Methodology

When you use Test Compiler at the module level, use the following methodology:

1. Define the appropriate scan style.

   If the design is purely combinational, define `combinational` as your scan style.

   If the design contains sequential logic, select the scan style from one of the scan styles supported by Test Compiler. The scan styles that Test Compiler supports are

   - Multiplexed flip–flop (`multiplexed_flip_flop`)
   - Clocked scan (`clocked_scan`)
   - LSSD (`lssd`)
   - Auxiliary clock LSSD (`aux_clock_lssd`)

Note

Test Compiler requires you to use the same scan style on the entire chip. Select the appropriate scan style for the chip and identify it before optimizing each subdesign of the chip.

2. Optimize the subdesign by using Design Compiler.

   For sequential subdesigns, defining the scan style before performing logic optimization invokes the *Test–Smart Compile* feature. Test–smart compile limits the set of sequential cells used during logic optimization. The restriction prevents the use of complex sequential cells, which do not have scan equivalents, for functional logic. For example, Test–Smart Compile would not use a multiplexed flip–flop cell for functional logic because this cell must be reserved for use as a scannable equivalent for a D flip–flop.

3. Check the test design rules.

   For combinational designs, the `check_test` command checks for combinational feedback loops. For sequential designs, the `check_test` command also analyzes the design for compliance with the scan design rules associated with the selected scan style, and reports any violations. Test Compiler links all violations reported by the `check_test` command to the schematics in the Design Analyzer so you can easily locate testability problems.

4. Estimate fault coverage results.

   Test Compiler does not require you to eliminate test design rule warnings to run ATPG, but many design rule violations cause significantly lower fault coverage results. Statistical ATPG (`create_test_patterns –sample n`) quickly estimates the fault coverage for a subdesign.

Note

Because of differences in input port controllability and output port observability after the subdesign is embedded within the hierarchy of the complete design, you need to consider the fault coverage numbers on an individual block as best–case fault coverage numbers. High fault coverage on each subdesign does not guarantee high fault on the complete chip, but low fault coverage on a subdesign results in lowered fault coverage of the entire chip.

5. Fix any testability problems.

   If the fault coverage estimate reports unacceptable fault coverage results, you need to fix the testability problems identified by `check_test`. Test Compiler identifies the source of the test design rule violation.

   Select the best method to correct the problem and manually modify the design to reflect the change—modify the RTL description, modify the gate–level netlist, or use the Design Compiler design editing commands.

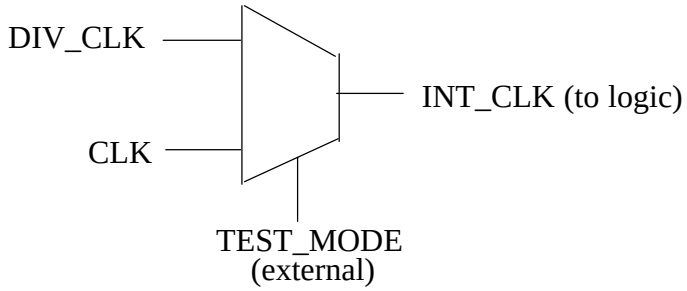
   After modifying the design to fix testability problems, repeat steps 1 through 5 until you reach an acceptable fault coverage result.

Note

If you modify the design with design editing commands, you do not need to repeat steps 1 and 2; no `read` command is executed to input changes.

Figure 2-12 shows that INT_CLK, the clock signal for some of the flip–flops in COMPUTE_BLOCK, is uncontrollable because it is driven by sequential logic. To improve COMPUTE_BLOCK testability, you need to provide a test mode that uses top–level clock port CLK as the clock signal for the affected flip–flops during scan testing, but continues to use internally generated clock signal as the clock signal for the flip–flops during functional operation. The test mode logic is added to the CLOCK_GEN block, which generates the uncontrollable clock signal, INT_CLK. To provide the test mode, you must add a new input port, TEST_MODE, to CLOCK_GEN (and to COMPUTE_BLOCK). TEST_MODE controls the test mode configuration. For this example, the test mode logic consists of a multiplexer with TEST_MODE as the select line and CLK and internally generated clock signal (renamed to DIV_CLK) as the data inputs.

