2: in i2_type; o: out o1_type)
end DESIGN;

architecture SCHEMATIC of DESIGN is
begin
...
end SCHEMATIC;

configuration CFG_DESIGN_SCHEMATIC of DESIGN
for SCHEMATIC
end for;
end CFG_DESIGN_SCHEMATIC;
```

Fig. 4.12: Schematic Version (V2) for the Design

version, for V2, use the schematic version). The VHDL model for the UUT is shown in Figure 4.14.

Figure 4.15 shows how to use the testbench to input stimuli to the UUT and compare the outputs from the UUT. The UUT is incorporated into the testbench as component DESIGN_U. There are two kinds of processes in the testbench: stimulus generation and response comparison processes. The stimulus generation process generates test vectors for the UUT, while the response comparison process compares the outputs from the UUT, (i.e., compares outputs from two components V1 and V2). This is done by using the assert and report statements. If o_v1 is not equal to o_v2, the message "There is a difference between two versions of the design!" will be displayed on the screen. There may be some variations for the VHDL model shown in Figure 4.15. For example, the stimulus generation process may be partitioned into a number of processes to make it easier to design complex stimuli. And the response comparison process may be merged into the same process as the stimulus generation. The stimulus generation process can also be the same one used in the testbench for the RTL level verification.

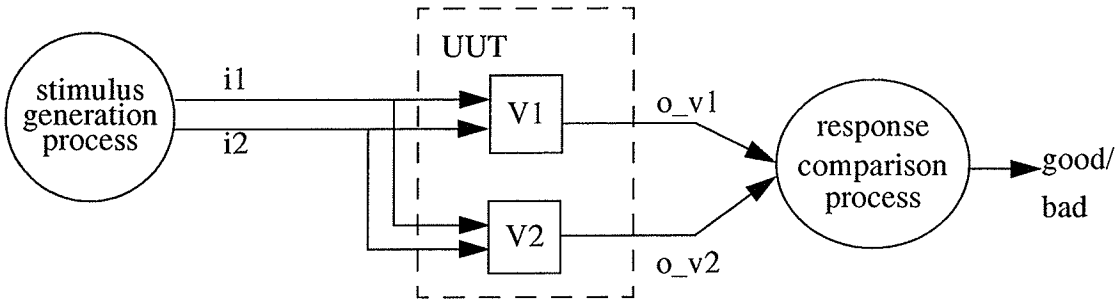


Fig. 4.13: Design in Two Versions V1 and V2

```
entity DESIGN_U is
port (i1: in i1_type; i2: in i2_type; o_v1, o_v2: out o1_type)
end DESIGN_U;

architecture MIX of DESIGN_U is
component DESIGN
port (i1: in i1_type; i2: in i2_type; o: out o1_type)
end component;

begin
V1: DESIGN port map (i1, i2, o_v1);
V2: DESIGN port map (i1, i2, o_v2);
end MIX;

configuration CFG_DESIGN_U_MIX of DESIGN_U is
for MIX
for V1: DESIGN use configuration work.CFG_DESIGN_BEHAVIORAL;
end for;
for V2: DESIGN use configuration work.CFG_DESIGN_SCHEMATIC;
end for;
end CFG_DESIGN_U_MIX;
```

Fig. 4.14: VHDL Model for UUT

```
entity E is
end E;

architecture TB of DESIGN_U is
component DESIGN_U
port (i1: in i1_type; i2: in i2_type; o_v1, o_v2: out o_type)
end component;

signal i1: i1_type;
signal i2: i2_type;
signal o_v1, o_v2: o_type;
begin
UUT: DESIGN_U
port map(i1,i2, o_v1, o_v2);

stimulus_generation: process
begin
...end process;

response_comparison: process
begin
...assert( o_v1 = o_v2)
report "There is difference between the two versions of the design! "
...end process;

end TB;

configuration CFG_E_TB of E is
for TB
for UUT: DESIGN_U use configuration work.CFG_DESIGN_U_MIX;
end for;
end for;
end CFG_E_TB;
```

Fig. 4.15: Testbench Simulation on UUT

This approach can be applied to designs of arbitrary complexity, both for combinational and sequential circuits.

Simulation results for the multiplexer example are shown in Figure 4.16. The output from the RTL version of the design, f1, is the same as the output, f, from the gate-level version.

There is no assertion error message output.

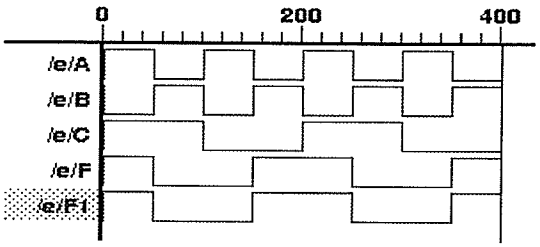


Fig. 4.16: Testbench Simulation on RTL and Gate-level Equivalence Checking

4.5 Testbench For Mixed Level Simulation

By modifying testbench configuration statements, the same testbench can also be simulated at different levels. The technique is demonstrated below.

Consider the UUT of Figure 4.17 which consists of three components: U1, U2, and U3. They have several signals connecting them. U1, U2, and U3 are instantiated in the TOP VHDL code, and in Figure 4.18, we can see that U1 and U3 are instantiated with a BEHAVIORAL statement which is at the RTL level, while U2 is instantiated with a SCHEMATIC statement which is at the gate-level. The TOP component is the UUT instantiated in the testbench. By using the configuration files shown in Figure 4.18 and 4.19, testbench simulation on a mixed level can be performed.

4.6 Simulation With Back-annotation

After equivalence checking for the RTL and schematic models has succeeded, the schematic model can be transformed into a layout for a specific technology. Timing information intro-

duced by the layout can be applied to the testbench simulation. Timing information is associated with the simulation results. The Synopsys Design Compiler can generate a SDF (Standard Delay Format) file which contains timing data information derived from the layout.

By doing simulation with back-annotation, functional and timing verification can both be done at the layout level.

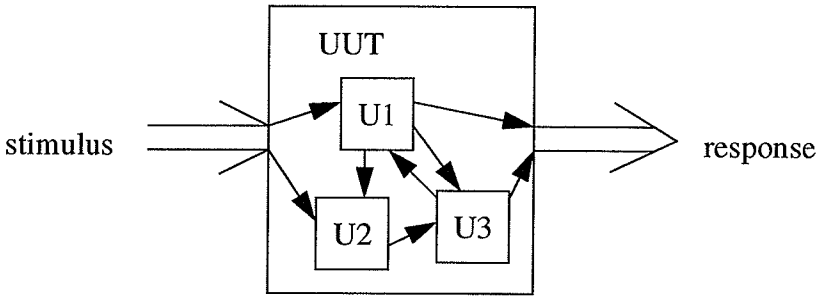


Fig. 4.17: UUT (TOP) and its Components

```
configuration CFG_TOP_SCHEMATIC of TOP is
for SCHEMATIC
for U1: A use configuration  work. CFG_A_BEHAVIORAL;
end for;
for U2: B use configuration work. CFG_B_SCHEMATIC;
end for;
for U3: C use configuration work. CFG_C_BEHAVIORAL;
end for;
end for;
end CFG_TOP_SCHEMATIC;
```

Fig. 4.18: Configuration Statement for TOP Design

