

Implementation of IEEE Std 1149.1-1990 in VHDL

Peter M. Campbell, Mankuan Vai, Zainalabedin Navabi

Electrical and Computer Engineering Department
Northeastern University
409 Dana Research Center
Boston, Massachusetts 02115
617-437-5413

campbell@nuvlsi.coe.northeastern.edu / vai@northeastern.edu / navabi@northeastern.edu

Abstract

This paper describes the implementation of IEEE Std 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture, using behavioral VHDL (VHSIC Hardware Description Language). The IEEE 1149.1 standard provides a structured method for implementing testability in circuit designs and may be used to provide many different levels of testability. By implementing IEEE Std 1149.1-1990 in VHDL, designs which use the standard may be constructed and simulated to determine the operation of the design and the effectiveness of the included testability. This paper describes the basic components of IEEE 1149.1 as well as the test bench used to stimulate the finished logic. The test bench includes low-level and high-level functions which ease the test application process, provide high-level control, and are portable between different implementations of the test logic. An example which employs the test logic and uses the test bench functions for test application is included.

1. Introduction

In the past few years, VLSI technology has improved so dramatically that the number of transistors possible on a single chip has reached the millions. With this many devices present, it becomes nearly impossible to test the chip externally for proper operation. As a result, incorporating testability into the design of the chip has increased tremendously in importance. The Institute of Electrical and Electronics Engineers has developed a test standard, *IEEE Std 1149.1-1990*, to assist in the test and maintenance of assembled printed circuit boards [1]. This standard defines the operation of test points within the circuit as well as a standard communications interface to access and control the test points. By strategically placing test points within the circuit, the testing of interconnections between distinct modules (i.e. circuit partitions, chips, circuit boards) as well as the testing of individual circuits may be accomplished.

To the best of our knowledge, it appears that little work has been performed on implementing IEEE 1149.1 in VHDL [2]. Other related research has yielded a language which can be used to describe boundary-scan devices using a subset of VHDL [3]. The paper proposes that if a device is not describable by the language, then the device does not conform to the IEEE 1149.1 standard. Despite the obvious advantages of such a language, it lacks any simulation capabilities which is one of the objectives of this project. Another benefit of this project is that high-level test procedures are provided, allowing the operator to specify the function to be performed rather than the specific bit patterns to be used. Furthermore, these high-level procedures may be used directly with different implementations of the test logic in VHDL, providing the finished test code with a high degree of portability.

2. Overview of IEEE 1149.1

As stated in the IEEE Std 1149.1-1990 manual, the goals of the standard were to provide a standardized approach to:

- testing the interconnections between integrated circuits once they have been assembled onto a printed circuit board or other substrate
- testing the integrated circuit itself
- observing circuit activity during the normal operation of the component(s)

The primary testing technique used in IEEE 1149.1 is called *Scan-Design* [4]. Scan-design can make testing easier by allowing internal circuit nodes to be observed and controlled without the use of a large amount of I/O pins. This technique uses registers, called *Scan-Registers*, which have both shift and parallel-load capability. Circuit nodes which are not directly accessible can be controlled and observed by placing a scan-cell (Figure 1) on the node. One type of scan-design is *Boundary-Scan* in which a scan-cell is placed at every module input and output, forming a *Boundary-Scan Register* (BSR).

Using this technique, the signals at the input and output pins may be verified by shifting the values through the Boundary-Scan Register. The component itself may be tested by shifting test data into the cells at the circuit inputs. Interconnections may be verified by shifting data into the cells placed at the input cells of the other connected components. The IEEE 1149.1 standard also allows scan cells to be placed at points in the design other than the component pins. This simplifies the test problem, allowing the component circuitry to be divided into sub-components which may be easier to test.

