BSD Compiler
User Guide

Version F-2011.09-SP2, December 2011

SYNOPSYS®

Contents

Preface

This preface includes the following sections:

- What’s New in This Release
- About This Guide
- Customer Support

What's New in This Release

Information about new features, enhancements, and changes, along with known problems and limitations and resolved Synopsys Technical Action Requests (STARs), is available in the *BSD Compiler Release Notes* in SolvNet.

To see the *BSD Compiler Release Notes*,

1. Go to the Download Center on SolvNet located at the following address:

   https://solvnet.synopsys.com/DownloadCenter

   If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.

2. Select BSD Compiler, and then select a release in the list that appears.

About This Guide

The *BSD Compiler User Guide* describes usage and methodology scripts for the BSD Compiler product.

BSD Compiler is the boundary-scan automation component of the 1-Pass test suite. BSD Compiler generates your boundary-scan logic for IEEE Std 1149.1 and IEEE Std 1149.6. BSD Compiler verifies that the logic conforms to IEEE Std 1149.1 and generates a Boundary-Scan Description Language (BSDL) file and test vectors for the design. BSD Compiler generates a BSDL file and test vectors for IEEE Std 1149.6.

Audience

The primary readers of the *BSD Compiler User Guide* are ASIC design engineers who are implementing boundary-scan logic and who are already familiar with the Design Compiler product.

A secondary audience is manufacturing engineers who develop tests for boards that include these boundary-scan devices.

Related Publications

For additional information about BSD Compiler, see the documentation on SolvNet at the following address:

https://solvnet.synopsys.com/DocsOnWeb

You can find related information in the documentation for the following Synopsys products:

- Design Compiler
- DFT Compiler
- VCS

These documents supply additional information:

- *BSD Compiler Reference Manual*
- *Supplement to IEEE Std 1149.1b-1994, IEEE Standard Test Access Port and Boundary-Scan Architecture*
- *IEEE Std 1149.1a-1993 Standard Test Access Port and Boundary-Scan Architecture*
- *IEEE Std 1149.1-2001 Standard Test Access Port and Boundary-Scan Architecture*
- *IEEE Std 1149.6-2003 Standard for Boundary-Scan Testing of Advanced Digital Networks*

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates syntax, such as `write_file`.
`Courier italic`	Indicates a user-defined value in syntax, such as `write_file` *`design_list`*.
`Courier bold`	Indicates user input—text you type verbatim—in examples, such as `prompt>` **`write_file top`**
[ ]	Denotes optional arguments in syntax, such as `write_file [-format` *`fmt`*`]`
...	Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*
|	Indicates a choice among alternatives, such as `low | medium | high`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

SolvNet includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access SolvNet, go to the following address:

https://solvnet.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.

If you need help using SolvNet, click HELP in the top-right menu bar.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a support case to your local support center online by signing in to SolvNet at https://solvnet.synopsys.com, clicking Support, and then clicking “Open A Support Case.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at http://www.synopsys.com/Support/GlobalSupportCenters/Pages
- Telephone your local support center.
  - Call (800) 245-8005 from within North America.
  - Find other local support center telephone numbers at http://www.synopsys.com/Support/GlobalSupportCenters/Pages

1

Introduction to BSD Compiler

This chapter introduces the BSD Compiler boundary-scan design and verification flow.

This chapter includes the following sections:

- BSD Compiler Features and Benefits
- Using the BSD Compiler Design Flow
- Using the BSD Compiler Verification Flow

BSD Compiler Features and Benefits

BSD Compiler supports the following features for the IEEE Std 1149.1:

- Inserting boundary-scan components
- Verifying boundary-scan designs
- Compliance checking
- BSDL and pattern generation

BSD Compiler supports the following features for the IEEE Std 1149.6:

- Inserting boundary-scan components
- Verifying boundary-scan designs
- BSDL and pattern generation

The design resulting from verification can be synthesized with the core design, creating a unified design. You can generate functional, leakage, and DC parametric test vectors for your boundary-scan design. You can invoke BSD Compiler commands from the `dc_shell` command line or in the Design Vision command window.

BSD Compiler allows you to generate your boundary-scan design, specify boundary-scan components and preview them before you synthesize the design. No separate synthesis of boundary-scan components of BSD mode logic is required because synthesis occurs automatically when you insert the boundary-scan logic.

The compliance checker ensures that your design complies with IEEE Std 1149.1 and prepares your design for BSDL generation, functional testbench generation, and pattern generation. In the process of ensuring design compliance, the compliance checker identifies functional components of the design that violate the standard.

The BSDL generator provides an easy-to-read BSDL description that describes the organization of the boundary-scan logic and specifies the implemented boundary-scan instructions.