Figure 2–12 CLOCK_GEN Test Mode Logic



A 2:1 multiplexor can be modeled by using an if statement in both Verilog and VHDL. The Verilog process to add a test mode to CLOCK_GEN is shown in Example 2-1. The VHDL process to add a test mode to CLOCK_GEN is shown in Example 2-2.

Example 2–1 Verilog Test Process

```
always @(TEST_MODE or CLK or DIV_CLK)
begin
   if (TEST_MODE)
      INT_CLK <= CLK;
   else
      INT_CLK <= DIV_CLK;
end
```

Methodology

When you use Test Compiler at the chip level, use the following methodology (steps 1 through 4).

1. Perform the analysis steps outlined in “Methodology” in Chapter 2, “Testability at the Module Level.”

   Performing the testability analysis steps on each module of your design minimizes the possibility of testability problems at the top level, but it is possible for testability problems to be introduced as you move up the hierarchy. Before you perform scan insertion, verify that all test violations have been identified and resolved.

2. Insert scan–test structures.

   Scan insertion works hierarchically. You insert scan logic at the top level of the design; Test Compiler automatically works through the entire design hierarchy.

   The exercises in this tutorial use the full–scan test methodology, in which *all* sequential cells are replaced with scannable equivalents. Test Compiler Plus also supports constraint–driven partial scan, which selects a subset of the sequential cells to scan, according to your performance, area, and testability constraints.

3. Perform timing analysis and incremental optimization, if necessary.

   The integration between Design Compiler and Test Compiler makes it easy to optimize your design with the scan–test structures in place, thus minimizing the performance and area effect of the scan technique.

   Use timing analysis to verify that no setup or hold violations are on the scan path. Test Compiler uses zero–delay models during ATPG; therefore, timing violations on the scan path may cause simulation mismatches or failing vectors on the ATE (automatic test equipment).

4. Generate and format manufacturing test patterns.

   Now that the design is complete, you are ready to use ATPG to generate the final set of manufacturing test patterns. The final destination of the manufacturing test patterns that Test Compiler generates is, in most cases, a semiconductor vendor. It is important that you define any vendor–specific requirements with the appropriate environment variables before you run ATPG. After the patterns are generated, they are formatted in the vector format you designate.

Note

The best source for vendor–specific requirements is your semiconductor vendor. Each vendor–specific vector format has accompanying documentation that gives examples of vendor requirements and describes how to define those requirements to Test Compiler.

Alarm Clock Design

You optimized the two subdesigns instantiated at the top level of the alarm clock design in the Chapter 2 exercises. Now, optimize the top level of the design. Remember that you added a port to COMPUTE_BLOCK to configure the design for test mode. Before you optimize the top–level design, update the source code to reflect the additional port.

To modify the source code for TOP:

1. Copy the source file into your working directory and add write permission to the read–only file.

   For Verilog, use the following UNIX commands:

   ```
   % cp verilog/TOP.v .
   ```

   ```
   % chmod u+w TOP.v
   ```

   For VHDL, use the following UNIX commands:

   ```
   % cp vhdl/TOP.vhd .
   ```

   ```
   % chmod u+w TOP.vhd
   ```

Resolving ATPG Conflicts

The typical causes of an ATPG conflict are

- Three–state logic that always causes bus contention or bus float.

  Test Compiler requires that one and only one driver be active on a three–state net at any time. If the bus decode logic always activates multiple drivers (bus contention) or no drivers (bus float), the single active driver condition cannot be met and an ATPG conflict results.

  Test Compiler considers pullup and pulldown resistors on a three–state net to be the active drivers on the bus when the decode logic selects no driver on the bus.

Note

If the bus decode logic generates at least one state with a single driver active on the bus, but also generates states in which Test Compiler sees bus float or bus contention on a three–state bus, reduced fault coverage occurs. However, Test Compiler does not flag an ATPG conflict.