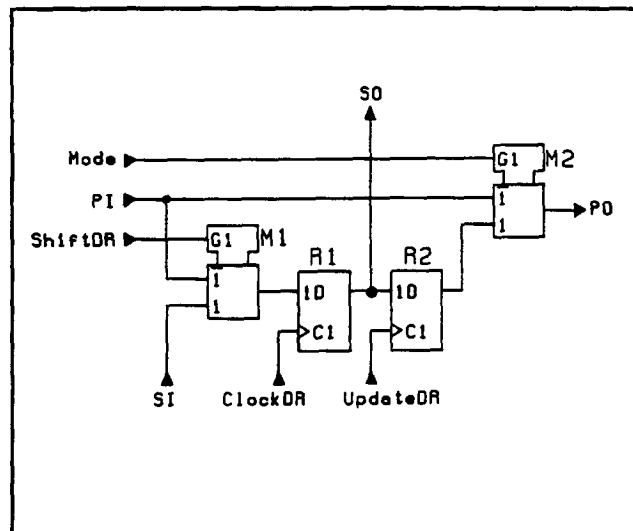


Figure 1 - Scan-Cell

2.1. Test Logic

There are four elements which are required in the test logic. The elements are listed and described below:

- Test Access Port (TAP)
- TAP controller
- Instruction Register (IR)
- Data Registers (DR)

2.1.1 Test Access Port (TAP)

The *Test Access Port* (TAP) provides a standard interface for communication between the test logic and external test equipment or busses. The TAP consists of four signals, listed below.

- TCK - Test Clock. Clock for the test logic.
- TMS - Test Mode Select. Signals at this input are decoded by the controller to control test operations.
- TDI - Test Data Input. Serial input for test logic instructions and data.
- TDO - Test Data Output. Serial output for test instructions and data from test logic.

2.1.2 TAP Controller

The TAP controller generates the internal clock and enable signals required by the test circuitry. It is a synchronous finite state machine consisting of 16 states, only one of which may be active at a time. State transitions are based upon the value of TMS and occur only on a rising edge of TCK. The state diagram is shown in Figure 2. The controller provides the three basic actions required for testing: stimulus application, execution and response capture. The states corresponding to these actions and other important controller states are described below.