BSD Compiler provides the following features and benefits:

- Allows the synthesis of boundary scan design with the core design
- Reduces the need for extensive simulation to verify that the design conforms to IEEE Std 1149.1
- Provides an easy-to-use interface for verifying the boundary-scan logic in the Synopsys environment

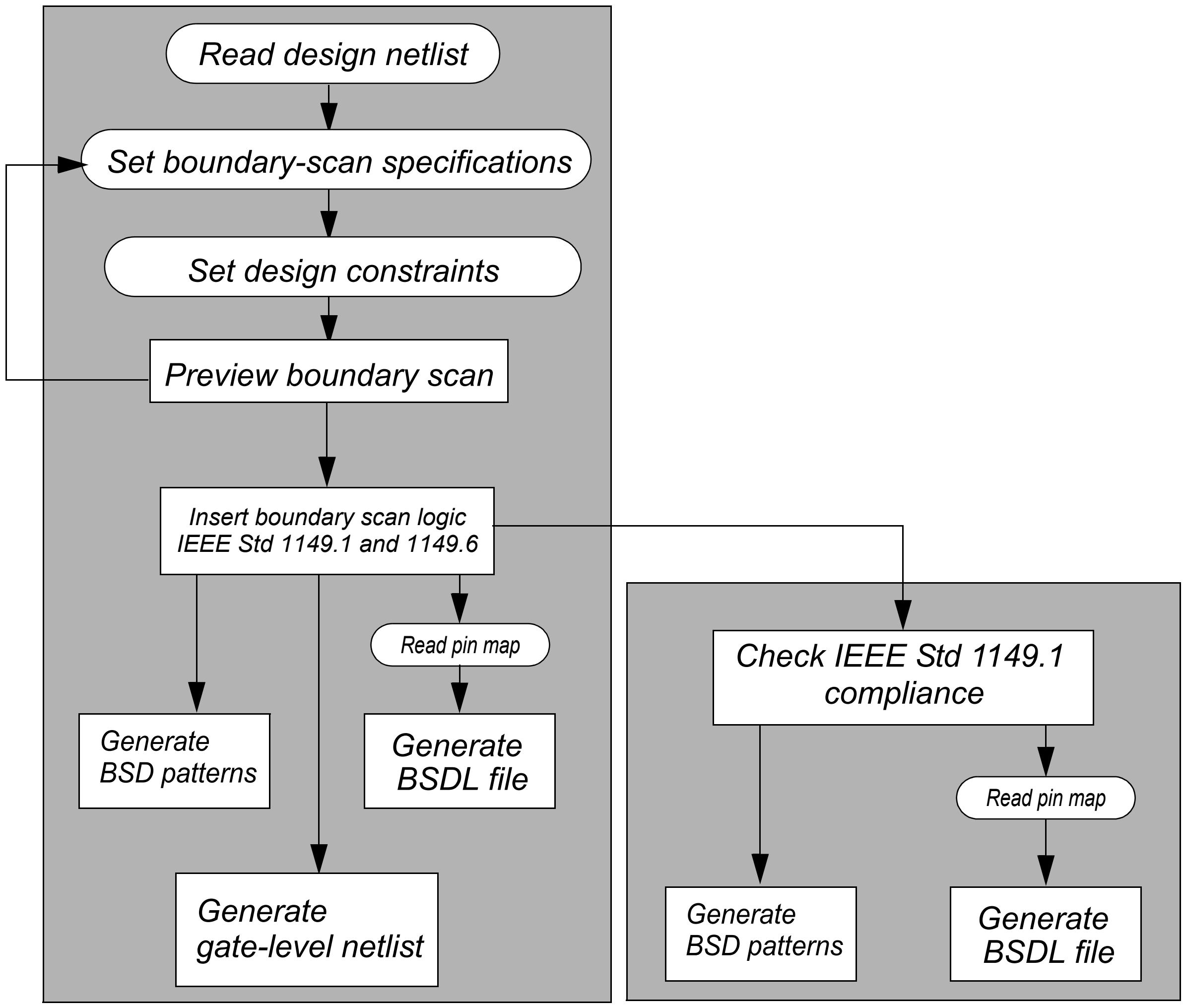
- Provides verification that supports any boundary-scan design; you can infer instructions, check for compliance, and generate patterns and BSDL file for designs that already incorporate boundary-scan logic and those that were not inserted by BSD Compiler and complies with IEEE Std 1149.1
- Supports IEEE Std 1149.6 logic insertion and BSDL and test vectors for the logic
- Supports both synchronous and asynchronous boundary-scan implementations
- Supports user-defined test data registers and user-defined instructions
- Supports user-defined boundary-scan register cells and TAP controller
- Supports customizing the implementation of the boundary-scan register
- Supports user-defined soft macro pad cells
- Supports linkage ports for analog port, power port, ground port, or ports driven by a black box cell
- Supports differential I/O pad cells
- Extracts boundary-scan information from designs that include an internal scan in any of the scan styles supported by DFT Compiler
- Generates the BSDL output file in an easy-to-read and ready-to-use format
- Enables the generation of boundary-scan test vectors

Using the BSD Compiler Design Flow

By using the BSD Compiler design flow provided in this section, you can generate a boundary-scan design, BSDL description for the design and patterns, boundary scan test vectors for IEEE Std 1149.1 and IEEE Std 1149.6. You can also verify the design for compliance with IEEE Std 1149.1.

Figure 1-1 on page 1-4 shows a typical BSD Compiler design and verification flow, which uses the synthesis capabilities of Design Compiler.

Figure 1-1 Typical BSD Compiler Design and Verification Flow



The BSD Compiler design flow with compliance checking in Figure 1-1 consists of the following steps:

1. Read the design netlist (RTL or gates) and technology libraries.
   - Define the interface for the core design.
   - Define pad cells for all inputs, outputs, and bidirectional ports on the top level of the design.
2. Set the boundary-scan design's specifications.
   - Enable BSD Compiler.
   - Select the required IEEE Std - 1149.1 or 1149.6.