- Asynchronous set or reset signals that cannot be simultaneously disabled.

  To perform scan shift, you must disable the asynchronous resets on all scannable sequential cells. If the circuit does not meet the disabling requirement, an ATPG conflict occurs.

- Conflicting requirements defined by test mode configurations.

  You may stipulate conflicting requirements when using the `set_test_hold` or `set_test_require` commands. If you do stipulate conflicting requirements, ATPG determines that the requested configuration cannot be achieved and generates an ATPG conflict.

➧ To generate the ATPG conflict report:

1. Click *Display Reports...* in the Test Synthesis dialog box.

   The Test Reports dialog box is displayed.
2. Click *ATPG Conflict* to select the ATPG conflict report.
3. Click *Apply* to display the Test Report window with the ATPG conflict report.

➧ To get the schematic representation of the conflict report:

1. Select the underlined line with the left mouse button.

   In the Test Report window, the text is highlighted and the Show button is enabled (displayed black, not grayed–out), as shown in Figure 3-1.

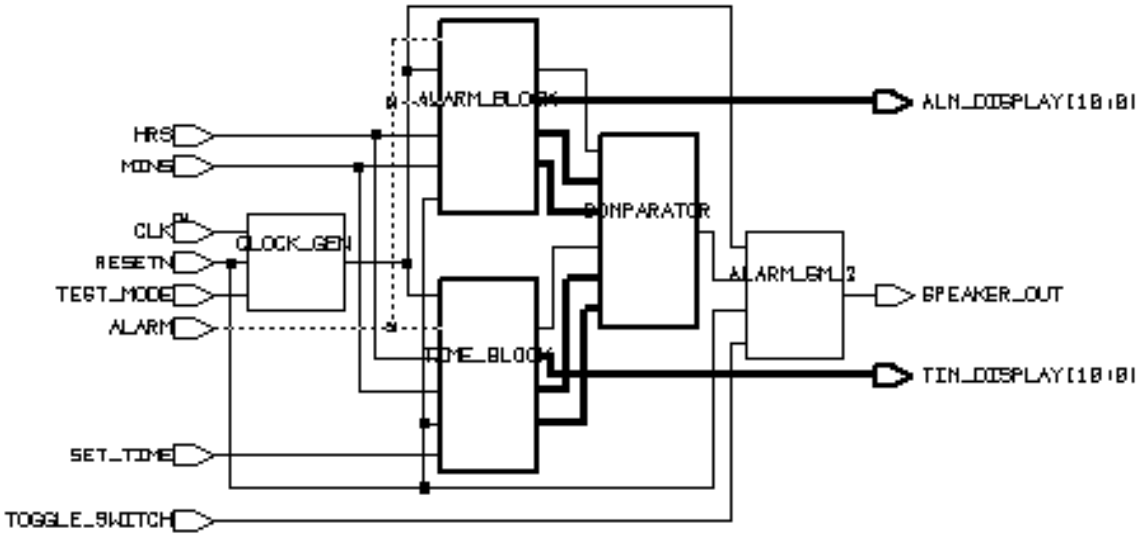
- To close the Test Report window lick *Cancel*.
- To close the Test Reports dialog box click *Cancel*.
- To close the Test Synthesis dialog box click *Cancel*.

The CLK_DISPLAY three–state bus is driven by the ALM_DISPLAY bus and the TIM_DISPLAY bus. Both the ALM_DISPLAY bus and the TIM_DISPLAY bus are outputs from COMPUTE_BLOCK.

- To push into Schematic View for COMPUTE_BLOCK double–click on the COMPUTE_BLOCK instance in the Schematic View of TOP.