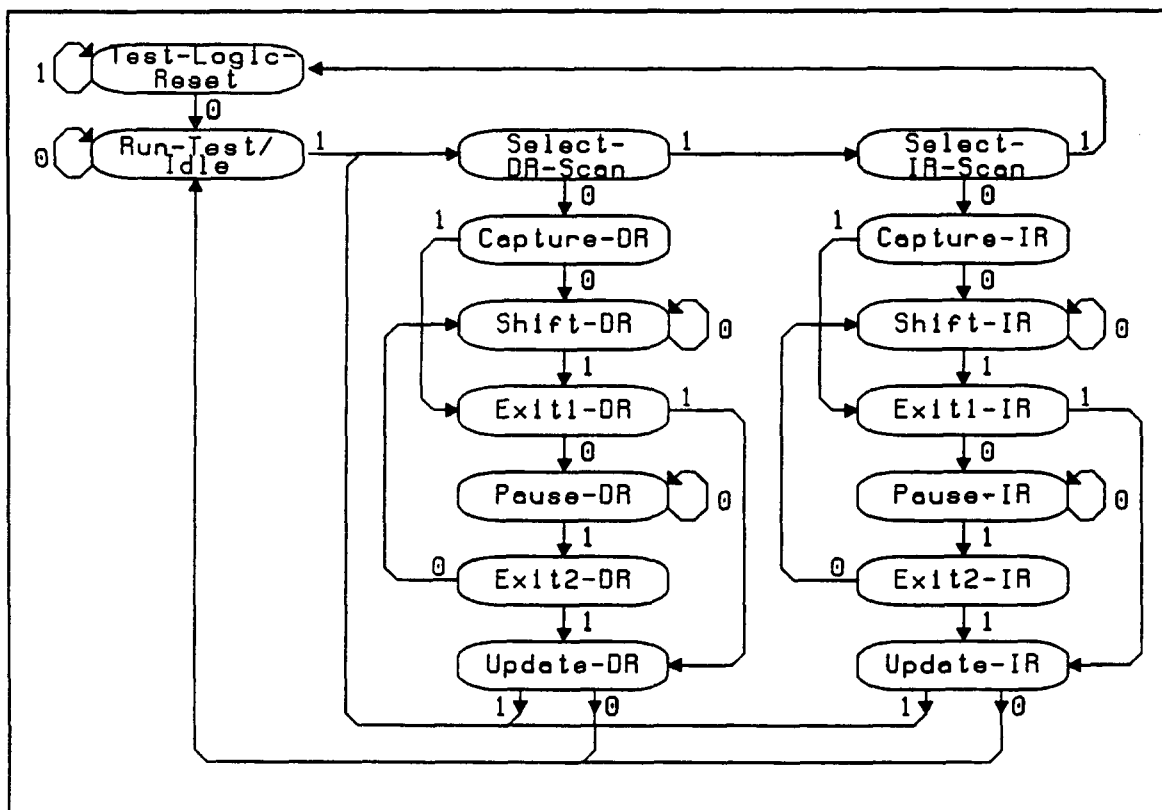


Figure 2 - TAP Controller State Diagram

Test-Logic-Reset

The test logic is disabled in this state so that the component logic can operate normally. The controller will return to this state when TMS is high for five consecutive rising edges on TCK, regardless of the original state. When this state is entered, the Instruction Register output lines are initialized to contain the *BYPASS* instruction (see Section 2.2).

Run-Test/Idle (test execution)

When certain user-defined instructions are present in the Instruction Register, this state is active and these instructions are executed. Instructions which cause no functions to execute do not change the test data registers.

Capture-DR / Capture-IR (response capture)

Data is loaded in parallel into the Data Register(s) / Instruction Register selected by the current instruction.

Shift-DR / Shift-IR

Data is shifted through the register connected between TDI and TDO. The data is shifted one stage for every rising edge of TCK.

Update-DR / Update-IR (stimulus application)

Data is latched onto the parallel outputs of the Data Register(s) / Instruction Register. This is done to prevent the outputs from changing as data is shifted through the register.

2.1.3 Instruction Register (IR)

This register stores the instruction which selects the test to be performed, the Data Register to be accessed, or both. The Instruction Register (IR) is comprised of a chain of two or more *Instruction Register Cells* (Figure 3) which allow data to be serially loaded through the TDI input. Data is latched into the IR in the *Capture-IR* state while data is latched onto the IR outputs in the *Update-IR* controller state. The IR output lines contain the current instruction. There must be a single IR for every TAP controller in a design.

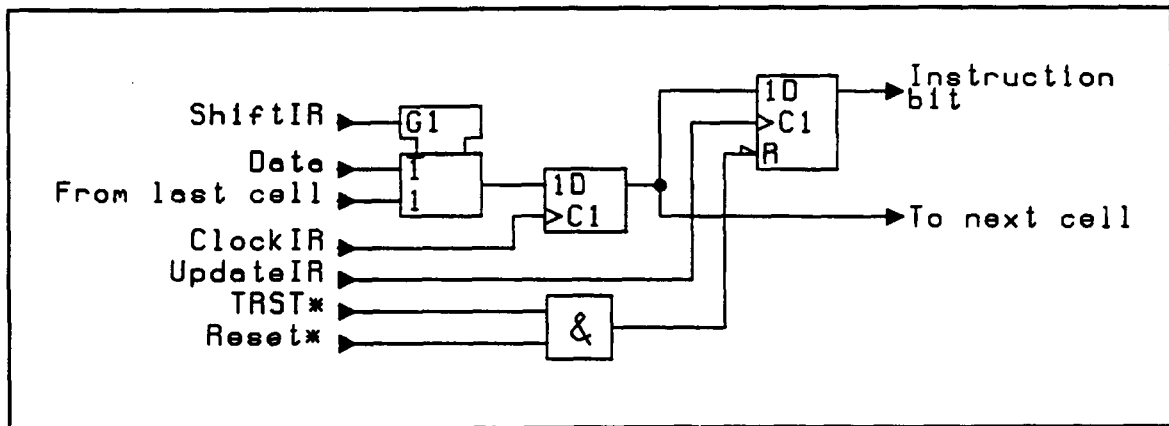


Figure 3 - Instruction Register Cell

2.1.4 Data Registers (DR)

The set of test data registers (DRs) must include a single *Bypass Register* and a single *Boundary-Scan Register*. Optionally, a single *Device-Identification Register* may be included along with *Design-Specific Test Registers*. The Bypass Register (BR) is a single-stage shift-register whose purpose is to allow the test data registers within this device to be skipped. This can shorten the path between the system TDI and TDO when several IEEE 1149.1 compliant devices have their scan-paths connected. Only one Boundary-Scan Register (as described in Section 2) is allowed per component. Included as an option, the Device-Identification Register allows information about the component (such as the part number and manufacturer) to be stored within the component. Other optional registers are Design-Specific Test Data Registers which may be used when additional test features (such as test points and self-tests) are required.