- Identify all boundary-scan test access ports and assign their attributes.
- Identify linkage ports.
- Specify the compliance-enable pattern.
- Define clock ports.
- Define the boundary-scan register configuration, and assign all boundary-scan register instructions.
- Configure the device identification register and an optional user code capture value.
- Customize the boundary-scan register as needed.
- Select boundary-scan cells from default cell types or from your user-defined cell types.
- (Optional) Read the pin map as a BSD specification.

3. Preview the boundary-scan circuitry.
4. Generate the boundary-scan circuitry.
5. Generate BSDL file, BSD functional, leakage, and DC parametric test vectors.
6. Write the BSD inserted gate-level netlist (optional).
7. Run the IEEE Std 1149.1 compliance checker.
8. Read the port-to-pin map file.
9. Generate BSDL file, BSD functional, leakage, and DC parametric test vectors.

Example 1-1 illustrates a standard design flow using BSD Compiler.

Example 1-1 Standard Design Flow Using BSD Compiler

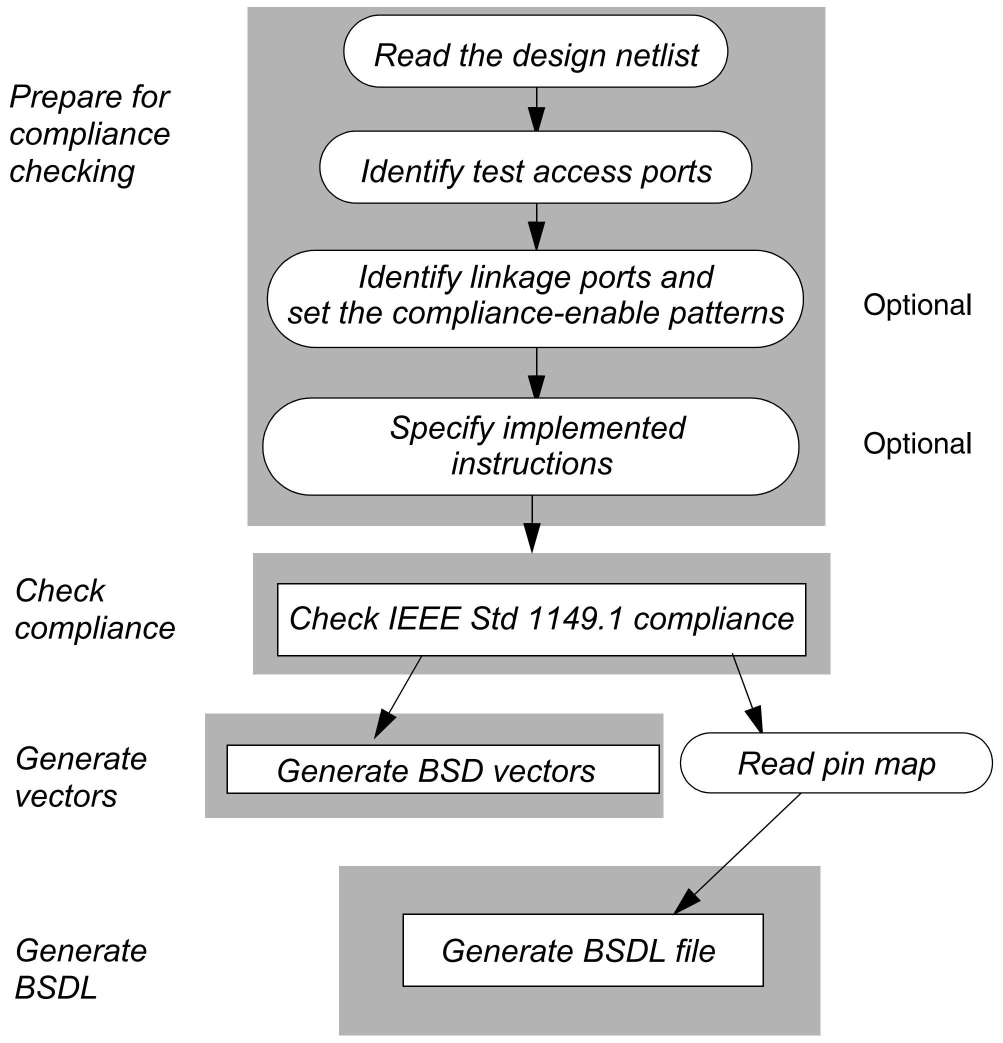
```
set company           {Synopsys Inc}
set search_path       [list "." $search_path]
set target_library    [list class.db]
set synthetic_library [list dw_foundation.sldb]
set link_library      [concat "*" $target_library $synthetic_library]
read_file -format verilog my_TOP.v
link
current_design TOP
set_dont_touch UP_CORE
# prevent DC to optimized the IO pad cells
set_dont_touch U*
#enable BSD Compiler
set_dft_configuration -bsd enable -scan disable
set_dft_signal -view spec -type tdi -port tdi
set_dft_signal -view spec -type tdo -port tdo
set_dft_signal -view spec -type tck -port tck
set_dft_signal -view spec -type tms -port tms
set_dft_signal -view spec -type trst -port trst_n
# Define linkage ports, no BSR inserted on these ports
#set_bsd_linkage_port -port_list [list en]
# Define compliance enable ports, no BSR inserted for these ports
set_bsd_compliance -name pattern_name_1 -pattern [list en 1]
# optional if BSR order is required
read_pin_map pin_map.txt
set_bsd_configuration -default_package my_package
set_bsd_configuration -asynchronous_reset true \
                      -ir_width 4 \
                  -check_pad_designs all \
                  -style synchronous \
                  -instruction_encoding binary
# To force BSRs on specific I/O pins using an I/O lib cell
#define_dft_design -type PAD -design_name my_lib_cell \
#                  -interface [ list data_in TA H enable TEN L  port Z H
] \
#     -params {  $pad_type$ string tristate_output \
                 $lib_cell$ string true }
# To force BSRs on specific I/O pins using a softmacro cell modeling
# the pin function of the I/O lib cell with structural verilog using
standard lib cell.
#define_dft_design -type PAD -design_name my_softmacro_cell \
#      -interface [ list data_out ZI H data_in A H enable EN \
        H port ZO H ] \
#      -params {  $pad_type$ string bidirectional $lib_cell$ \
        string false }
# To modify default TCK timing
set test_default_period 100
set test_bsd_default_strobe 95
set test_bsd_default_strobe_width 0
set test_bsd_default_delay 0
set test_bsd_default_bidir_delay 0
# All clocks must be define to insert a BC_4
create_clock clk -period 100 -waveform [list 20 30 ]
```

```
# To override the default BSR for inputs.
# INTEST not supported for inputs with BC_4 cells (1149.1)
set_boundary_cell -class bsd -type BC_4  -ports [list in0]
set_bsd_instruction -view spec [list EXTEST]  -code [list 1010] \
     -register BOUNDARY
set_bsd_instruction -view spec [list BYPASS]  -code [list 1111] \
     -register BYPASS
set_bsd_instruction -view spec [list HIGHZ]   -code [list 0001] \
     -register BYPASS
# These two instructions can be merge by using the same op-code
set_bsd_instruction -view spec [list SAMPLE]  -code [list 1100] \
     -register BOUNDARY
set_bsd_instruction -view spec [list PRELOAD] -code [list 1100] \
     -register BOUNDARY
set_bsd_instruction -view spec [list IDCODE]  -code [list 1011] \
     -register DEVICE_ID -capture_value 32'h10002007
# USERCODE supported with the new DW_tap_uc component
set_bsd_instruction -view spec [list USERCODE]  -code [list 1101] \
     -capture_value [list 4'b0001 16'b1111111100000000 12'h2ab ]
#set_bsd_instruction -view spec INTEST  -input_clock_condition TCK \
     -code [list 0001] -output_condition BSR
#set_bsd_instruction INTEST -code [list 0001] -input_clock_condition PI \
     -output_condition HIGHZ
# Need to specify the waiting time for BSDL
# define User instructions
set_bsd_instruction -view spec UDI1  -code [list 0011] -register BYPASS
# define Private instructions
set_bsd_instruction -view spec PRV1  -private -code [list 0111] \
     -register BOUNDARY
set_bsd_instruction -view spec PRV1  -private -code [list 0110] \
     -register BYPASS
# bsdc reports
preview_dft -bsd tap
preview_dft -bsd cells
preview_dft -bsd data_registers
preview_dft -bsd instructions
preview_dft -script
preview_dft -bsd all
insert_dft
## write the bsd inserted netlist
change_names -rules verilog -hierarchy
write -format ddc -output top_bsd.ddc -hier
write_bsdl -naming_check BSDL -output TOP.bsdl
## can write patterns after bsd insertion
create_bsd_patterns
write_test -format stil_testbench -output top_stil_tb
write_test -format wgl_serial -output top_wgl_tb
write_test -format verilog -output top_verilog_tb
check_bsd  -verbose
#check_bsd  -verbose -infer_instructions true
## bsd patterns and BSDL file can be generated after compliance check
using the same commands.
exit
```