```
configuration CFG_TB_BEHAVIORAL of TB is
for BEHAVIORAL
for UUT: TOP use configuraion work. CFG_TOP_SCHEMATIC;
end for;
end for;
end CFG_TB_BEHAVIORAL;
```

Fig. 4.19: Configuration Statement for Testbench

4.7 A Survey of Behavioral Test Pattern Generation Algorithms

The classical test generation approaches described in Chapter 2 operate at the gate level. DFT techniques, such as scan, add chip area, decrease circuit performance. Further, gated clock signals make the creation of a complete scan path unachievable. In addition, asynchronous designs make it important to resort to a complete scan solution. It is pointless to design a 10 million gate chip which works perfectly to specification if the specification is wrong. Test generation at the behavioral level has become increasingly important in verifying that the specification is also correct. When the circuit is very large and complicated, hand coded functional testbenches are hard pressed to satisfy the verification completeness. As a result, some algorithms are used for generating functional test vectors from a high level description. Within these algorithms, variations of the D-algorithm are the most important ones. These test vectors then augment the test suite of hand code and structurally derived test vectors.

ATPG Views

Automatic test pattern generation has structural and behavioral views. The structural view was discussed in Chapter 2. The behavioral view can be realized by means of two kinds of representation: alphanumeric or graph-based. Alphanumeric representations include differ-

ential equations, boolean equations, truth tables, state tables, and programs. Graph-based representations include BDDs (Binary Decision Diagrams), Transformation Graphs, and Petri-Nets. Alphanumeric representations can be transformed into graph-based representations. When the circuit is very large, the alphanumeric representation grows exponentially with the number of variables involved, in this case, it is better to use a BDD which is a very promising representation. There are various techniques available, the most popular one is the path sensitization method. A few representative publications will be used to discuss this method in the following. No matter what name they call their approaches (B-algorithm, E-algorithm, etc.), or what kind of circuit representation they use, or what kind of fault models they defined, the basic idea is path sensitization.

Path sensitization is an extension of the D-algorithm, including three steps: 1) sensitization ---the effect of the fault is made to manifest itself at the fault site, 2) fault propagation--- the fault is moved through the circuit until it can be observed at an output, and 3) justification--- the procedure required to establish the propagation path back to the inputs. Path sensitization employs a standard notation, using logical values 0, 1, X, D, $\overline{D}$. This principle has been applied to several abstraction levels: logic level [3], RTL level [23], and higher levels [12].

Levendel and Menon [3] proposed extensions to the gate-level D-algorithms to handle faults in functional blocks of HDL's. The test generation algorithm consists of the following conceptual steps: 1. insert a fault effect, 2. propagate the fault effect to an observable point, 3. justify all the decisions made in 1 and 2 by sequences applied at the primary inputs of the circuit. Their approach gives strategies for D-propagation through functional blocks, but has a high computational cost for complex control structures.

The fault model they used are: function variables stuck at 0 or 1, control faults, and general function faults.

The E-algorithm [23] is another approach to test generation from a Hardware Description Language, using propagation rules proposed by Levendel and Menon. It generates tests for the following modeled faults: control, operation, and data faults (stuck-at faults in data or control lines and defects in functional operation blocks). The HDL it uses is limited to non-procedural data-flow representations. It uses a graph transformation representation derived from the VHDL description.

Barclay and Armstrong's approach [4] employs the FCON (involves faulting the control points that switch between sequences) and FMOP (involves faulting the individual micro-operations) fault models, and attempts to develop an equivalence of the D-algorithm for HDL descriptions of the chip logic. The approach is as follows. Given a FMOP or FCON, a test is developed by 1) activating the faulted operation (fault sensitization), 2) propagating the effect of the fault to an output (path sensitization), and 3) determining input combinations that will justify the path sensitization. The models are represented in VHDL, and the form of the representation is restricted to data flow models. Barclay's algorithm first generates the fault list which gives chip-level faults for each statement, then starts the test generation process. They used the artificial intelligence concept of goal trees to organize the algorithm structure. Their technique generates tests for faults in control structures as well as data path faults, but is computationally expensive.

Some other approaches are also based on the D-algorithm. The B-algorithm [5] uses three behavioral faults: behavioral stuck-at faults (BSA), behavioral stuck-open faults (BSO), and micro-operation faults (MOP). It defines its own activation and propagation rules. The B-algorithm has two unique features, one is that it can generate tests for behavioral stuck-open faults, which can detect some gate-level transition faults. The other is that it incorporates the concept of two-phase testing, a testing strategy where a fault is detected using two consecutive test sequences. Abadir and Reghbati [15] use two kinds of fault models. A class 1 fault

is a single stuck-at fault affecting a module input line, output line, or memory element. A class 2 fault is a module internal fault which adversely affects the outcome of one of its experiments. They used a BDD representation to do path sensitization. Courbis [21] uses three fault models which are F1: Stuck-at fault of an element of the data model, F2: stuck-at fault of an element of the control model, and F3: stuck-at fault of an element of the interaction. He uses Petri-Nets to do activation and propagation to derive test vectors.

4.8 Simulation Results and Discussions

The elevator controller design example was developed utilizing the high level design methodology previously discussed. Simulation with the testbench was used to verify the functionality of each sub-module and the top module. During this phase, the behavioral VHDL code for each module was iteratively modified, satisfactory simulation results were obtained and are presented in detail as follows. For additional information on the design, refer to Appendix A.

Figure 4.20 gives the simulation results for the *door_control* module. The states "00", "01", "10", and "11" stand for *closed*, *opening*, *open*, and *closing* respectively. If *rst* is effective, the elevator goes to the initial state which is *closed*. After *ready_to_stop* is effective (which means the elevator is going to stop), if *manual_open* is '1', the door starts opening. When the *open_done* signal is active (which means the door has finished opening), the door state becomes *open*. In this state, if the *manual_close* becomes '1', the door state becomes *closing*, until the *close_done* signal is active, and the next state is *closed*.

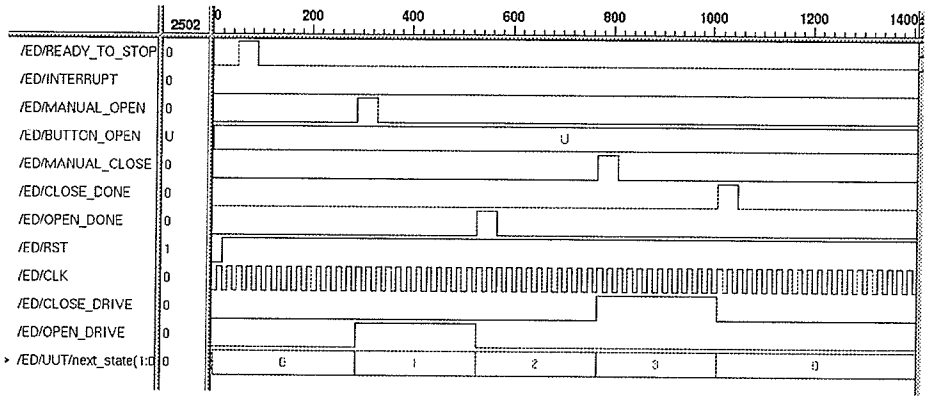


Fig. 4.20: Simulation Result for door_control Module

Figure 4.21 shows the simulation results for the *button_timing_light* module. In this simulation, the button increments by 2, e.g., at first, the *button* is "0000001", which means the button to1 is pressed, the next time, *button* is "0000011", which means the button to1 and to2 are pressed, incrementally, until the button is "7E". When *start_count* is effective, and certain buttons are pressed it starts counting the time for buttons which have been pressed, during which time those buttons' lights are on. When *start_count* is ineffective, the timing count for the buttons become zero, and the buttons' lights are off.

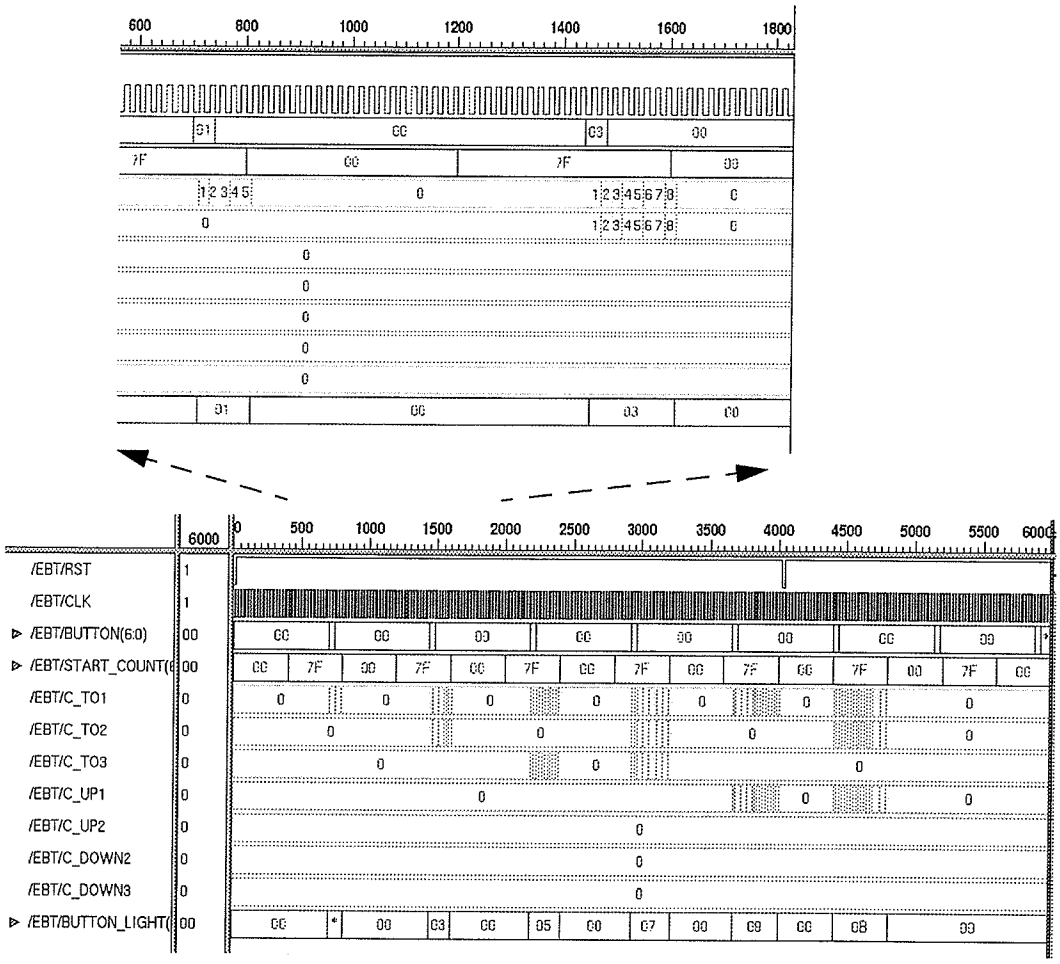


Fig. 4.21: Simulation Results for button_timing_light Module

Simulation results for the *decision_make* module are demonstrated in Figure 4.22. After *rst* is effective, the initial state is *w_1* (waiting on the first floor), when c_to3 is "50" (button to3 is pressed, timing is "50"), the elevator will go up to the third floor, so the elevator state becomes *t13*, consequently, *drive_up* and *light_up* signals are effective. When *ready_to_stop* is '1', the elevator stops, the state becomes *f_3* which means the elevator reaches the third floor, it loads or unloads the passengers. After the *close_done* is '1', the state becomes *w_3*. In this state, c_to1 = "4D" is checked, so the elevator returns to the first floor, the state then becomes *t31*,...

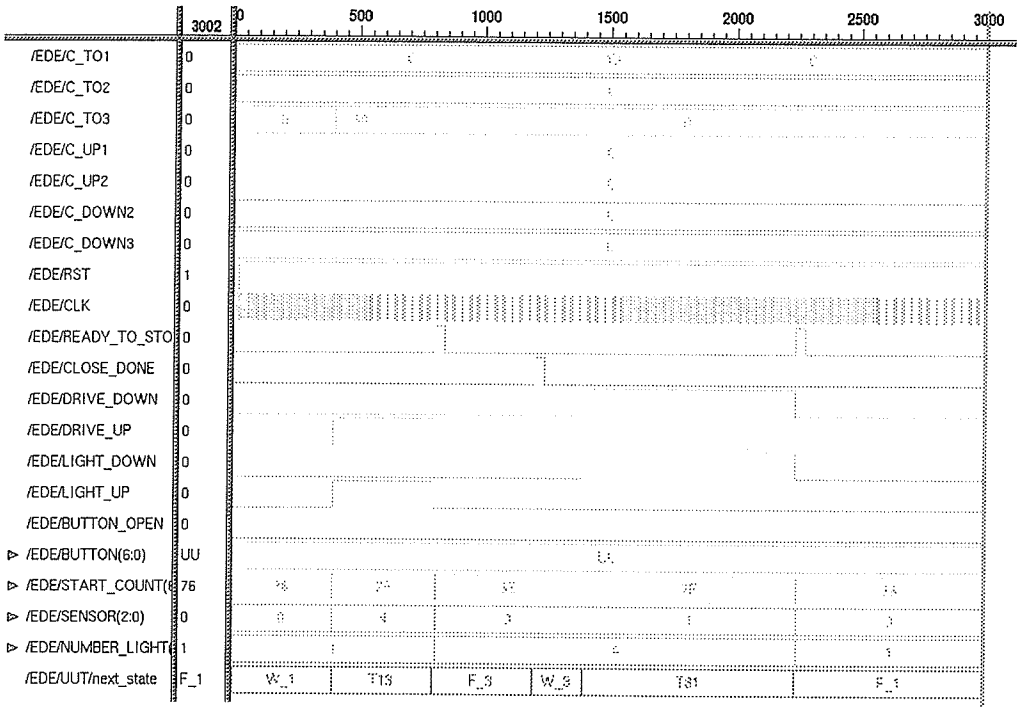


Fig. 4.22: Simulation Results for decision_make Module

Figure 4.23 shows the simulation waveforms for the *elevator_controller* top module. After *rst* is effective, the initial state is *w_1*. When the *button* is "04" (button to3 is pressed), the next state is *t13*, consequently, the *drive_up* and *light_up* signals are effective, and the sensor is 4, which means the destination is the third floor. When *ready_to_stop* is '1', the state becomes *f_3* which means the elevator has reached the third floor. At this point, the door control signals start working. *manual_open*, *open_done*, *manual_close*, and *close_done* work orderly to control the door opening and closing. After the *close_done* is '1', the state becomes *w_3* (elevator is standing at the third floor and waiting for a call). In this state, when *button* is "02" (button to2 is pressed), the elevator goes down to the second floor,...

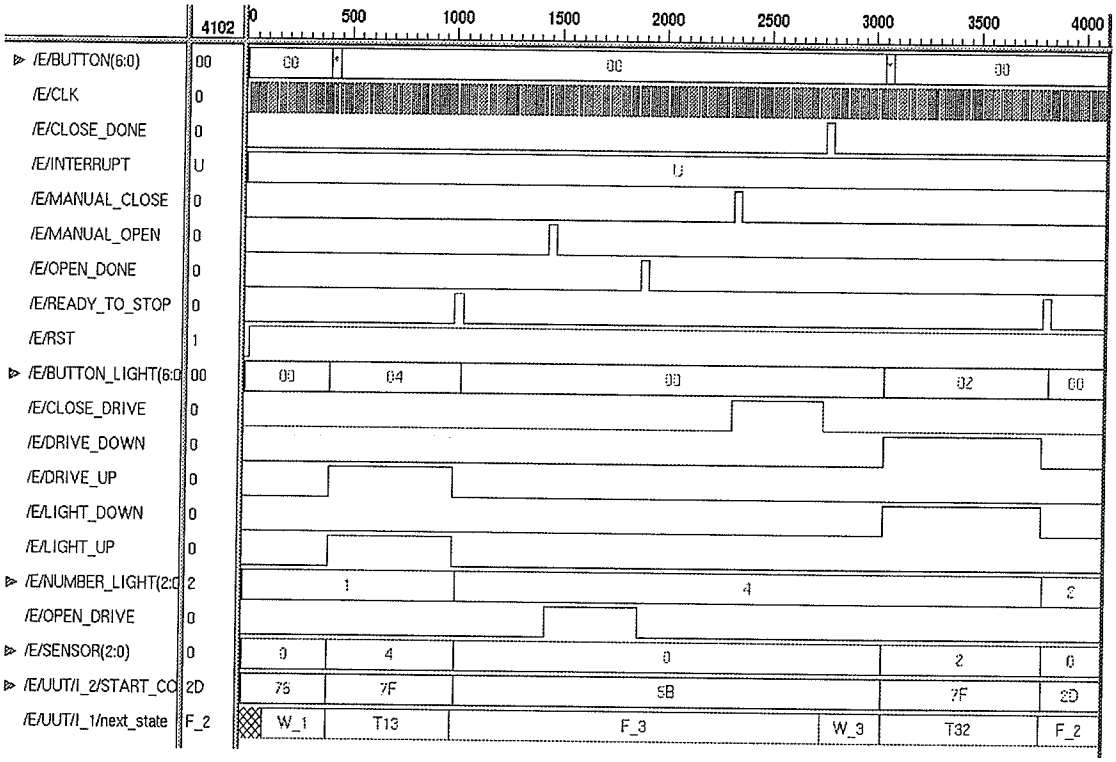


Fig. 4.23: Simulation Result for Elevator Controller Top Module

The simulation waveforms indicate that the design elevator controller work according to its intentions. This type of functional test although is informal, is sufficient for many designs and is largely successful as a direct consequence of a designer's experience. In reality, a design is validated based on the functional tests provided by the designer.

4.9 Summary