It is required that the Boundary-Scan and Bypass Registers each provide a direct connection between TDI and TDO. Aside from this rule, registers may be connected in any fashion. The DR to be exercised is usually selected based on the value in the IR. Each register (or combination of registers) must have a unique name, i.e. if two registers are connected to make a larger register, the new register must have a name distinct from its sub-registers. The length of each possible DR must be constant.

2.2. Instructions

Instructions are used to select the test function to be performed and/or the registers to be used. There are three mandatory instructions, *BYPASS*, *SAMPLE/PRELOAD*, and *EXTEST*, which are described below. Other instructions may be defined by the component designer.

BYPASS

The *BYPASS* instruction is the only instruction which uses the Bypass Register. The binary code for the *BYPASS* instruction is "11...1" (i.e. logic '1' stored in every IR cell). Other binary codes may also be used for this instruction. This instruction must be loaded onto the IR outputs when the *Run-Test-Idle* controller state is entered.

SAMPLE/PRELOAD

The *SAMPLE/PRELOAD* instruction samples data at the parallel inputs to the selected Boundary-Scan Register and allows data to be shifted into the register. Only the Boundary-Scan Register may be selected. The data at the system pins is loaded into the register in the *Capture-DR* controller state. The data in the shift-register stage is shifted through in the *Shift-DR* state, and is latched onto the parallel output buffers of the register in the *Update-DR* controller state (usually through use of the *EXTEST* instruction). The binary value of the *SAMPLE/PRELOAD* instruction may be selected by the designer.

EXTEST

The *EXTEST* instruction permits testing of off-chip circuitry and interconnections. The data stored in the output-pin Boundary-Scan Register cells is applied and data at the input pins is latched into the register. The data in the shift-registers/latches is typically loaded using the *SAMPLE/PRELOAD* instruction. The *EXTEST* instruction may only select the Boundary-Scan Register. The binary value of the instruction is "00...0" (i.e. a logic '0' is placed on the output of every IR cell). Note that the cells at the input pins may be designed to allow signals to be driven onto the logic inputs when this instruction is selected (in order to prevent misoperation when performing the interconnect test).

3. VHDL Implementation

3.1 Test Logic

The basic test logic is implemented in several parts which are described below.

- The TAP controller
- Instruction, Data and Bypass Register cells
- Instruction and Data Registers created from their respective cells
- The creation of finished test logic using additional logic (multiplexers) and interconnections

Although the basic structure of the test logic is generally the same, the final configuration depends on the designer and on the design itself. The test logic described in the example (Figure 8) uses the minimum entities required by the standard and is in the simplest configuration possible. It contains a TAP controller, an Instruction Register (IR), a Boundary-Scan Register (BSR), a Bypass Register (BR) and the system logic.

3.2. Test Bench

A test bench is a VHDL description which provides stimuli to the input nodes and can observe the output nodes of the circuit under test. A test bench has no external ports, as noted by the absence of a PORT clause in the declaration. The test bench specifies the waveforms applied to the input ports of the circuit under test and allows the output ports to be observed.

A specific design style is used within the test bench presented here. This style has several advantages: it allows tests to be specified sequentially as procedure calls, it allows test data files to remain open (without being reset) for subsequent procedure calls, and it permits signal assignments to occur in parallel with the operation of the test logic. The style is shown in Figure 4.

To simplify the test-application process, several packages may be made visible to the test bench. These packages are called *micro_procedures*, *macro_procedures*, and *operation_macros*. The level of abstraction increases in each package, reducing the amount of code needed to specify a test sequence or operation.