Using the BSD Compiler Verification Flow

Figure 1-2 shows a BSD Compiler verification flow that does not use the synthesis capabilities of Design Compiler. Boundary scan was already incorporated into the design before entering this verification flow.

Figure 1-2 BSD Compiler Verification Flow


Prepare for compliance checking
Read the design netlist
Identify test access ports
Identify linkage ports and set the compliance-enable patterns
Optional
Specify implemented instructions
Optional
Check compliance
Check IEEE Std 1149.1 compliance
Generate vectors
Generate BSD vectors
Read pin map
Generate BSDL
Generate BSDL file

The BSD Compiler verification flow in Figure 1-2 consists of the following steps:

1. Read the boundary-scan design's gate-level netlist and synthesis technology library.
2. Enable BSD Compiler.

3. Identify the test access ports that drive the IEEE Std 1149.1 test signals.
4. Identify linkage ports.
5. Set BSD compliance-enable patterns.
6. Define clock ports.
7. Specify implemented instructions.
8. Run the IEEE Std 1149.1 compliance checker.

   Note:
   If you want to specify the opcode manually, use `check_bsd -infer_instructions false` (the default). If you want BSD Compiler to automatically extract your design instruction's opcode, use `check_bsd -infer_instructions true`. To execute BSD Compiler according to IEEE Std 1149.1a-1993 standards, use `set_bsd_configuration -std {-ieee1149.1_1993} enable`. By default, `check_bsd` checks compliance against IEEE Std 1149.1-2001.

9. Generate BSD functional, leakage, and DC parametric test vectors.
10. Read the port-to-pin mapping file.
11. Generate your BSDL file.

For more information on the verification flow for your design, see Chapter 4, "Verifying the Boundary-Scan Design."

Example 1-2 illustrates a standard verification flow using BSD Compiler.