Testbenches provide an environment where a design can be stimulated to verify compliance to a specification. Testbenches work as a part of the high level design process. This greatly helps improve productivity, and facilitates simulation of a design at different levels, from RTL to implementation. VHDL-based testbenches take advantage of VHDL features. The

user can choose the style he/she likes to write for different designs. The diversity of test-bench design allows the designer to take advantage of each style. The concurrent approach developed for checking the equivalence of RTL and gate-level representation has proven to be efficient.

A series of behavioral test pattern generation algorithms were presented. There is a correlation between structural-level fault coverage and behavioral-level fault coverage. However, at this time, there are no good standard behavioral fault models available.

CHAPTER 5
CONCLUSIONS AND FUTURE WORK

5.1 Summary and Conclusions

This thesis has presented state-of-the-art methodologies for productive ASIC design and test.

High level design flow addresses the problem of *designing the right system.* It solves this problem by taking advantage of the ability of VHDL to act both as a specification language and as an implementation language which is capable of representing the detailed behavior of a digital ASIC. High and low level descriptions can both be simulated with the same VHDL testbench and simulator. VHDL features are especially suitable for synthesis and simulation.

Logic and optimization tools are used to achieve an almost entirely automatic implementation of the functional description in a specific technology, thus the designer can concentrate on his/her primary task which is to define the circuit's behavior and architecture. Any malfunctions and errors can be identified and eliminated early in the design process.

Chapter 3 presented methodologies typically employed in a high level design flow. The state machine is a very useful method to describe the behavior of the system, and this chapter

demonstrated a very efficient way to derive RTL VHDL code from a Mealy State Transition Diagram. The flow described used the Synopsys tools to get a gate-level VHDL file from the RTL VHDL.

As an important aspect of IC design, IC test strategies were over viewed, and developed from the testbench point of view. Testbench approaches split into two main categories: structural and behavioral. The structural approach is for manufacturing testing purposes, while the behavioral approach is for functional verification. Behavioral testing occurs in the early stages of the design process.

The structural approach was discussed in Chapter 2. Traditional DFT concepts and techniques were reviewed in this chapter. Test synthesis strategies utilize these techniques in an automatic way, (i.e. scan chain insertion) and ATPG can be done by the test synthesis tools. Design flow using the Synopsys Test Compiler tool includes: module level, chip level, and board level testability analysis and design. The ATPG test vectors generated by the Test Compiler can be simulated on both RTL and gate-level descriptions.

Behavioral testbenches can be used to simulate RTL and gate-level descriptions, and to check the equivalence of the two descriptions. Furthermore, mixed level simulation can be done. There are various coding styles (such as behavioral VHDL code, VHDL data structure, and reading test vectors from an external file) for testbenches, and the designer's experience and imagination will suggest the combinations of the styles and other possibilities best suited to a particular application. There are a couple of ways to drive simulation of the optimized gate-level design using the functional testbench for comparison with the result from the RTL description. A novel approach developed by the author combines the RTL and gate-level description as the UUT, and uses a process to compare the responses from the two versions. This approach turned out to be very efficient and accurate. Details were discussed

in Chapter 4.

Testbenches have high reusability, and are applicable for components of different complexity, i.e., for sub-module test and system test. Complex systems are usually partitioned into smaller, less complex modules, thus testbench simulation is done first on the sub-modules to ensure that the functionality of each sub-module is correct, then on the system level to check the system level functionality.

5.2 Future Work

Currently, the Synopsys tools can only do automated structural testing which happens in a later stage of the design process. If the functionality of the system is wrong, there is actually no point in doing so. As a result, future work should focus on raising the level of testbench generation, and automating the functional test vector generation process. First, choosing an algorithm (refer to section 4.7) which is used to derive the test vectors, then writing programs to automate the process of functional test vector generation.

In addition, the clock generation and reset signal generation processes should also be developed in an automatic way.

APPENDIX A

DESIGN EXAMPLE 1: ELEVATOR CONTROLLER

The elevator controller design was selected as a design example as it illustrated many of the design flow and methodologies addressed in the thesis.

A.1 Functional Specifications and Assumptions

The elevator operates in a building with 3 floors and it serves all floors.

In the elevator, there are two arrow lights indicating whether the elevator is moving up or down, three number lights showing which floor the elevator is at, and three number buttons for the users to make requests for which floor they want to reach. On the second floor, there are two arrow buttons to choose for going up or down. On the first floor, there is only a going up arrow button, on the third floor, there is only going down arrow button. Within the elevator, there are two door control buttons---for open and close. The door automatically closes if no one presses the "close" button and no one comes in or out for 5 secs, and also automatically opens if the elevator reaches a required floor but no one presses the "open" button for 3 seconds. When there are people getting in/out of the elevator during the time the door is closing, the state of the door will change to open.

For simplicity, some other assumptions were also made: in particular, buttons were ignored

for emergency or other services. Requests from users standing on the three floors and those inside the elevator are all queued on the basis of their initiated timings, which is a First come, first Served fashion. Some additional signals are also assumed by the designer as shown in A.2.

A.2 Schematic Diagram

The Synopsys SGE schematic and symbolic editors were used to enter the following top level schematic.

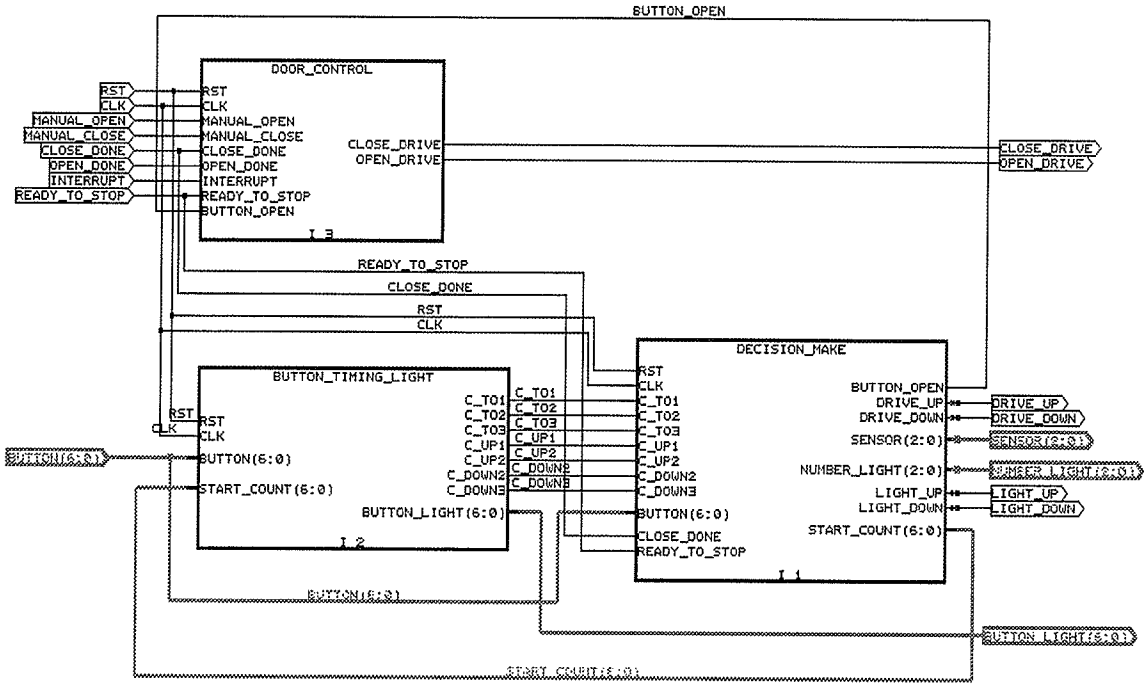


Fig. A.1: Schematic Entry--- Block Diagram for Elevator Controller

A.3 Signal Names and Assignments

Button Names and Assignments are shown in Figure A.2.

System Signals:

Input signals:

manual_open (*manual_close*): open (close) door button signal in the elevator.

open_done (*close_done*): the elevator door has finished opening (closing).

interrupt: when the door is closing, and some one is getting in or out the elevator, a sensor gives this signal to the elevator.

ready_to_stop: the elevator has reached the intended floor.

button (6:0): array of the button signals (Button names and bit assignments are shown in Figure A.2).

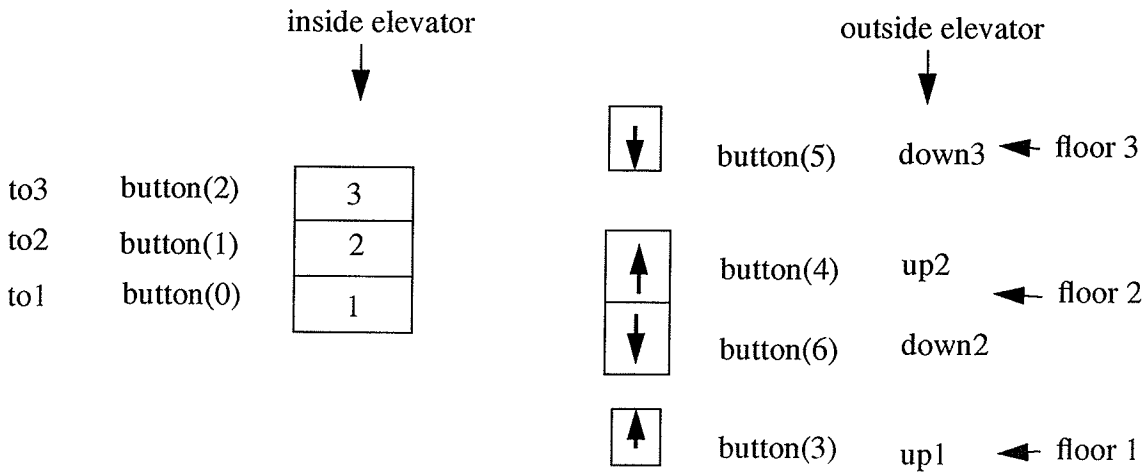


Fig. A.2: Button Signal Name Assignment

Output signals:

drive_up (*drive_down*): the motor gets this signal to drive the elevator up (down).

light_up (*light_down*): the up (down) arrow light which indicates that the elevator is moving

up (down).

open_drive (*close_drive*): the motor gets this signal to drive the elevator door open (close).

button_light *(3:0)*: array of button light signals. If the light is on, someone has made a request.

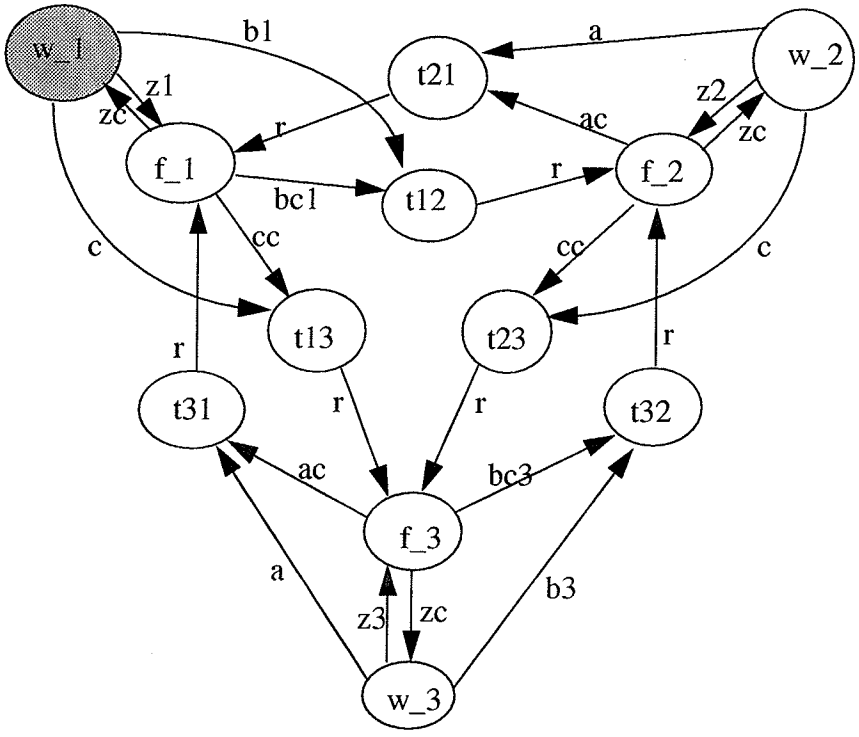
sensor (2:0): array of signals. Indicates the next floor the elevator is going to, e.g., "010" means the destination floor of the elevator is the second floor.

Internal signals:

c_to1: stores the timing of button to1.

start_count (6:0): array of signals for enabling the buttons to count their timing, e.g., *start_count = "1110110"* means start_count signal enables the button to2, up2, down3, and down2 to count their timing, but does not enable button to1 and up1 to count their timing.

A.4 Finite State Machine for the Module *decision-make*:



Decision_make Module ---- Comparison of the Button Timing:

A = max2 (c_to1, c_up1)

B = max3 (c_to2, c_up2, c_down2)

C = max2 (c_to3, c_down3)

X = max3 (A, B, C)

max2 (a, b) is the function for choosing the largest number from a and b. For example, Function max 2 (c_to1, c_up1) is the longer button request timing from button to1 and up1. If X = A, the elevator should go to the first floor; if X = B, the elevator should go to the second floor; If X=C, the elevator should go to the third floor.

Notations for the States:

w_1: elevator is stopped on the first floor, waiting for a call, door is closed (initial state).

f_1: elevator arrived to the first floor, door is going to open, then close.

t12: elevator is moving up, the origin is the first floor, the destination is the second floor.

t32: elevator is moving down, the origin is the third floor, the destination is the second floor.

Notations for the Transitions:

Table 1: Notations for Transitions

transition	condition	action
zc	X=0 and close_done= '1'	
ac	X=A and close_done= '1'	light_down= '1', drive_down= '1' sensor= "001"
bc1	X=B and close_done= '1'	light_up= '1', drive_up= '1' sensor= "010"
bc3	X=B and close_done= '1'	light_down= '1', drive_down= '1' sensor= "010"
cc	x=C and close_done= '1'	light_up= '1', drive_up= '1' sensor= "100"
a	X=A	light_down= '1', drive_down= '1' sensor= "001"
b1	X=B	light_up= '1', drive_up= '1' sensor= "010"
b3	X=B	light_down= '1', drive_down= '1' sensor= "010"
c	X=C	light_up= '1', drive_up= '1' sensor= "100"
z1	X=0 and button(0)= '1' or button(3)= '1'	button_open = '1'
z2	X=0 and button(1)= '1' or button(4)= '1' or button(6)= '1'	button_open = '1'
z3	X=0 and button(3)= '1' or button(5)= '1'	button_open = '1'
r	ready_to_stop	light_up= '0', light_down= '0', drive_up= '0', drive_down= '0'

A.5 VHDL Template Generated by SGE

Synopsys SGE can generate structural VHDL and the testbench template for the top level design, and generate behavioral VHDL templates for every bottom level subdesign. These are illustrated below.

4.5.1 Structural VHDL for Top Design