3.3 Micro-Procedures

Low-level procedures, called *Micro-procedures*, are used to assign signals directly to the TAP inputs. Through the use of overloading, the signal values to be applied can be specified as either a bit vector or a datafile. Procedures are also included to modify the characteristics of the test bench. The design style used in the test bench allows sequential procedure calls which use the datafile to continue from the last item read in the file. The micro-procedures are listed in Figure 5 (overloaded procedures are listed only once). Note that the coding of micro-procedures generally depends upon the implementation of the test logic in VHDL.

```

USE WORK.micro_procedures.ALL;
USE WORK.macro_procedures.ALL;
USE WORK.operation_macros.ALL;

ENTITY example IS END example;

ARCHITECTURE io OF example IS
  declarations
  ...
BEGIN
  instantiation of circuit-under-
  test and test logic
  initialization procedures
  concurrent signal assignments
  ...
  a : PROCESS
    file declarations for procedures
  BEGIN
    sequential procedure calls
    ...
    WAIT;
  END PROCESS a;
END io;

```

Figure 4 - Test Bench Design Style

```

TYPE bit_data IS FILE OF CHARACTER;

PROCEDURE write_clock ( time_low, time_high : IN TIME );
PROCEDURE read_clock ( time_low, time_high : OUT TIME );
PROCEDURE write_state ( current_state : IN INTEGER );
PROCEDURE read_state ( current_state : OUT INTEGER );
PROCEDURE write_next_time ( time_value : IN TIME );
PROCEDURE read_next_time ( time_value : OUT TIME );
PROCEDURE goto_ ( SIGNAL theclock, thesignal, thedata : OUT BIT;
                 to_state : IN INTEGER );
PROCEDURE assign_bits ( SIGNAL theclock, thesignal, thedata : OUT BIT;
                       VARIABLE thesignal_file : IN bit_data;
                       data_vector : IN BIT_VECTOR );

```

Figure 5 - Micro_procedures Package Procedures

The clock duty cycles may be specified using the *write_clock* and *read_clock* procedures. The *write_state* and *read_state* procedures respectively store and read the active state of the machine. The *goto* procedure uses *read_state* and *write_state* to generate a bit pattern which will change the state of the machine from the current state to the desired state. The *write_next_time* and *read_next_time* procedures are used to allow sequential calls to *assign_bits* without overwriting

old transactions. The `assign_bits` procedure is used to apply signals to TMS (`thesignal`) and TDI (`thedata`) synchronized with TCK (`theclock`) using clock parameters provided by `read_clock`. The procedure calls `read_next_time` to determine when the next signal assignment can be made without overwriting old transactions. Before the procedure ends, `write_next_time` is called to store the next time a signal assignment can be made, preventing the new signal assignments from being overwritten.

3.4 Macro-procedures

Higher-level procedures, called *Macro-procedures*, use micro-procedures to apply signals to the TAP inputs, allowing the test bench to specify the function(s) to be performed rather than the actual bit patterns to be applied. These procedures are stored in the `macro_procedures` package. As with the micro-procedures, overloading is used to allow signal values to be specified as either a bit vector or a datafile, and sequential procedure calls using a datafile continue from the last item read in the file. The macro-procedures are listed in Figure 6 (overloaded procedures are listed only once). The operations performed by macro-procedures are basically defined by the IEEE 1149.1 specification and are independent of the VHDL implementation of IEEE 1149.1 and of the final design of the test logic and target circuit (i.e. they are general enough to be used in testing any IEEE 1149.1 design). This portability between different logic implementations allows macro-procedures to be used in a range of designs without being rewritten. Note that the micro-procedures may require recoding for each type of test logic.

```
PROCEDURE dr_scan ( SIGNAL theclock, thesignal, thedata : OUT BIT;
                   VARIABLE thedata_file : IN bit_data;
                   number_of_cycles : IN INTEGER );
PROCEDURE ir_scan ( SIGNAL theclock, thesignal, thedata : OUT BIT;
                   VARIABLE thedata_file : IN bit_data;
                   number_of_cycles : IN INTEGER );
```

Figure 6 - Macro_procedures Package

The `dr_scan` and `ir_scan` procedures load a bit pattern into the TAP Data Register and Instruction Register, respectively, and leave the controller in the *EXITI-DR/IR* state. The procedures use the `goto`, `assign_bits` and `write_state` procedures to perform their function.