Example 1-2 Standard Verification Flow Using BSD Compiler

```
set company {Synopsys Inc}
set search_path {"." $search_path}
set target_library "class.db"
set synthetic_library {dw_foundation.sldb}
set link_library [concat "*" $target_library
$synthetic_library]

#XG inserted design
read_file -format verilog TOP_xgbsd.v
#read_file -format ddc TOP_xgbsd.ddc

link
current_design TOP

# Boundary-scan option enable
set_dft_configuration -bsd enable -scan disable

# specify TAP ports
set_dft_signal -view existing_dft -type TDI -port tdi
set_dft_signal -view existing_dft -type TDO -port tdo
set_dft_signal -view existing_dft -type TCK -port tck -timing
{45 55}
set_dft_signal -view existing_dft -type TMS -port tms
set_dft_signal -view existing_dft -type TRST -port trst_n

## To specify tck timing for the patterns file
#set test_default_period 100
#set_dft_signal -view existing_dft -type TCK -port tck -
timing
{45 55}

## specify system clocks
create_clock clk -period 50 -waveform {15 35}

## specify implemented instructions
set_bsd_instruction {EXTEST} -code {1100} -register BOUNDARY \
     -view existing_dft
set_bsd_instruction {SAMPLE} -code {1110} -register BOUNDARY \
     -view existing_dft
set_bsd_instruction {PRELOAD} -code {1110} -register BOUNDARY -view
existing_dft
set_bsd_instruction {BYPASS} -code {1111} -register BYPASS \
     -view existing_dft

## specify implemented user instructions
set_bsd_instruction {UDI_1} -code {1001} -register BYPASS \
     -view existing_dft
set_bsd_instruction {UDI_2} -code {1011} -register BOUNDARY \
     -view existing_dft

## specify implemented private instruction
set_bsd_instruction -private {PDI_1} -code {0001} -register BYPASS \
     -view existing_dft
set_bsd_instruction -private {PDI_2} -code {0010} -register BOUNDARY \
     -view existing_dft

## specify implemented optional IDCODE and USERCODE
instructions
set_bsd_instruction {IDCODE} -code {1010} \
     -capture_value {32'b10000001000111010111000011110111} -register \
      DEVICE_ID -view existing_dft
set_bsd_instruction {USERCODE} -code {1101} \
     -capture_value {32'b00011100111110000000010101010010} -register \
     DEVICE_ID -view existing_dft

## set compliance enable patterns
set_bsd_compliance -name pat1 -pattern {tm 1}

## specify linkage ports
set_bsd_linkage_port -port_list {in0}

## check for compliance to the 1149.1, and infer the instructions
#check_bsd -verbose -infer_instructions true
```

```
#check for compliance to the 1149.1

check_bsd -verbose
########################################################
## For BSDL full design package
read_pin_map pin_map1.txt
set_bsd_configuration -default_package my_package1
write_bsdl -naming_check BSDL -output TOP_xgpkg1.bsdl

## optional to generate patterns for private instructions
#set test_bsdvec_create_patterns_for_private_instructions
true

## generating BSD patterns
create_bsd_patterns
write_test -format stil
write_test -format stil_testbench
## generating wgl_serial patterns pck1 and verilog patterns
write_test -format wgl_serial
write_test -format verilog

########################################################
## For BSDL reduced package with No connect ports (NC)
read_pin_map pin_map2.txt
set_bsd_configuration -default_package my_package2
write_bsdl -naming_check BSDL -output TOP_xgpkg2.bsdl

## generating BSD patterns
create_bsd_patterns
write_test -format stil -output TOP_stil_2_tb
write_test -format stil_testbench -output TOP_stil_2_pkg2
## generating wgl_serial patterns pck2
write_test -format wgl_serial -output TOP_wgl_serial_pkg2
write_test -format verilog -output TOP_verilog_pkg2
########################################################
exit
```

Using the -view Option

Use the `set_dft_signal -view spec` command when you plan to use BSD Compiler to insert your boundary-scan logic. If you already have boundary-scan logic inserted and are merely trying to identify your TAP ports for the verification flow, see the `set_dft_signal -view existing_dft` command in “Defining IEEE Std 1149.1 Test Access Ports (TAPs)” on page 2-23.

2

Inserting Boundary-Scan Components

This chapter describes how to insert boundary-scan components into your top-level design, which includes setting your boundary-scan specifications, previewing the scan logic, and generating the boundary scan design. BSD Compiler supports writing patterns after generating the boundary scan design.