The signals on the ENABLE input ports on the ALARM_BLOCK and TIME_BLOCK designs are the three–state enable signals. In the COMPUTE_BLOCK design (Figure 3-3), you can see that the three–state enables for both the ALM_DISPLAY bus and the TIM_DISPLAY bus are controlled by the ALARM input port. You always have bus float (ALARM=0) or bus contention (ALARM=1). The three–state enable signals are actually a design error; the behavior recommended by Test Compiler is to drive the bus with ALM_DISPLAY when ALARM=1 and drive the bus with TIM_DISPLAY when ALARM=0.

Figure 3–3 COMPUTE_BLOCK Design



4 Testability at the Board Level

Previous exercises dealt with test techniques applicable at the module and chip level. Although Test Compiler does not directly support board–level test, in this chapter you add logic to the alarm clock design to provide an interface between chip–level and board–level testing.

Boundary Scan is a test technique defined by the IEEE 1149.1 standard that can be applied at the printed circuit board level. Test Compiler automatically synthesizes 1149.1– compliant (or –compatible) boundary scan logic around your core logic design. Refer to Chapter 8, “Adding Boundary Scan Test Circuitry,” of the *Test Compiler Reference Manual* for a full discussion of Test Compiler’s programmable boundary scan synthesis capability.

Working with the alarm clock core design, you learn how to

Prepare the chip–core for boundary scan insertion.

Insert boundary scan circuitry into a design.

Generate reports on the boundary scan implementation.

Analyze boundary scan testability.

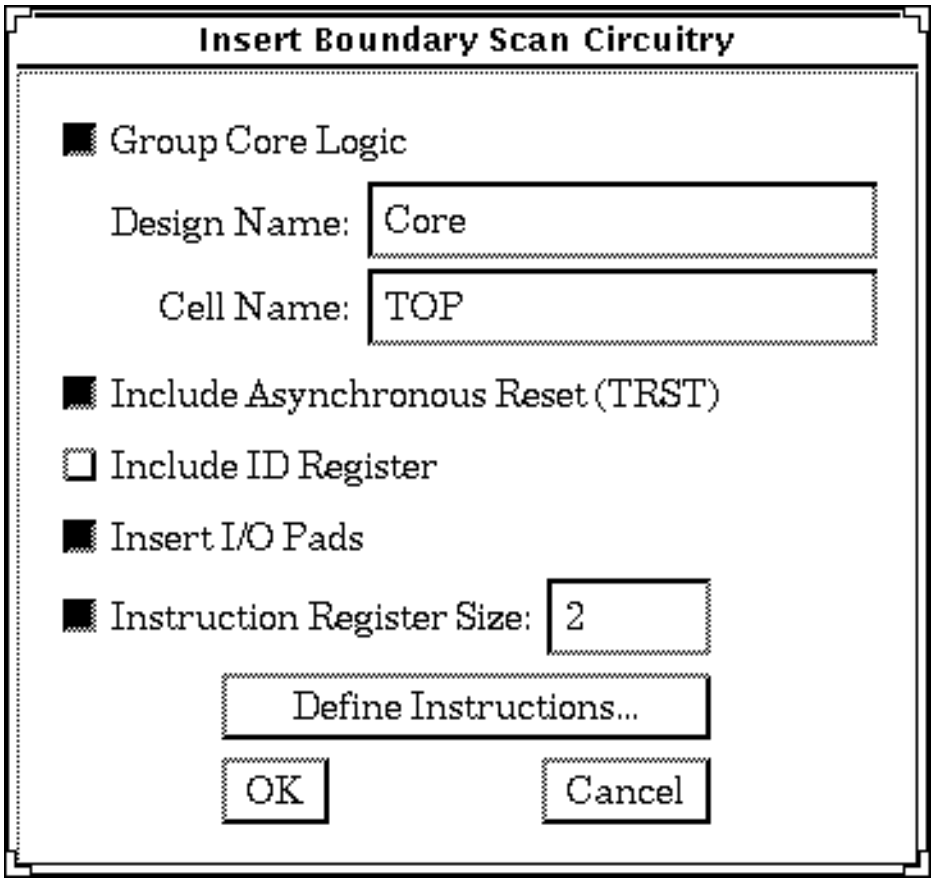
Generate boundary scan test patterns.