3.5 Operation Macros

An even higher level of procedures may be defined as *Operation Macros*. These operation macros are similar to macro-procedures but are written for a specific design incorporating IEEE 1149.1 and generally cannot be directly shared between different designs. Typically, operation macros would be written after the required test operations are known and have been developed. Operation macros are contained in the `operation_macros` package. An operation macro developed for the example is shown in Figure 7. The `extest_op` macro loads the *Sample-Preload* instruction into the IR, loads a bit pattern into the BSR, loads the *Extend* instruction into the IR and then goes to the *Update-IR* state to apply the data stored in the BSR.

```
PROCEDURE extest_op ( SIGNAL theclock, thesignal, thedata : IN BIT;
                    length : IN INTEGER );
```

Figure 7 - Operation_macros Package

4. Example Implementation

This example is based on a nibble-comparator. The comparator is designed to allow several comparators to be connected in a bit-slice fashion to create a larger comparator. It has two 4-bit inputs (a and b), three mode inputs (a_gt_b, a_eq_b, and a_lt_b) and three mode outputs (gt_r, eq_l, and ls_s). The comparator is made IEEE 1149.1 compliant by adding boundary-scan cells to the inputs and outputs, thereby forming a BSR. The scannable comparator is wired with a TAP Controller, an IR, a Bypass Register (BR) and assorted glue logic to create the finished design (Figure 8). The test bench applies signals to the comparator inputs and the TAP inputs simultaneously. The test is designed to allow the comparator to operate normally for a period of time. The test logic is kept inactive until midway through the comparator input sequence (1000 NS), at which point data is loaded into the boundary-scan

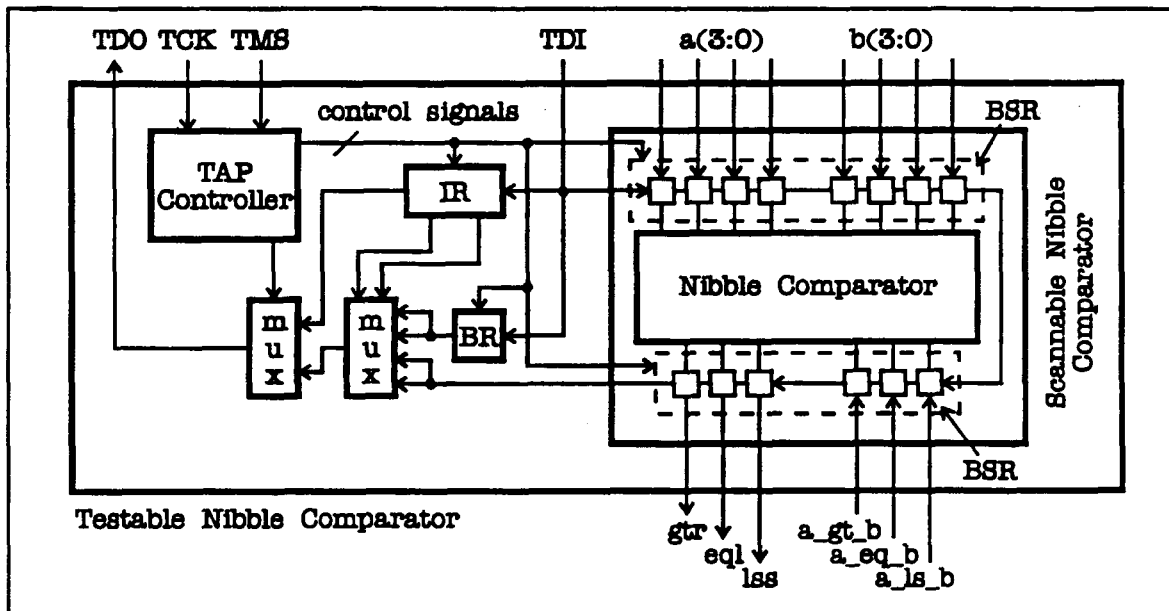


Figure 8 - Testable Nibble Comparator Interconnections

register and applied, effectively blocking the comparator inputs and performing an interconnection test. The TAP controller then returns to the Test-Logic-Reset state, making the test logic inactive. The test bench Architecture is shown in Figure 9.