```
-- VHDL Model Created from SGE Schematic top.sch -- Jun  5 19:53:58 1996

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_misc.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_components.all;

entity TOP is
    Port ( BUTTON : In    std_logic_vector (6 downto 0);
           CLK : In    STD_LOGIC;
       CLOSE_DONE : In    STD_LOGIC;
       INTERRUPT : In    STD_LOGIC;
       MANUAL_CLOSE : In    STD_LOGIC;
       MANUAL_OPEN : In    STD_LOGIC;
       OPEN_DONE : In    STD_LOGIC;
       READY_TO_STOP : In    STD_LOGIC;
         RST : In    STD_LOGIC;
       BUTTON_LIGHT : Out   std_logic_vector (6 downto 0);
       CLOSE_DRIVE : Out   STD_LOGIC;
       DRIVE_DOWN : Out   STD_LOGIC;
       DRIVE_UP : Out   STD_LOGIC;
       LIGHT_DOWN : Out   STD_LOGIC;
       LIGHT_UP : Out   STD_LOGIC;
       NUMBER_LIGHT : Out   std_logic_vector (2 downto 0);
       OPEN_DRIVE : Out   STD_LOGIC;
        SENSOR : Out   std_logic_vector (2 downto 0) );
end TOP;

architecture SCHEMATIC of TOP is

  signal START_COUNT : std_logic_vector(6 downto 0);
  signal  C_DOWN2 : INTEGER RANGE 0 TO 511;
  signal  C_DOWN3 : INTEGER RANGE 0 TO 511;
  signal   C_UP2 : INTEGER RANGE 0 TO 511;
  signal   C_UP1 : INTEGER RANGE 0 TO 511;
  signal   C_TO3 : INTEGER RANGE 0 TO 511;
  signal   C_TO2 : INTEGER RANGE 0 TO 511;
  signal   C_TO1 : INTEGER RANGE 0 TO 511;
  signal  button_open: std_logic;

  component DECISION_MAKE
    Port ( BUTTON : In    std_logic_vector (6 downto 0);
         C_DOWN2 : In    INTEGER RANGE 0 TO 511;
         C_DOWN3 : In    INTEGER RANGE 0 TO 511;
          C_TO1 : In    INTEGER RANGE 0 TO 511;
          C_TO2 : In    INTEGER RANGE 0 TO 511;
```

```
          C_TO3 : In   INTEGER RANGE 0 TO 511;
          C_UP1 : In   INTEGER RANGE 0 TO 511;
          C_UP2 : In   INTEGER RANGE 0 TO 511;
           CLK : In   STD_LOGIC;
        CLOSE_DONE : In   STD_LOGIC;
        READY_TO_STOP : In   STD_LOGIC;
          RST : In   STD_LOGIC;
        DRIVE_DOWN : Out  STD_LOGIC;
        DRIVE_UP : Out  STD_LOGIC;
        LIGHT_DOWN : Out  STD_LOGIC;
        LIGHT_UP : Out  STD_LOGIC;
        button_OPEN : Out  STD_LOGIC;
        NUMBER_LIGHT : Out  std_logic_vector (2 downto 0);
         SENSOR : Out  std_logic_vector (2 downto 0);
        START_COUNT : Out  std_logic_vector (6 downto 0) );
  end component;

  component BUTTON_TIMING_LIGHT
    Port ( BUTTON : In   std_logic_vector (6 downto 0);
           CLK : In   STD_LOGIC;
           RST : In   STD_LOGIC;
        START_COUNT : In   std_logic_vector (6 downto 0);
        BUTTON_LIGHT : Out  std_logic_vector (6 downto 0);
        C_DOWN2 : Out  INTEGER RANGE 0 TO 511;
        C_DOWN3 : Out  INTEGER RANGE 0 TO 511;
         C_TO1 : Out  INTEGER RANGE 0 TO 511;
         C_TO2 : Out  INTEGER RANGE 0 TO 511;
         C_TO3 : Out  INTEGER RANGE 0 TO 511;
         C_UP1 : Out  INTEGER RANGE 0 TO 511;
         C_UP2 : Out  INTEGER RANGE 0 TO 511 );
  end component;

  component DOOR_CONTROL
    Port (   CLK : In   STD_LOGIC;
        CLOSE_DONE : In   STD_LOGIC;
        INTERRUPT : In   STD_LOGIC;
        MANUAL_CLOSE : In   STD_LOGIC;
        MANUAL_OPEN : In   STD_LOGIC;
button_open: in std_logic;
        OPEN_DONE : In   STD_LOGIC;
        READY_TO_STOP : In   STD_LOGIC;
          RST : In   STD_LOGIC;
        CLOSE_DRIVE : Out  STD_LOGIC;
        OPEN_DRIVE : Out  STD_LOGIC );
  end component;

begin

  I_1 : DECISION_MAKE
    Port Map ( BUTTON(6 downto 0)=>BUTTON(6 downto 0),
           C_DOWN2=>C_DOWN2, C_DOWN3=>C_DOWN3, C_TO1=>C_TO1,
           C_TO2=>C_TO2, C_TO3=>C_TO3, C_UP1=>C_UP1, C_UP2=>C_UP2,
           CLK=>CLK, CLOSE_DONE=>CLOSE_DONE,
           READY_TO_STOP=>READY_TO_STOP, RST=>RST,
           DRIVE_DOWN=>DRIVE_DOWN, DRIVE_UP=>DRIVE_UP,
           LIGHT_DOWN=>LIGHT_DOWN, LIGHT_UP=>LIGHT_UP,
           button_OPEN=>button_OPEN,
           NUMBER_LIGHT(2 downto 0)=>NUMBER_LIGHT(2 downto 0),
           SENSOR(2 downto 0)=>SENSOR(2 downto 0),
           START_COUNT(6 downto 0)=>START_COUNT(6 downto 0) );
  I_2 : BUTTON_TIMING_LIGHT
    Port Map ( BUTTON(6 downto 0)=>BUTTON(6 downto 0), CLK=>CLK,
           RST=>RST,
           START_COUNT(6 downto 0)=>START_COUNT(6 downto 0),
           BUTTON_LIGHT(6 downto 0)=>BUTTON_LIGHT(6 downto 0),
```

```
            C_DOWN2=>C_DOWN2, C_DOWN3=>C_DOWN3, C_TO1=>C_TO1,
            C_TO2=>C_TO2, C_TO3=>C_TO3, C_UP1=>C_UP1, C_UP2=>C_UP2 );
    I_3 : DOOR_CONTROL
      Port Map ( CLK=>CLK, CLOSE_DONE=>CLOSE_DONE, INTERRUPT=>INTERRUPT,
            MANUAL_CLOSE=>MANUAL_CLOSE, MANUAL_OPEN=>MANUAL_OPEN,
  button_open=>button_open,
            OPEN_DONE=>OPEN_DONE, READY_TO_STOP=>READY_TO_STOP,
            RST=>RST, CLOSE_DRIVE=>CLOSE_DRIVE,
            OPEN_DRIVE=>OPEN_DRIVE );

  end SCHEMATIC;

  configuration CFG_TOP_SCHEMATIC of TOP is

    for SCHEMATIC
      for I_1: DECISION_MAKE
        use configuration WORK.CFG_DECISION_MAKE_BEHAVIORAL;
      end for;
      for I_2: BUTTON_TIMING_LIGHT
        use configuration WORK.CFG_BUTTON_TIMING_LIGHT_BEHAVIORAL;
      end for;
      for I_3: DOOR_CONTROL
        use configuration WORK.CFG_DOOR_CONTROL_BEHAVIORAL;
      end for;
    end for;

  end CFG_TOP_SCHEMATIC;
```

4.5.2 Behavioral VHDL Template for Subdesign ---- *door_control* block

```
-- VHDL Model Created from SGE Symbol door_control.sym -- Jun 10 15:55:22 1996

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_misc.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_components.all;

entity DOOR_CONTROL is
    Port ( BUTTON_OPEN : In    STD_LOGIC;
           CLK : In    STD_LOGIC;
         CLOSE_DONE : In    STD_LOGIC;
         INTERRUPT : In    STD_LOGIC;
         MANUAL_CLOSE : In    STD_LOGIC;
         MANUAL_OPEN : In    STD_LOGIC;
         OPEN_DONE : In    STD_LOGIC;
         READY_TO_STOP : In    STD_LOGIC;
           RST : In    STD_LOGIC;
         CLOSE_DRIVE : Out   STD_LOGIC;
         OPEN_DRIVE : Out   STD_LOGIC );
end DOOR_CONTROL;

architecture BEHAVIORAL of DOOR_CONTROL is

  begin

end BEHAVIORAL;

configuration CFG_DOOR_CONTROL_BEHAVIORAL of DOOR_CONTROL is
  for BEHAVIORAL
```

```
  end for;

end CFG_DOOR_CONTROL_BEHAVIORAL;
```

4.5.3 Testbench Template for Top Design

```
-- VHDL Test Bench Created from SGE Symbol top.sym.sym -- Jun 10 15:55:22 1996