Insert Boundary Scan Logic

The default boundary scan implementation synthesized by Test Compiler includes a two–bit IR (Instruction Register) to support the mandatory BYPASS, EXTEST, and SAMPLE/PRELOAD instructions, a bypass register, and the BSR (Boundary Scan Register) with all ports included. The default TAP controller is implemented with an asynchronous reset.

When you work in the Design Analyzer environment, the default behavior is to group the core logic. Grouping the core logic allows you to maintain its identity in the final design for schematic generation and simulation. Figure 4-1 shows the Insert Boundary Scan Circuitry dialog box. Note that the *Group Core Logic* toggle button is selected.

Figure 4–1 Insert Boundary Scan Circuitry Dialog Box



Fault Simulating ATPG Vectors

TestSim supports fault simulation of ATPG vectors either in serial mode (multiplexed flip–flop designs only) or in parallel mode (default or initialization test protocols only).

Note

TestSim does not support fault simulation for LSSD designs.

In general, the fault coverage results reported by TestSim will be greater than or equal to the fault coverage reports reported by Test Compiler. Increased fault coverage results in TestSim are usually due to cells which are assumed to be black boxes by Test Compiler, because of test design rule violations, which are not black boxes for TestSim and can be used to increase fault coverage results or internal 3–state nets. In some special cases, fault coverage results reported by TestSim can be lower than the fault coverage results reported by Test Compiler. Decreased fault coverage results in TestSim are usually due to redundant faults, which cannot be identified by TestSim and are marked as untested, or probable detects, which are counted as untested for fault coverage calculations. For more information, refer to Chapter 10, “Fault Simulation with TestSim,” in the *Test Compiler Reference Manual*.

Parallel Fault Simulation of ATPG Vectors

Parallel fault simulation of the ATPG pattern set is accomplished by fault simulating the `.vdb` file generated by `create_test_patterns`. You will use the `TOP.vdb` file generated in Chapter 3 of this tutorial to perform parallel fault simulation.

To fault simulate:

1. Click on the `TOP` design icon.
2. Select *Tools→Test Synthesis...*

   The Test Synthesis dialog box is displayed.

- To restore the test status:

  1. Click *Restore/Delete Test Program...* in the Test Manger dialog box.
  2. Click on the *Restore Test Program* toggle button.
  3. Enter **`TOP.vdb`** in the *Name:* text box.
  4. Click *Apply*.
  5. Click *Cancel*.

- To generate the coverage report:

  1. Click *Analyze Fault Coverage...* in the Test Manger dialog box.
  2. Click *Coverage* to select the coverage report.
  3. Click *Apply*.

  Enter the Test Compiler coverage information in the Test Compiler column in Table 5-1.

  Notice that TestSim detected an additional 4 faults (3394 versus 3390). These additional faults are faults on the enable line of a 3–state driver. In addition, the number of tied and untested faults differ between TestSim and Test Compiler. This difference is partially due to the additional detected faults in TestSim, but there is still a difference of 46 faults between the ATPG and fault simulation results. The difference is explained by the difference in classification of tied faults between TestSim and Test Compiler. Some of the faults marked as tied by TestSim are marked as redundant by Test Compiler – in this case, due to the `test_hold` attributes on the design, the redundant faults are then marked as untested by Test Compiler. For more information on difference in fault classifications between TestSim and Test Compiler, refer to Chapter 10, “Fault Simulation with TestSim,” in the *Test Compiler Reference Manual*.

- To close the Test Report window click *Cancel*.

- To close the Test Reports dialog box click *Cancel*.

Generating a TestSim Model

You will be doing several fault simulation runs during the course of this chapter, so you should generate a TestSim model for the alarm clock design.

TestSim libraries are required to generate a TestSim model for your design. TestSim automatically searches for the TestSim library files *library*_testsim.db in your search_path when generating TestSim design models and performing fault simulation. The TestSim library file names should *not* be added to the link_library variable

➧ To read the design database:

1. Select *File→Read...*

   The Read File dialog box is displayed.