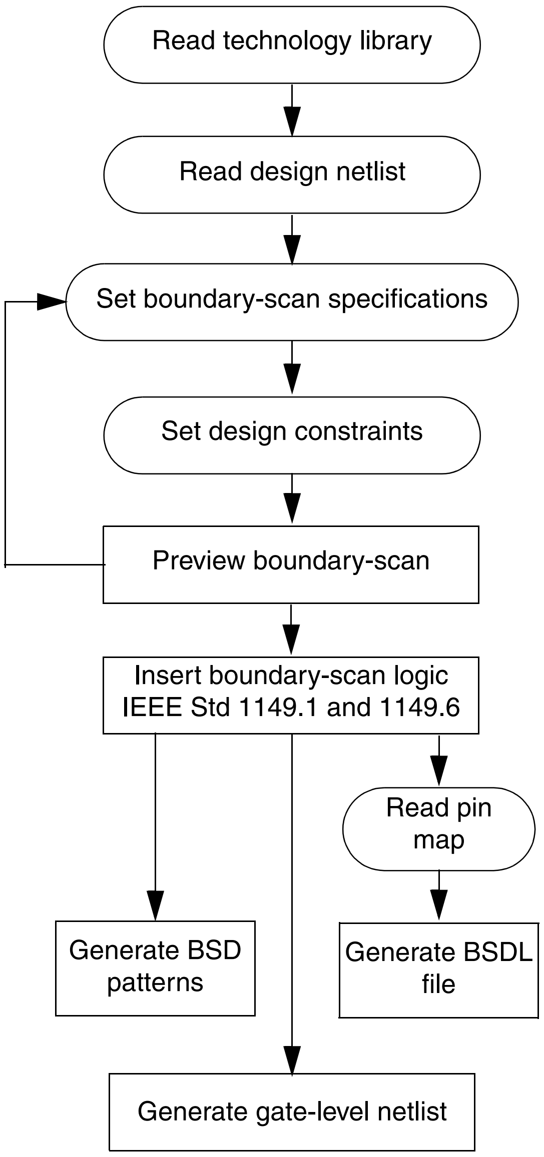
This chapter includes the following sections:

- Reading the RTL or Gate-Level Netlist
- Specifying Complex Pad Designs
- RTL Generation Flow

Design Flow for Inserting Boundary-Scan Components

The design flow for inserting boundary-scan components is shown in Figure 2-1.

Figure 2-1 Design Flow for Inserting Boundary-Scan Components



Insert boundary-scan components by using the following design flow:

1. Read technology library. See “Setting Up the Design Environment” on page 2-4.
2. Generate BSDL file, BSD functional, leakage, and DC parametric test vectors.
3. Write the BSD inserted gate-level netlist (optional).
4. Read the design’s RTL or gate-level netlist. See “Reading the RTL or Gate-Level Netlist” on page 2-70.
5. Set boundary-scan specifications. See “Setting Boundary-Scan Specifications” on page 2-23.
6. If boundary-scan cell order is required, read the pin map as a BSD specification (optional).
7. Preview the boundary-scan design. See “Previewing the Boundary-Scan Design” on page 2-64.
8. Generate a new design including boundary scan. See “Generating the Boundary-Scan Design” on page 2-65.

Setting Up the Design Environment

Use the following Synopsys system variables to define the key parameters of your design environment:

`search_path` – A list of alternate directory names to search to find the `link_library`, `target_library`, and design files.

`target_library` – Usually the same as your link library, unless you are translating a design between technologies.

`link_library` – The ASIC vendor libraries where your pad cells and core cells are initially represented. You should also include your DesignWare library in this variable definition.

Note:
To learn more about library variables and search paths, see the Design Compiler documentation.

The following commands illustrate this:

```
dc_shell> set search_path {. $search_path}
dc_shell> set target_library asic_vendor.db
dc_shell> set synthetic_library {dw_foundation.sldb}
dc_shell> set link_library [ concat * \
               $target_library] $synthetic_library ]
```

Specifying BSR Segment Pad Cells

Use the `define_dft_design -interface` command to specify the signal types for pad cells, pad cells with test receivers (RX), and BSR segments. The signal types listed in Table 2-1 support pads, pad with test receivers, and BSR segments.

Table 2-1 The define_dft_design -interface Option Signal Types That Support BSR Segment Pad Cells

Signal type	Polarity	Definition
ac_init_clk	high	Signal common to all test receivers used to load the hysteretic memory
ac_init_clk	low	Signal common to all test receivers used to load the hysteretic memory with inverted polarity
ac_mode	high	AC1/AC2 cell AC mode signal
ac_mode	low	AC1/AC2 cell AC mode with inverted polarity
ac_test	high	AC1/AC2 cell AC test signal
ac_test	low	AC1/AC2 cell AC test signal with inverted polarity
capture_clk	high	BSR cell capture stage clock
capture_clk	low	BSR cell capture stage clock with inverted polarity
capture_en	high	BSR cell capture stage load enable for synchronous style with inverted polarity
capture_en	low	BSR cell capture stage load enable for synchronous style with inverted polarity
highz	high	Highz pin for the BSR cells within the pad and BC7 mode_3
highz	low	Highz pin for the BSR cells within the pad with inverted polarity and BC_7 mode 3
mode_in	high	Input BC1/BC2 cell mode
mode_in	low	Input BC1/BC2 cell mode with inverted polarity
mode_out	high	Output BC1/BC2/AC1/AC2 cell mode

Table 2-1 The define_dft_design -interface Option Signal Types That Support BSR Segment Pad Cells (Continued)

Signal type	Polarity	Definition
mode_out	low	Output BC1/BC2/AC1/AC2 cell mode with inverted polarity
mode1_inout	high	BC7 cell mode1
mode1_inout	low	BC7 cell mode1 with inverted polarity
mode2_inout	high	BC7 cell mode2
mode2_inout	low	BC7 cell mode2 with inverted polarity
shift_dr	high	BSR cell serial data enable
shift_dr	low	BSR cell serial data enable with inverted polarity
si	high	BSR cell serial data in
si	low	BSR cell serial data in with inverted polarity
so	high	BSR cell serial data out
so	low	BSR cell serial data out with inverted polarity
update_clk	high	BSR cell update stage clock
update_clk	low	BSR cell update stage clock with inverted polarity
update_en	high	BSR cell update stage load enable for synchronous style
update_en	low	BSR cell update stage load enable for synchronous style with inverted polarity
port	high	Port associated to pad, excluded for pad type hybrid interface
port	low	Port inverted associated to pad, excluded for pad type hybrid interface

A boundary-scan cell and a pad cell can be physically bound in a single design module to form a BSR segment. The tool can automatically stitch this BSR segment to the top-level BSR chain. Use the `define_dft_design -params $bsr_segment$` parameter to describe the BSR segment.