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_misc.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_components.all;

entity E is
end E;

Architecture A of E is

  signal  BUTTON : std_logic_vector (6 downto 0);
  signal    CLK : STD_LOGIC;
  signal CLOSE_DONE : STD_LOGIC;
  signal INTERRUPT : STD_LOGIC;
  signal MANUAL_CLOSE : STD_LOGIC;
  signal MANUAL_OPEN : STD_LOGIC;
  signal OPEN_DONE : STD_LOGIC;
  signal READY_TO_STOP : STD_LOGIC;
  signal    RST : STD_LOGIC;
  signal BUTTON_LIGHT : std_logic_vector (6 downto 0);
  signal CLOSE_DRIVE : STD_LOGIC;
  signal DRIVE_DOWN : STD_LOGIC;
  signal DRIVE_UP : STD_LOGIC;
  signal LIGHT_DOWN : STD_LOGIC;
  signal LIGHT_UP : STD_LOGIC;
  signal NUMBER_LIGHT : std_logic_vector (2 downto 0);
  signal OPEN_DRIVE : STD_LOGIC;
  signal  SENSOR : std_logic_vector (2 downto 0);

  component TOP
    Port (  BUTTON : In   std_logic_vector (6 downto 0);
            CLK : In   STD_LOGIC;
         CLOSE_DONE : In   STD_LOGIC;
         INTERRUPT : In   STD_LOGIC;
         MANUAL_CLOSE : In   STD_LOGIC;
         MANUAL_OPEN : In   STD_LOGIC;
         OPEN_DONE : In   STD_LOGIC;
         READY_TO_STOP : In   STD_LOGIC;
           RST : In   STD_LOGIC;
         BUTTON_LIGHT : Out   std_logic_vector (6 downto 0);
         CLOSE_DRIVE : Out   STD_LOGIC;
         DRIVE_DOWN : Out   STD_LOGIC;
         DRIVE_UP : Out   STD_LOGIC;
         LIGHT_DOWN : Out   STD_LOGIC;
         LIGHT_UP : Out   STD_LOGIC;
         NUMBER_LIGHT : Out   std_logic_vector (2 downto 0);
         OPEN_DRIVE : Out   STD_LOGIC;
          SENSOR : Out   std_logic_vector (2 downto 0) );
  end component;

begin
  UUT : TOP
```

```
    Port Map ( BUTTON, CLK, CLOSE_DONE, INTERRUPT, MANUAL_CLOSE,
          MANUAL_OPEN, OPEN_DONE, READY_TO_STOP, RST,
          BUTTON_LIGHT, CLOSE_DRIVE, DRIVE_DOWN, DRIVE_UP,
          LIGHT_DOWN, LIGHT_UP, NUMBER_LIGHT, OPEN_DRIVE,
          SENSOR );

-- *** Test Bench - User Defined Section ***
  TB : block
  begin


  end block;
-- *** End Test Bench - User Defined Section ***

end A;

configuration CFG_TB_TOP_BEHAVIORAL of E is
  for A
    for UUT : TOP
      use configuration WORK.CFG_TOP_SCHEMATIC;
    end for;

-- *** User Defined Configuration ***
    for TB
    end for;
-- *** End User Defined Configuration ***

  end for;
end CFG_TB_TOP_BEHAVIORAL;
```

A.6 Behavioral VHDL Source Code for Subdesigns

4.6.1 *door_control* Module

```
-- VHDL Model Created from SGE Symbol door_control.sym -- Jun  5 18:57:57 1996

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_misc.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_components.all;

entity DOOR_CONTROL is
    Port (    CLK : In   STD_LOGIC;
        CLOSE_DONE : In   STD_LOGIC;
        INTERRUPT : In   STD_LOGIC;
        MANUAL_CLOSE : In   STD_LOGIC;
        MANUAL_OPEN : In   STD_LOGIC;
button_open: in STD_LOGIC;
        OPEN_DONE : In   STD_LOGIC;
        READY_TO_STOP : In   STD_LOGIC;
          RST : In   STD_LOGIC;
        CLOSE_DRIVE : Out  STD_LOGIC;
        OPEN_DRIVE : Out  STD_LOGIC );
end DOOR_CONTROL;

architecture BEHAVIORAL of DOOR_CONTROL is

signal current_state, next_state: std_logic_vector(1 downto 0);
```

```
signal keep:std_logic;
begin


process(rst,clk,ready_to_stop, interrupt, manual_open,button_open, manual_close,      close_done, open_done, current_state)

variable count, count1: integer range 0 to 10000;

begin

--default assignment


--sychronous reset

if (rst='0') then
next_state<="00";
close_drive<='0';
open_drive<='0';
count:=0;
count1:=0;
keep<='0';

else

 -- state transition and output logic
case current_state is
when "00" =>if( ready_to_stop='1') then
keep<='1';
end if;
if (keep<='1')then
if(manual_open='1'or button_open='1') then
close_drive<='0';
  open_drive<='1';
next_state<="01";
else
count:=count+1;
if (count>1000) then
close_drive<='0';
  open_drive<='1';
next_state<="01";
count:=0;
end if;
end if;
end if;


when "01" =>keep<='0';
if (open_done='1')then
close_drive<='0';
   open_drive<='0';
next_state<="10";
  end if;

when "10" => keep<='0';
if ( manual_close='1') then
 close_drive<='1';
  open_drive<='0';
next_state<="11";
else
count1:=count1+1;
if (count>1000) then
close_drive<='1';
  open_drive<='0';
next_state<="11";
```

```
 count1:=0;
 end if;
   end if;

 when "11" =>keep<='0';
 if (close_done='1') then
 close_drive<='0';
       open_drive<='0';
  next_state<="00";

   else
 if( interrupt='1' or manual_open='1') then
 close_drive<='0';
    open_drive<='1';
 next_state<="01";
 end if;
   end if;

 when others => null;

 end case;
 end if;

 end process;

 ----synchronize state value with clock
 process(clk)
 begin
 if (clk'event and clk='1') then
 current_state<=next_state;
  end if;
 end process;

 end BEHAVIORAL;

 configuration CFG_DOOR_CONTROL_BEHAVIORAL of DOOR_CONTROL is
   for BEHAVIORAL

   end for;

 end CFG_DOOR_CONTROL_BEHAVIORAL;
```

4.6.2 *button_timing_light* Module

```
-- VHDL Model Created from SGE Symbol button_timing_light.sym -- Jun  5 19:53:57 1996

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_misc.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_components.all;

entity BUTTON_TIMING_LIGHT is
    Port (  BUTTON : In    std_logic_vector (6 downto 0);
            CLK : In    STD_LOGIC;
            RST : In    STD_LOGIC;
        START_COUNT : In    std_logic_vector (6 downto 0);
        BUTTON_LIGHT : Out   std_logic_vector (6 downto 0);
        C_DOWN2 : Out   INTEGER RANGE 0 TO 511;
        C_DOWN3 : Out   INTEGER RANGE 0 TO 511;
```

```
        C_TO1 : Out   INTEGER RANGE 0 TO 511;
        C_TO2 : Out   INTEGER RANGE 0 TO 511;
        C_TO3 : Out   INTEGER RANGE 0 TO 511;
        C_UP1 : Out   INTEGER RANGE 0 TO 511;
        C_UP2 : Out   INTEGER RANGE 0 TO 511 );
end BUTTON_TIMING_LIGHT;

architecture BEHAVIORAL of BUTTON_TIMING_LIGHT is

signal keep0, keep1, keep2,keep3,keep4,keep5,keep6: std_logic;

  begin

process(rst,clk,button,start_count)
variable  cc_to1, cc_to2,cc_to3,cc_up1,cc_up2,cc_down2,cc_down3: integer range 0 to 511;
begin

------default assignments

------synchronous reset

if (rst='0') then
c_to1<=0;
c_to2<=0;
c_to3<=0;
c_up1<=0;
c_up2<=0;
c_down2<=0;
c_down3<=0;
button_light<="0000000";
keep0<='0';
keep1<='0';
keep2<='0';
keep3<='0';
keep4<='0';
keep5<='0';
keep6<='0';

elsif (clk'event and clk='1') then


if (start_count(0)='1') then
if (button(0)='1')then
keep0<='1';
cc_to1:=cc_to1+1;
button_light(0)<='1';
else
if (keep0='1')then
cc_to1:=cc_to1+1;
button_light(0)<='1';
end if;
end if;
else
keep0<='0';
cc_to1:=0;
button_light(0)<='0';
end if;


if (start_count(1)='1') then
if (button(1)='1')then
keep1<='1';
cc_to2:=cc_to2+1;
button_light(1)<='1';
else
```

```
if (keep1='1')then
cc_to2:=cc_to2+1;
button_light(1)<='1';
end if;
end if;
else
keep1<='0';
cc_to2:=0;
button_light(1)<='0';
end if;


if (start_count(2)='1') then
if (button(2)='1')then
keep2<='1';
cc_to3:=cc_to3+1;
button_light(2)<='1';
else
if (keep2='1')then
cc_to3:=cc_to3+1;
button_light(2)<='1';
end if;
end if;
else
keep2<='0';
cc_to3:=0;
button_light(2)<='0';
end if;

if (start_count(3)='1') then
if (button(3)='1')then
keep3<='1';
cc_up1:=cc_up1+1;
button_light(3)<='1';
else
if (keep3='1')then
cc_up1:=cc_up1+1;
button_light(3)<='1';
end if;
end if;
else
keep3<='0';
cc_up1:=0;
button_light(3)<='0';
end if;

if (start_count(4)='1') then
if (button(4)='1')then
keep4<='1';
cc_up2:=cc_up2+1;
button_light(4)<='1';
else
if (keep4='1')then
cc_up2:=cc_up2+1;
button_light(4)<='1';
end if;
end if;
else
keep4<='0';
cc_up2:=0;
button_light(4)<='0';
end if;


if (start_count(5)='1') then
```

```
 if (button(5)='1')then
 keep5<='1';
 cc_down3:=cc_down3+1;
 button_light(5)<='1';
 else
 if (keep5='1')then
 cc_down3:=cc_down3+1;
 button_light(5)<='1';
 end if;
 end if;
 else
 keep5<='0';
 cc_down3:=0;
 button_light(5)<='0';
 end if;

 if (start_count(6)='1') then
 if (button(6)='1')then
 keep6<='1';
 cc_down2:=cc_down2+1;
 button_light(6)<='1';
 else
 if (keep6='1')then
 cc_down2:=cc_down2+1;
 button_light(6)<='1';
 end if;
 end if;
 else
 keep6<='0';
 cc_down2:=0;
 button_light(6)<='0';
 end if;

 c_to1<=cc_to1;
 c_to2<=cc_to2;
 c_to3<=cc_to3;
 c_up1<=cc_up1;
 c_up2<=cc_up2;
 c_down3<=cc_down3;
 c_down2<=cc_down2;

 end if;

 end process;

 end BEHAVIORAL;

 configuration CFG_BUTTON_TIMING_LIGHT_BEHAVIORAL of BUTTON_TIMING_LIGHT is
   for BEHAVIORAL

   end for;

end CFG_BUTTON_TIMING_LIGHT_BEHAVIORAL;
```

4.6.3 *decision_make* Module

```
-- VHDL Model Created from SGE Symbol decision_make.sym -- Jun  5 19:53:57 1996

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_misc.all;
```

```
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_components.all;

package MAX is
function max2(arg1,arg2: integer) return integer;
function max3(arg1,arg2,arg3: integer) return integer;
function max4(arg1,arg2,arg3,arg4: integer ) return integer ;
end MAX;

package body MAX is
function max2(arg1,arg2: integer) return integer is
variable arg: integer;
begin
arg:=arg1;
if ( arg1<arg2) then
arg:=arg2;
end if;
return arg;
end ;

function max3(arg1,arg2,arg3: integer ) return integer  is
variable arg,arg12: integer;
begin
arg12:=arg1;
if ( arg1<arg2) then
arg12:=arg2;
end if;
arg:=arg12;
if (arg12<arg3) then
arg:=arg3;
end if;
return arg;
end ;

function max4(arg1,arg2,arg3,arg4: integer) return integer is
variable arg,arg12,arg34: integer;
begin
arg12:=arg1;
arg34:=arg3;
if ( arg1<arg2) then
arg12:=arg2;
end if;
if (arg3<arg4) then
arg34:=arg4;
end if;
arg:=arg12;
if (arg12<arg34) then
arg:=arg34;
end if;
return arg;
end ;

end MAX;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.MAX.all;

entity DECISION_MAKE is
    Port ( BUTTON : In    std_logic_vector (6 downto 0);
           C_DOWN2 : In    INTEGER RANGE 0 TO 511;
           C_DOWN3 : In    INTEGER RANGE 0 TO 511;
            C_TO1 : In    INTEGER RANGE 0 TO 511;
            C_TO2 : In    INTEGER RANGE 0 TO 511;
```