2. Select ../ (Move up one directory).
3. Click *OK*.
4. Select TOP.db.
5. Click *OK*.

➧ To generate the TestSim model for the alarm clock design:

1. Select *Setup→Command Window...*

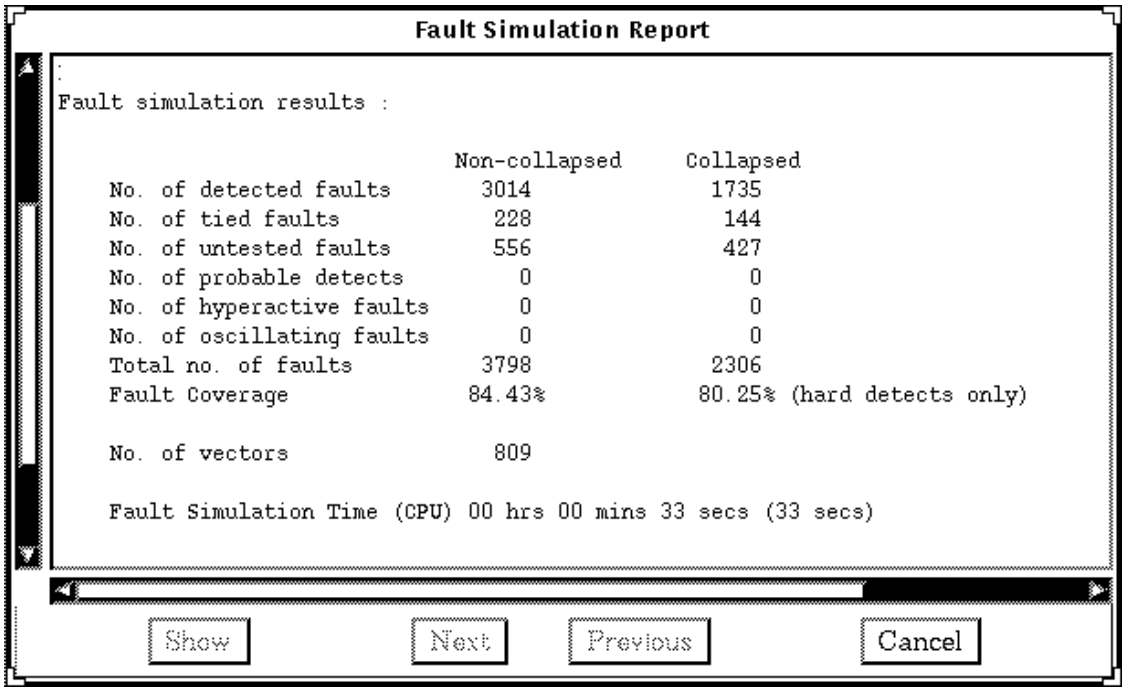
   The Command Window is displayed.

2. Enter **create_testsim_model TOP_testsim.db** in the *design_analyzer>* text box. Press Return.
3. Click on the icon in the upper–lefthand corner.

   The Command Window is iconified.

Now that you have the TestSim library and TestSim model, you are ready to start fault simulating. First you will fault simulate the patterns generated using Test Compiler ATPG, then you will generate and fault simulate some design verification (functional) vectors. Finally, you will use TestSim and Test Compiler together to create a multi–style, multi–pass test process.

Figure 5–5 Fault Simulation Results – VHDL



The 809 functional vectors achieved 84.43% fault coverage. If fault coverage above 84% is required, you can use the multi–style, multi–pass capability of Test Manager to run incremental ATPG on this design. Refer to Chapter 9, “Test Manager,” of the *Test Compiler Reference Manual* for more information on multi–style, multi–pass testing.

Wrap–Up

The exercises in this chapter introduced the basics of using TestSim to fault simulate ATPG patterns and functional vectors. TestSim and Test Compiler can be used together in a multi–style, multi–pass test process to achieve the highest possible fault coverage results for your design.