The syntax is

```
-params {$bsr_segment$ list string "bsr_spec1"\
         "bsr_spec2"... $end_list$}
```

where `bsr_spec<n>` are BSR specifications for different segments, as follows:

`pos bsr_type pi po safe_val/dis_rslt cntrl_bsr_pos`

- *`pos`*: Identifies the position of the BSR cell within the pad cell BSR cell segment.
- *`bsr_type`*: Specifies the BSR cell type. Valid cells types are: BC1, BC2, BC4, BC7, AC1, AC2, AC_SelX, AC_SelU.
- *`pi`*: Specifies the primary input of the BSR cell in the BSR segment. Since the primary input of the boundary cell in a segment is internal, use the pin input of the pad cell as the `bsr_spec<n>`. If no primary input exists in the BSR-segment design, specify “-” for this value.
- *`po`*: Specifies the primary output of the BSR cell in the BSR segment. Since the primary output of the boundary cell in a segment is internal, use the pin output of the pad cell as the `bsr_spec<n>`. If no primary output exists in the BSR-segment design, specify “-” for this value.
- *`safe_val/dis_rslt`*: For input and two-state output BSR cells, specifies the safe value of the BSR cell. For tristate or bidirectional BSR cells, specifies the disable result of the port when the associated control BSR cell is in safe state. Allowed values are 0 1 X Z.
- *`cntrl_bsr_pos`*: Specifies the position of the control BSR cell associated with this BSR cell. If there is no control BSR cell, specify “-” for this value.

Note:

If the `$bsr_segment$` parameter is not used for a BSR segment pad cell, BSD Compiler adds BSR cells to the design for the pad cell during synthesis and the embedded BSR cells are not included in the generated BSDL or patterns. An IEEE Std 1149.1 BSR segment is validated only with the `check_bsd` command. However, an IEEE Std 1149.1 BSR segment can be validated by simulation after the `insert_dft` patterns are generated.

The `$bsr_segment$` parameter supports bus notation on pin names. The tool expands BSR segments that use bus notations within their specifications. The order of expanded BSR cells is the same as the order of specified BSR specifications.

The following example describes a 32-bit two-state output pad with embedded BSR cells.

```
define_dft_design...-interface {data_in di h port po h...} \
   -params { $bsr_segment$ list string \
             "0 BC1 di[0:31] po[0:31] X -" $end_list$}
```

The following example describes a 32-bit output pad with the first 16 bits supporting two-state output ports and the next16 bits supporting tristate output ports with a single control BSR cell.

```
define_dft_design...-interface {enable en h data_in di h \
    port po h...} \
   -params { $bsr_segment$ list string \
            "0 BC1 di[0:15] po[0:15] X -" \
            "1 BC2 en - X 1" \
            "2 BC1 di[16:31] po[16:31] Z 1 -" $end_list$}
```

BSR Segment Pad Cell Examples

The examples of this section present various pad cell designs in which the BSR cells associated with the pads are embedded inside the pad cells. All BSR cells embedded in the pads are connected together to form a BSR chain segment. All other interface signals of BSR cells are brought up to the pad cell interface.

The following examples are simple BSR embedded pad designs assumed to use asynchronous-style BSR cells.

Figure 2-2 on page 2-9 and the following example script show an input pad cell with an embedded control-and-observe BSR cell.

```
define_dft_design -design in_pad1 -type PAD \
     -interface {port pi h data_out po h si \
      si h so so h capture_clk cclk h update_clk \
      uclk h mode_in mode2 h} \
     -params {$pad_type$ string input $bsr_segment$ \
      list string "0 BC2 pi po X -" $end_list$}
```

Figure 2-2 Input Pad Cell

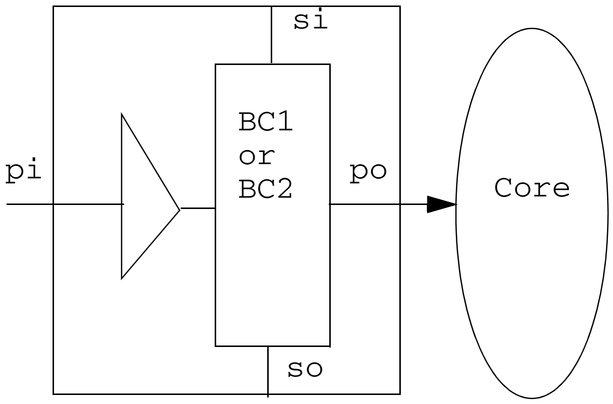


Figure 2-3 and the following script show an input pad cell with an observe-only BSR cell.

```
define_dft_design -design in_pad2 -type PAD \
     -interface {port pi h data_out po h si \
      si h so so h capture_clk cclk h} \
     -params {$pad_type$ string input $bsr_segment$ \
      list string "0 BC4 pi - X -" end_list$}
```