```
        C_TO3 : In   INTEGER RANGE 0 TO 511;
        C_UP1 : In   INTEGER RANGE 0 TO 511;
        C_UP2 : In   INTEGER RANGE 0 TO 511;
         CLK : In   STD_LOGIC;
      CLOSE_DONE : In   STD_LOGIC;
      READY_TO_STOP : In   STD_LOGIC;
        RST : In   STD_LOGIC;
      DRIVE_DOWN : Out  STD_LOGIC;
      DRIVE_UP : Out  STD_LOGIC;
      LIGHT_DOWN : Out  STD_LOGIC;
      LIGHT_UP : Out  STD_LOGIC;
      BUTTON_OPEN : Out  STD_LOGIC;
      NUMBER_LIGHT : Out  std_logic_vector (2 downto 0);
       SENSOR : Out  std_logic_vector (2 downto 0);
      START_COUNT : Out  std_logic_vector (6 downto 0) );
end DECISION_MAKE;

architecture BEHAVIORAL of DECISION_MAKE is

type state_type is (f_1,w_1,f_2,w_2,f_3,w_3,t12,t21,t13,t31,t23,t32);

signal current_state,next_state: state_type;

  begin

process( c_to1, c_to2,c_to3,c_up1,c_up2,c_down2,c_down3, rst,clk,READY_TO_STOP,CLOSE_DONE,button,current_state)

variable A,B,C,X: integer;

begin

--default assignment

next_state<=current_state;

--sychronous reset
if (rst='0')then
next_state<=w_1;
LIGHT_UP<='0';
LIGHT_DOWN<='0';
DRIVE_DOWN<='0';
DRIVE_UP<='0';
start_count<="1110110";
sensor<="000";
NUMBER_LIGHT<="001";
BUTTON_OPEN<='0';

else

A:=max2(c_to1,c_up1);
B:=max3(c_to2,c_up2,c_down2);
C:=max2(c_to3, c_down3);
X:=max3(A,B,C);

-- state transition and output logic

case current_state is

when f_1 =>start_count(0)<='0';
start_count(3)<='0';
LIGHT_UP<='0';
LIGHT_DOWN<='0';
DRIVE_DOWN<='0';
DRIVE_UP<='0';
sensor<="000";
```

```
NUMBER_LIGHT<="001";
button_OPEN<='0';

if (X=0 and close_done='1')then
next_state<=w_1;
else
if (X=B and close_done='1') then
next_state<=t12;
start_count(0)<='1';
start_count(3)<='1';
LIGHT_UP<='1';
LIGHT_DOWN<='0';
DRIVE_UP<='1';
DRIVE_DOWN<='0';
sensor<="010";
end if;

if (X=C and close_done='1') then
start_count(0)<='1';
start_count(3)<='1';
next_state<=t13;
LIGHT_UP<='1';
LIGHT_DOWN<='0';
DRIVE_UP<='1';
DRIVE_DOWN<='0';
NUMBER_LIGHT<="001";
sensor<="100";

end if;
end if;


when w_1 =>

 start_count(0)<='0';
start_count(3)<='0';
LIGHT_UP<='0';
LIGHT_DOWN<='0';
DRIVE_DOWN<='0';
DRIVE_UP<='0';
sensor<="000";
NUMBER_LIGHT<="001";
button_OPEN<='0';

if (X=0)then

if ( (button(0)='1') or (button(3)='1')) then
next_state<=f_1;
button_open<='1';
end if;
else

if (X=B ) then
next_state<=t12;
start_count(0)<='1';
start_count(3)<='1';
LIGHT_UP<='1';
LIGHT_DOWN<='0';
DRIVE_UP<='1';
DRIVE_DOWN<='0';
sensor<="010";
end if;

if (X=C ) then
start_count(0)<='1';
```

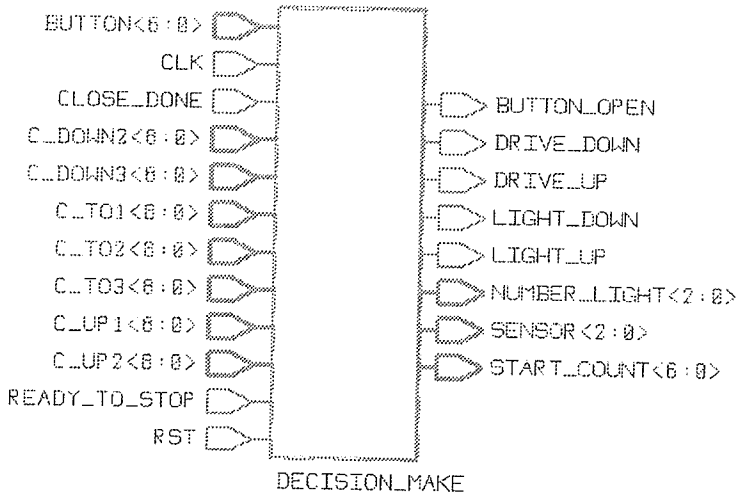
```
start_count(3)<='1';
next_state<=t13;
LIGHT_UP<='1';
LIGHT_DOWN<='0';
DRIVE_UP<='1';
DRIVE_DOWN<='0';
sensor<="100";
end if;
end if;


when f_2 =>start_count(1)<='0';
start_count(4)<='0';
start_count(6)<='0';
LIGHT_DOWN<='0';
DRIVE_DOWN<='0';
DRIVE_UP<='0';
sensor<="000";
NUMBER_LIGHT<="010";
button_OPEN<='0';

if (X=0 and close_done='1')then
next_state<=w_2;
else
if (X=A and close_done='1' ) then
next_state<=t21;
start_count(1)<='1';
start_count(4)<='1';
start_count(6)<='1';
LIGHT_UP<='0';
LIGHT_DOWN<='1';
DRIVE_UP<='0';
DRIVE_DOWN<='1';
sensor<="001";
end if;

if (X=C and close_done='1') then
start_count(1)<='1';
start_count(4)<='1';
start_count(6)<='1';
next_state<=t23;
LIGHT_UP<='1';
LIGHT_DOWN<='0';
DRIVE_UP<='1';
DRIVE_DOWN<='0';
sensor<="100";
end if;
end if;


when w_2 =>start_count(1)<='0';
start_count(4)<='0';
start_count(6)<='0';

LIGHT_UP<='0';
LIGHT_DOWN<='0';
DRIVE_DOWN<='0';
DRIVE_UP<='0';
sensor<="000";
NUMBER_LIGHT<="010";
button_OPEN<='0';


if (X=0)then
if ( (button(1)='1') or (button(4)='1')or (button(6)='1')) then
```

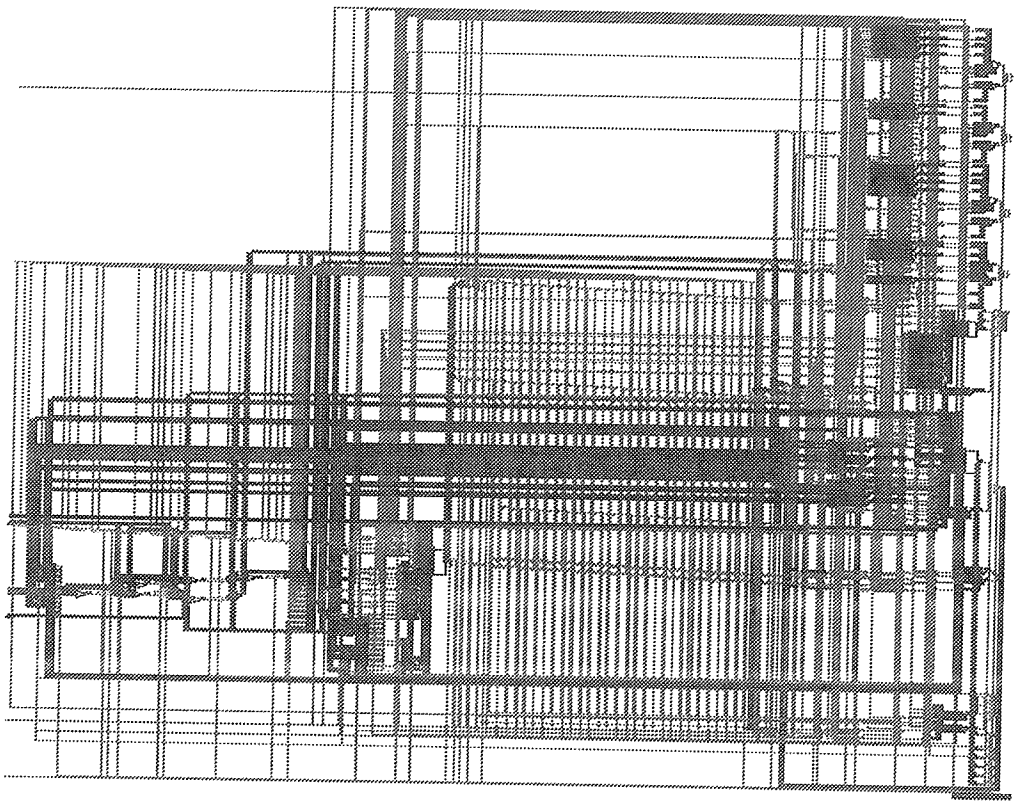
```
next_state<=f_2;
button_open<='1';
end if;
else
if (X=A ) then
next_state<=t21;
start_count(1)<='1';
start_count(4)<='1';
start_count(6)<='1';
LIGHT_UP<='0';
LIGHT_DOWN<='1';
DRIVE_UP<='0';
DRIVE_DOWN<='1';
sensor<="001";
end if;

if (X=C ) then
start_count(1)<='1';
start_count(4)<='1';
start_count(6)<='1';
next_state<=t23;
LIGHT_UP<='1';
LIGHT_DOWN<='0';
DRIVE_UP<='1';
DRIVE_DOWN<='0';
sensor<="100";
end if;
end if;


when f_3 =>start_count(2)<='0';
start_count(5)<='0';
LIGHT_UP<='0';
LIGHT_DOWN<='0';
DRIVE_DOWN<='0';
DRIVE_UP<='0';
sensor<="000";
NUMBER_LIGHT<="100";
button_OPEN<='0';

if (X=0 and close_done='1')then
next_state<=w_3;
else


if (X=B and close_done='1') then
next_state<=t32;
start_count(2)<='1';
start_count(5)<='1';
LIGHT_UP<='0';
LIGHT_DOWN<='1';
DRIVE_UP<='0';
DRIVE_DOWN<='1';
sensor<="010";
end if;

if (X=A and close_done='1') then
start_count(2)<='1';
start_count(5)<='1';
next_state<=t31;
LIGHT_UP<='0';
LIGHT_DOWN<='1';
DRIVE_UP<='0';
DRIVE_DOWN<='1';
sensor<="001";
```

```
end if;
end if;



when w_3 =>start_count(2)<='0';
start_count(5)<='0';
LIGHT_UP<='0';
LIGHT_DOWN<='0';
DRIVE_DOWN<='0';
DRIVE_UP<='0';
sensor<="000";
NUMBER_LIGHT<="100";
button_OPEN<='0';



if (X=0) then

if ( (button(2)='1') or (button(5)='1')) then
next_state<=f_3;
button_open<='1';
end if;
else

if (X=B ) then
next_state<=t32;
start_count(2)<='1';
start_count(5)<='1';
LIGHT_UP<='0';
LIGHT_DOWN<='1';
DRIVE_UP<='0';
DRIVE_DOWN<='1';
sensor<="010";
end if;

if (X=A ) then
start_count(2)<='1';
start_count(5)<='1';
next_state<=t31;
LIGHT_UP<='0';
LIGHT_DOWN<='1';
DRIVE_UP<='0';
DRIVE_DOWN<='1';
sensor<="001";
end if;
end if;


when t12 => start_count<="1111111";
if (ready_to_stop='1') then
next_state<=f_2;
LIGHT_UP<='0';
LIGHT_DOWN<='0';
DRIVE_UP<='0';
DRIVE_DOWN<='0';
end if;

when t21 => start_count<="1111111";
if (ready_to_stop='1') then
next_state<=f_1;
LIGHT_UP<='0';
LIGHT_DOWN<='0';
DRIVE_UP<='0';
DRIVE_DOWN<='0';
```