5. Conclusions

As circuits grow in complexity, the importance of testability in the design process will increase dramatically as will the use of Hardware Description Languages. The IEEE Std 1149.1-1990 test standard provides a structured method of incorporating testability into a design and makes it easier for modules from different suppliers to become part of a testable system. In implementing IEEE 1149.1 in VHDL, a model of the standard is created, allowing an entire system which uses the standard to be simulated and the test patterns developed before the system is constructed. The test procedures developed for the VHDL implementation of IEEE 1149.1 ease the test application process, allow for high-level control and are portable among different implementations of the test logic.

6. References

1. "IEEE Standard Test Access Port and Boundary-Scan Architecture", IEEE Std 1149.1-1990, The Institute of Electrical and Electronics Engineers, Inc., 1990.
2. "IEEE Standard VHDL Language Reference Manual", IEEE Std 1076-1987, The Institute of Electrical and Electronics Engineers, Inc., 1988.
3. Parker, K.P. and S. Oresjo, "A Language for Describing Boundary-Scan Devices", *Proceedings of the 1990 ASIC Seminar and Exposition*, September 1990.
4. M. Abramovici, M.A. Breuer and A.D. Friedman, *Digital Systems Testing and Testable Design*. New York: Computer Science Press.

```
USE WORK.micro_procedures.ALL;
USE WORK.macro_procedures.ALL;

ENTITY testable_nibble_comparator_tester IS
END testable_nibble_comparator_tester;

ARCHITECTURE io OF testable_nibble_comparator_tester IS
  COMPONENT testable_nibble_comparator PORT ( a, b : IN BIT_VECTOR...
  ...
  END COMPONENT;
  CONSTANT test_file : STRING := "test_data";
  CONSTANT extest : BIT_VECTOR (1 DOWNTO 0) := "00";
  CONSTANT sample_preload : BIT_VECTOR (1 DOWNTO 0) := "01";
  SIGNAL a, b : BIT_VECTOR (3 DOWNTO 0);
  SIGNAL tck, tms, tdi, tdo, gtr, eq1, lss : BIT;
BEGIN
  t1 : testable_nibble_comparator PORT MAP ( a, b, gnd, vdd, gnd, tms,
      tck, tdi, gtr, eq1, lss, tdo);
  write_clock ( 50 NS, 50 NS );      -- Initialize clock duty cycles
  write_next_time ( 0 FS );          -- Init. next assignment time
  write_state ( test_logic_reset );  -- Store current state of FSM
  a <= "0000",                       -- a = b
      "1111" AFTER 0500 NS,          -- a > b
      ...
      "1111" AFTER 6000 NS;          -- a > b
  b <= "0000",                       -- a = b
      "1110" AFTER 0500 NS,          -- a > b
      ...
      "0000" AFTER 6000 NS;          -- a > b

  p1 : PROCESS
    FILE thetest_file : bit_data IS IN test_file; -- Test vector file
  BEGIN
    -- Stay in Run-Test-Idle state (to disable test logic)
    -- until time = 1000 NS (i.e. for 10 cycles).
    assign_bits ( tck, tms, tdi, "1", "0", 10 );
    -- Load Sample/Preload instruction code into IR.
    ir_scan ( tck, tms, tdi, sample_preload );
    -- Load 14-bit test vector into BSR from "test_data".
    dr_scan ( tck, tms, tdi, thetest_file, 14);
    -- Load Extest instruction code into IR so data is driven
    -- onto BSR outputs in the Update-IR state.
    ir_scan ( tck, tms, tdi, extest );
    -- Go to the Test-Logic-Reset state to disable test logic.
    -- Pass thru Update-IR state so new instruction is present.
    goto ( tck, tms, tdi, 15 );
    -- Keep the clock running until comparator data is depleted.
    assign_bits ( tck, tms, tdi, "0", "0", 5 );
    WAIT;
  END PROCESS p1;
END io;
```

Figure 9 - TAP_example Test Bench