```
    end if;

    when t13 => start_count<="1111111";
    if (ready_to_stop='1') then
    next_state<=f_3;
    LIGHT_UP<='0';
    LIGHT_DOWN<='0';
    DRIVE_UP<='0';
    DRIVE_DOWN<='0';
    end if;
    when t31 => start_count<="1111111";
    if (ready_to_stop='1') then
    next_state<=f_1;
    LIGHT_UP<='0';
    LIGHT_DOWN<='0';
    DRIVE_UP<='0';
    DRIVE_DOWN<='0';
    end if;

    when t23 => start_count<="1111111";
    if (ready_to_stop='1') then
    next_state<=f_3;
    LIGHT_UP<='0';
    LIGHT_DOWN<='0';
    DRIVE_UP<='0';
    DRIVE_DOWN<='0';
    end if;

    when t32 => start_count<="1111111";

    if (ready_to_stop='1') then
    next_state<=f_2;
    LIGHT_UP<='0';
    LIGHT_DOWN<='0';
    DRIVE_UP<='0';
    DRIVE_DOWN<='0';
    end if;

    when others=>null;
    end case;
    end if;

    end process;

    process(clk)
    begin

    if (clk'event and clk='1') then
    current_state<=next_state;
    end if;
    end process;



  end BEHAVIORAL;

  configuration CFG_DECISION_MAKE_BEHAVIORAL of DECISION_MAKE is
    for BEHAVIORAL

    end for;

  end CFG_DECISION_MAKE_BEHAVIORAL;
```

A.7 Synthesized Schematic of Module *decision_make*





APPENDIX B

DESIGN EXAMPLE 2: ALU FUNCTION GENERATOR DM74181

B.1 Behavioral Description

Refer to [24], p5-100--p5-107.

B.2 Source Code for the ALU

```
------------------------------------------------------------
------- VHDL behavioral code for alu (chip DM74181)
------- Written by Ruomei Wang, March 10, 1996

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

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

package ART00 is
function "+"(arg1,arg2:std_logic_vector(3 downto 0)) return std_logic_vector;
function "-" ( arg1,arg2:std_logic_vector(3 downto 0)) return std_logic_vector;
function cy( arg1,arg2:std_logic_vector(3 downto 0)) return std_logic;
function cn( arg1,arg2:std_logic_vector(3 downto 0)) return std_logic;
end ART00;

package body ART00 is

function "+" (arg1,arg2:std_logic_vector(3 downto 0)) return std_logic_vector is
variable sum:std_logic_vector(3 downto 0);
variable carry:std_logic;
begin
carry:='0';
for i in 0 to 3 loop
sum(i):=arg1(i) xor arg2(i) xor carry;
```

```
carry:=(arg1(i)and arg2(i)) or (arg1(i)and carry) or (carry and arg2(i));
end loop;
return sum;
end;

function "-"(arg1,arg2:std_logic_vector(3 downto 0)) return std_logic_vector is
variable sum,arg22:std_logic_vector(3 downto 0);
variable carry:std_logic;
begin
carry:='1';
arg22:=not(arg2);
for i in 0 to 3 loop
sum(i):=arg1(i) xor arg22(i) xor carry;
carry:=(arg1(i)and arg22(i)) or (arg1(i)and carry) or (carry and arg22(i));
end loop;
return sum;
end;

function cy(arg1,arg2:std_logic_vector(3 downto 0)) return std_logic is

variable sum:std_logic_vector(3 downto 0);
variable carry:std_logic;
begin
carry:='0';
for i in 0 to 3 loop
sum(i):=arg1(i) xor arg2(i) xor carry;
carry:=(arg1(i)and arg2(i)) or (arg1(i)and carry) or (carry and arg2(i));
end loop;
return carry;
end;


function cn(arg1,arg2:std_logic_vector(3 downto 0)) return std_logic is
variable sum,arg22:std_logic_vector(3 downto 0);
variable carry:std_logic;
begin
carry:='1';
arg22:=not(arg2);
for i in 0 to 3 loop
sum(i):=arg1(i) xor arg22(i) xor carry;
carry:=(arg1(i)and arg22(i)) or (arg1(i)and carry) or (carry and arg22(i));
end loop;
carry:=not(carry);
return carry;
end;



end ART00;


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.ART00.all;


entity alu00 is
port( A,B,S: in std_logic_vector(3 downto 0);
F: out std_logic_vector(3 downto 0);
cin: in std_logic;
m: in std_logic;
cout: out std_logic;
comp: out std_logic;
P: out std_logic;
```

```
G: out std_logic
);
end alu00;

architecture behavioral of alu00 is
begin

process(A,B,S,m,cin)
variable D: std_logic_vector(3 downto 0):="0001";
begin
if (A=B) then
comp<='1';
else
comp<='0';
end if;
P<=A(3) and B(3);
G<=A(3) or B(3);
if(m='0') then
cout<='0';
case S is

when "0000" => if (cin='0') then
F<=A;
else
F<=A+D;
cout<=cy(A,D);
end if;
when "0001" => if (cin='0')then
F<=A or B;
else
F<=(A or B)+D;
cout<=cy(A or B,D);
end if;
when "0010" => if (cin='0')then
F<=A or (not(B));
else
F<=(A or (not(B)))+D;
cout<=cy(A or (not(B)),D);
end if;
when "0011" => if (cin='0')then
F<= (A or B)+(A and B);
else
F<="0000";
end if;
when "0100" => if (cin='0')then
F<=A + (A and (not(B)));
cout<=cy(A,A and not(B));
else
F<=(A + (A and (not(B))))+"0001";
cout<=cy(A,A and (not(B))) or cy(A+A and (not (B)),D);
end if;
when "0101" => if (cin='0')then
F<=(A or B) + (A and (not(B)));
cout<=cy(A or B, A and not(B));
else
F<=(A or B) + (A and (not(B)))+"0001";
cout<=cy(A or B, A and (not(B))) or cy( (A or B)+(A and (not(B))), D);
end if;
when "0110" => if (cin='0')then
F<=(A-B)-"0001";
cout<=cn(A,B) or cn(A-B,D);
else
F<=A-B;
cout<=cn(A,B);
end if;
```

```
when "0111" =>if (cin='0')then
F<=A and (not(B))-D;
cout<=cn(A and (not(B)),D);
else
F<=A and not(B);
end if;
when "1000" =>if (cin='0')then
F<=A+(A and B);
cout<=cy(A,A and B);
else
F<=(A+(A and B))+"0001";
cout<=cy(A,A and B) or cy(A+(A and B), D);
end if;
when "1001" =>if (cin='0')then
F<=A + B;
cout<=cy(A,B);
else
F<=(A +B)+"0001";
cout<=cy(A+B,D) or cy(A,B);
end if;
when "1010" =>if (cin='0')then
F<=(A or (not(B)))+ (A and B);
cout<=cy(A or (not(B)),A and B);
else
F<=(A or (not(B)))+ (A and B)+"0001";
cout<=cy(A or (not (B)), A and B) or cy((A or (not (B)))+(A and B),D);
end if;
when "1011" =>if (cin='0')then
F<=A and B-D;
cout<=cn(A and B,D);
else
F<=A and B;
end if;
when "1100" =>if (cin='0')then
F<=A + A;
cout<=cy(A,A);
else
F<=(A+A)+"0001";
cout<=cy(A,A) or cy(A+A,D);
end if;
when "1101" =>if (cin='0')then
F<=(A or B)+A;
cout<=cy(A or B, A);
else
F<=((A or B) + A)+"0001";
cout<=cy(A or B,A) or cy( A or B+A,D);
end if;
when "1110" =>if (cin='0')then
F<=(A or (not(B)))+A;
cout<=cy(A or not(B),A);
else
F<=(A or (not(B))+A)+"0001";
cout<=cy(A or not(B),A)or cy(A or not(B)+A,D);
end if;
when others =>if (cin='0')then
F<=A-D;
cout<=cn(A,D);
else
F<=A;
end if;


end case;
else
cout<='0';
```

```
case S is
when "0000" => F<=not(A);
when "0001" => F<=not( A or B);
when "0010" => F<=(not(A)) and  B;
when "0011" => F<="0000";
when "0100" => F<=not(A and B);
when "0101" => F<=not(B);
when "0110" => F<=A xor B;
when "0111" => F<=A and (not(B));
when "1000" => F<=(not(A))or B;
when "1001" => F<=not(A xor B);
when "1010" => F<=B;
when "1011" => F<=A and B;
when "1100" => F<="1111";
when "1101" => F<=A or (not(B));
when "1110" => F<=A or B;
when others => F<=A;


end case;
end if;
end process;


end behavioral;
```

B.3 Functional Testbench for the ALU

```
-------------------------------------------------------------
-------VHDL testbench for alu (chip 74LS181)
------- Written by Ruomei Wang, March 10, 1996


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


library IEEE;
use IEEE.std_logic_1164.all;

package OP is
function "+"(arg1,arg2:std_logic_vector(3 downto 0)) return std_logic_vector;
end OP;

package body OP is
function "+" (arg1,arg2:std_logic_vector(3 downto 0)) return std_logic_vector is
variable sum:std_logic_vector(3 downto 0);
variable carry:std_logic;
begin
carry:='0';
for i in 0 to 3 loop
sum(i):=arg1(i) xor arg2(i) xor carry;
carry:=(arg1(i)and arg2(i)) or (arg1(i)and carry) or (carry and arg2(i));
end loop;
return sum;
end;
end OP;

library IEEE;
use IEEE.std_logic_1164.all;
```

```
use work.OP.all;

entity E11 is
end;

architecture AE11 of E11 is

signal A,S: std_logic_vector(3 downto 0);
signal B: std_logic_vector(3 downto 0);
signal F: std_logic_vector(3 downto 0);
signal m,cin,cout,comp,P,G:std_logic;

component alu11
port( A,B,S: in std_logic_vector(3 downto 0);
F: out std_logic_vector(3 downto 0);
cin: in std_logic;
m: in std_logic;
cout: out std_logic;
comp: out std_logic;
P: out std_logic;
G: out std_logic
);

end component;

begin

UUT: alu11
port map( A,B,S,F,cin,m,cout,comp,P,G);

M_C_stimulus: process
begin
m<='0';
cin<='0';
wait for 3200 ns;
cin<='1';
wait for 3200 ns;
m<='1';
wait for 3200 ns;
end process;


S_stimulus: process
variable i: integer;
variable seed: std_logic_vector(3 downto 0);
begin
S<="0000";
seed:="0000";
wait for 200 ns;
for i in 1 to 15 loop
seed:=seed+"0001";
S<=seed;
wait for 200 ns;
end loop;
end process;

A_stimulus: process
variable i: integer;
variable seed: std_logic_vector(3 downto 0);
begin
A<="0000";
seed:="0000";
wait for 20 ns;
for i in 1 to 7 loop
seed:=seed+"0010";
```

```
A<=seed;
wait for 20 ns;
end loop;
end process;

B_stimulus: process
variable i: integer;
variable seed: std_logic_vector(3 downto 0);
begin
B<="0000";
seed:="0000";
wait for 20 ns;
for i in 1 to 5 loop
seed:=seed+"0011";
B<=seed;
wait for 20 ns;
end loop;
end process;


end AE11;

configuration cfg_E11 of E11 is
for AE11
for UUT: alu11
end for;
end for;
end cfg_E11;
```

B.4 Simulation Results

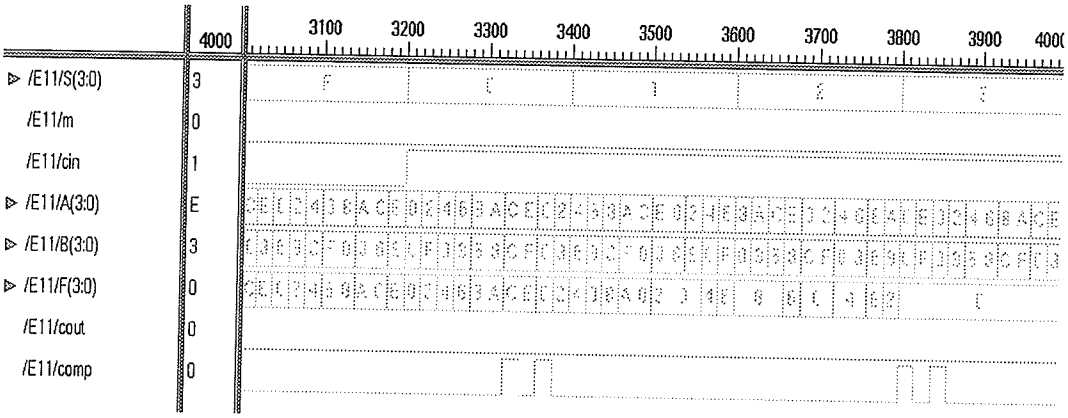


Fig. B.1: Simulation Results

B.5 ATPG Results by Test Compiler (in WGL format)

```
#=============================================================================
#      DESIGN NAME: alu11
```

```
#        CUSTOMER:
#     LIBRARY TYPE: class
#        REVISION: 1.00
#           DATE: 03/13/96
#=============================================================================


waveform alu11

signal
 "A<3>" : input ;
 "A<2>" : input ;
 "A<1>" : input ;
 "A<0>" : input ;
 "B<3>" : input ;
 "B<2>" : input ;
 "B<1>" : input ;
 "B<0>" : input ;
 "S<3>" : input ;
 "S<2>" : input ;
 "S<1>" : input ;
 "S<0>" : input ;
 "F<3>" : output ;
 "F<2>" : output ;
 "F<1>" : output ;
 "F<0>" : output ;
 cin : input ;
 m : input ;
 cout : output ;
 comp : output ;
 P : output ;
 G : output ;
end

timeplate alu11_tp period 100NS
 "A<3>" := input[0PS:P, 5NS:S];
 "A<2>" := input[0PS:P, 5NS:S];
 "A<1>" := input[0PS:P, 5NS:S];
 "A<0>" := input[0PS:P, 5NS:S];
 "B<3>" := input[0PS:P, 5NS:S];
 "B<2>" := input[0PS:P, 5NS:S];
 "B<1>" := input[0PS:P, 5NS:S];
 "B<0>" := input[0PS:P, 5NS:S];
 "S<3>" := input[0PS:P, 5NS:S];
 "S<2>" := input[0PS:P, 5NS:S];
 "S<1>" := input[0PS:P, 5NS:S];
 "S<0>" := input[0PS:P, 5NS:S];
 "F<3>" := output[0PS:X, 95NS:Q'edge, 100NS:X];
 "F<2>" := output[0PS:X, 95NS:Q'edge, 100NS:X];
 "F<1>" := output[0PS:X, 95NS:Q'edge, 100NS:X];
 "F<0>" := output[0PS:X, 95NS:Q'edge, 100NS:X];
 cin := input[0PS:P, 5NS:S];
 m := input[0PS:P, 5NS:S];
 cout := output[0PS:X, 95NS:Q'edge, 100NS:X];
 comp := output[0PS:X, 95NS:Q'edge, 100NS:X];
 P := output[0PS:X, 95NS:Q'edge, 100NS:X];
 G := output[0PS:X, 95NS:Q'edge, 100NS:X];
end

pattern group_ALL ("A<3>", "A<2>", "A<1>", "A<0>", "B<3>", "B<2>", "B<1>",
   "B<0>", "S<3>", "S<2>", "S<1>", "S<0>", "F<3>", "F<2>", "F<1>",
   "F<0>", cin, m, cout, comp, P, G)
# Synopsys Test Compiler, v3.3b (Aug 27, 1995) was used to generate this pattern set
# INPUT VECTOR FILE = alu11.vdb was the source file for this pattern set
 vector(alu11_tp) := [ X X X X X X X X X X X X X X X X X X X X X X X X X ];
```

```
  vector(alu11_tp) := [ X X X X X X X X X X X X X X X X X X X X X X X X ];
 # Pattern 0
  vector(alu11_tp) := [ 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 1 0 0 0 0 1 ];
 # Pattern 1
  vector(alu11_tp) := [ 1 1 1 1 0 0 0 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 ];
 # Pattern 2
  vector(alu11_tp) := [ 1 1 0 0 0 1 0 0 0 1 0 0 1 1 0 0 1 0 1 0 0 1 ];
 # Pattern 3
  vector(alu11_tp) := [ 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1 0 0 1 0 0 ];
 # Pattern 4
  vector(alu11_tp) := [ 0 1 0 0 0 1 0 0 0 1 1 1 1 1 0 0 1 0 0 1 0 0 ];
 # Pattern 5
  vector(alu11_tp) := [ 1 0 0 1 1 1 1 0 0 1 0 1 1 0 0 0 1 0 1 0 1 1 ];
 # Pattern 6
  vector(alu11_tp) := [ 1 1 1 1 1 0 1 1 0 1 1 0 0 0 1 1 0 0 0 0 1 1 ];
 # Pattern 7
  vector(alu11_tp) := [ 1 1 1 1 1 1 0 0 1 0 0 1 1 1 0 0 1 0 1 0 1 1 ];
 # Pattern 8
  vector(alu11_tp) := [ 1 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 1 0 0 1 ];
 # Pattern 9
  vector(alu11_tp) := [ 1 1 1 1 1 0 1 0 0 1 0 1 1 0 1 0 1 0 1 0 1 1 ];
 # Pattern 10
  vector(alu11_tp) := [ 0 1 1 1 0 1 1 0 0 1 0 1 1 1 1 0 1 0 0 0 0 0 ];
 # Pattern 11
  vector(alu11_tp) := [ 0 1 0 0 1 0 1 1 1 0 0 0 0 1 0 0 1 0 1 0 0 1 ];
 # Pattern 12
  vector(alu11_tp) := [ 0 0 0 1 1 1 1 0 1 0 0 0 0 0 0 1 1 0 1 0 0 1 ];
 # Pattern 13
  vector(alu11_tp) := [ 0 1 1 1 1 1 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0 1 ];
 # Pattern 14
  vector(alu11_tp) := [ 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 1 1 0 0 1 0 0 ];
 # Pattern 15
  vector(alu11_tp) := [ 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 ];
 # Pattern 16
  vector(alu11_tp) := [ 0 1 1 0 1 0 0 1 1 0 0 0 0 1 1 0 1 0 1 0 0 1 ];
 # Pattern 17
  vector(alu11_tp) := [ 1 1 1 1 0 1 0 0 1 1 0 1 0 1 0 0 1 0 1 0 0 1 ];
 # Pattern 18
  vector(alu11_tp) := [ 1 1 1 1 1 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 1 1 ];
 # Pattern 19
  v  vector(alu11_tp) := [ 1 1 0 1 0 1 0 0 1 0 1 0 0 1 1 1 1 0 1 0 0 1 ];


 ( omit)


 # Pattern 187
  vector(alu11_tp) := [ 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 ];
 # Pattern 188
  vector(alu11_tp) := [ 1 1 1 1 1 0 0 1 0 0 1 0 1 0 0 1 0 1 0 0 1 1 ];
 # Pattern 189
  vector(alu11_tp) := [ 1 1 1 1 1 0 0 0 1 1 1 0 0 1 1 0 0 0 1 0 1 1 ];
 # Pattern 190
  vector(alu11_tp) := [ 1 1 1 1 1 0 0 0 1 1 1 0 X X X X 0 0 X X X X ];
 end

 end
```

BIBLIOGRAPHY

[1] M. Masud and M. Karunaratne, "Test Generation based on Synthesizable VHDL Descriptions", *Proc. of European DAC*, pp.446-451, 1993.

[2] V. Pla, J. Santucci, N. Giambiasi, "On the Modeling and Testing of VHDL Behavioral Descriptions of Sequential Circuits", *Proc. of Euro-VHDL*, pp. 440-445, 1993.

[3] Y. H. Levendel, P. R. Menon, "Test Generation Algorithms for Computer Hardware Description Languages", *IEEE Trans. on Computers*, vol. c-31, no. 7, July 1982.

[4] R. Khorram, "Functional Test Pattern Generation For Integrated Circuits", *IEEE ITC*, 1984.

[5] Chang Hyun Cho, James R. Armstrong, "B-algorithm: A Behavioral Test Generation Algorithm", *Proc. of IEEE ITC*, 1994.

[6] A. Ghosh, S. Devadas, "Test Generation and Verification for Highly Sequential Circuits", *IEEE Trans. on CAD*, Vol.10, No.5, pp.652-666, May 1991.

[7] J. R. Armstrong, "Hierarchical Test Generation: Where We Are, And Where We Should Be Going", *Proc. of European DAC*, pp.434-439, 1993.

[8] T. Murata, "Petri Nets: Properties, Analysis and Applications", *IEEE*, 1989

[9] S. Olcoz, J. M. Colom, "Toward a Formal Semantics of IEEE Std. VHDL 1076", *Proc. of European DAC*, pp.526-531, 1993.

[10] J. Muller, H. Kramer, "Analysis of Multi-Processor VHDL Specifications with a Petri Net Model", *Proc. of Euro-VHDL*, pp.474-479, 1993.

[11] D.Boussebha, N. Giambiasi, J. Magnier, "Temporal Verification of Behavioral Descriptions in VHDL", *Proc. of European DAC*, pp.692-697, 1992.

[12] J. R. Armstrong, *Chip-Level Modeling with VHDL,* Prentice Hall, Eaglewood Cliffs, New Jersey 07632, 1989.

[13] J. B. Gosling, *Simulation in the Design of Digital Electronic Systems*, Cambridge University Press, 1993.

[14] A. Bechir and B. Kaminska, "CLCLOGEN: Automatic, Functional-Level Test Generator Based on the Cyclomatic complexity Measure and on the ROBDD Representation", *IEEE Trans. on Circuits and Systems,* Vol. 42, No. 7, July 1995.

[15] M. S. Abadir, H. K. Reghbati, "Functional Test Generation for digital Circuits Described Using Binary Decision Diagrams", *IEEE Trans. on Computers*, vol.C-35, no. 4, pp. 375-379, 1986.

[16] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Computers*, vol.C-35, no. 8, pp. 677-691, 1986.

[17] M. Fujita, H. Fujisawa, N. Kawato, "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams", *IEEE ICCAD*, 1988.

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

[19] R. C. Oaken, "An Overview of Test Synthesis Tools", *IEEE Design and Test of Computers,* 1995.

[20] N. H. E. West, C. Thrashing, *Principles of CMOS VLSI Design -A Systems Perspective, Second Edition,* Addison-Wesley Publishing Company, 1993

[21] A. Courbis, J. Santucci, N. Giambiasi, "Automatic Behavioral Test Pattern Generation for Digital Circuits", *Proc. of Euro-VHDL,* pp. 112-117, 1992.

[22] S. B. Akers, "Binary Decision Diagrams", *IEEE Trans. on Computers*, vol.C-27, no. 6, pp. 509-516, June 1978.

[23] F. E. Norrod, "An Automatic Test Generation Algorithm for Hardware Description Lan-

guages", *Proc. of European DAC*, pp. 429-434, June, 1989.

[24] National Semiconductor Corporation, *Logic Databook, Volume II*, 1984.

[25] Applied Microelectronics Inc., *CMC IC Test Head operating Manual, Version 1.0, Document #: 1025*, 1995.

[26] Hewlett Packard, *HP 75000 Model D20 Digital Functional Test System*, 1992.

[27] R.G.Bennetts, "Progress in Design for Test: A Personal View", *IEEE Design and Test of Computers*, pp. 53-58, Spring, 1994.

[28] Synopsys, *Test Compiler Reference*, V3.2b, 1995.

[29] Synopsys, *Test Compiler Tutorial*, V3.2b, 1995.

[30] S. Mazor and P. Langstraat, *A Guide to VHDL*, Kluwer Academic Publishers, 1993.