Figure 2-3 Input Pad Cell With Observe-Only BSR Cell

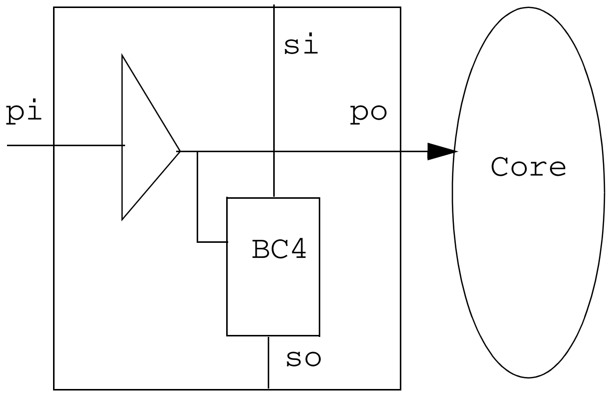


Figure 2-4 on page 2-10 and the following script show an output pad cell.

```
define_dft_design -design out_pad1 -type PAD \
     -interface {data_in pi h port po h si \
      si h so so h capture_clk cclk h update_clk \
      uclk h mode_out mode1 h} \
     -params {$pad_type$ string output $bsr_segment$ \
      list string "0 BC1 pi po X -" $end_list$}
```

Figure 2-4 Two-State Output Pad Cell

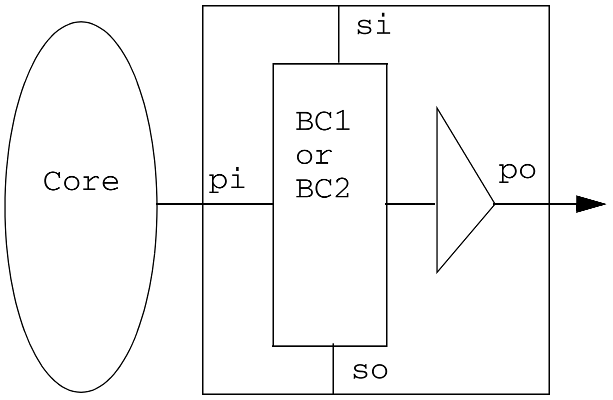


Figure 2-5 and the following script show a tristate output pad cell with control and data BSR cells.

```
define_dft_design -design out_pad2 -type PAD \
     -interface {data_in pi h port po h enable en h si \
      si h so so h capture_clk cclk h update_clk \
      uclk h mode_out mode1 h} \
     -params {$pad_type$ string tristate_output \
      $bsr_segment$ list string "0 BC2 en po 1" \
      "1 BC1 pi po Z 0 -" $end_list$}
```

Figure 2-5 Tristate Output Pad Cell

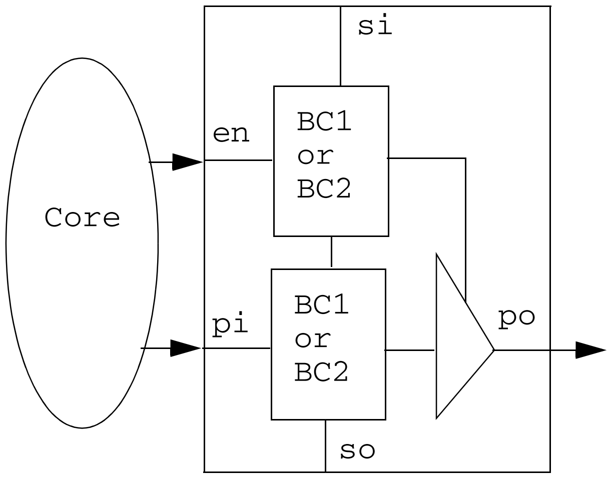


Figure 2-6 and the following script show a bidirectional pad cell with control and BC7 BSR cells.

```
define_dft_design -design bidi_pad1 -type PAD \
     -interface {data_in pi h data_out po2 h port po h  \
      enable en h si si h so so h capture_clk cclk  \
      h update_clk uclk h mode1_inout mode 1 h \
      mode2_inout mode2 h}
     -params {$pad_type$ string bidirectional \
      $bsr_segment$ list string "0 BC2 en po 1" \
      "1 BC7 pi po Z 0 -" $end_list$}
```

Figure 2-6 Bidirectional Pad With BC7 Cell

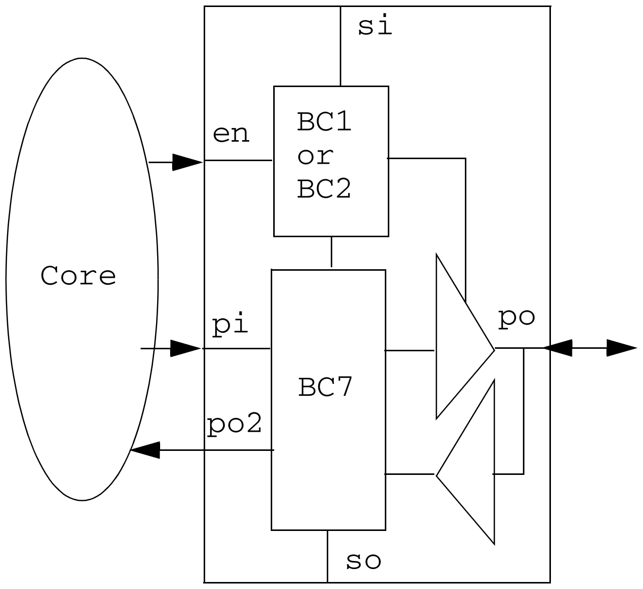
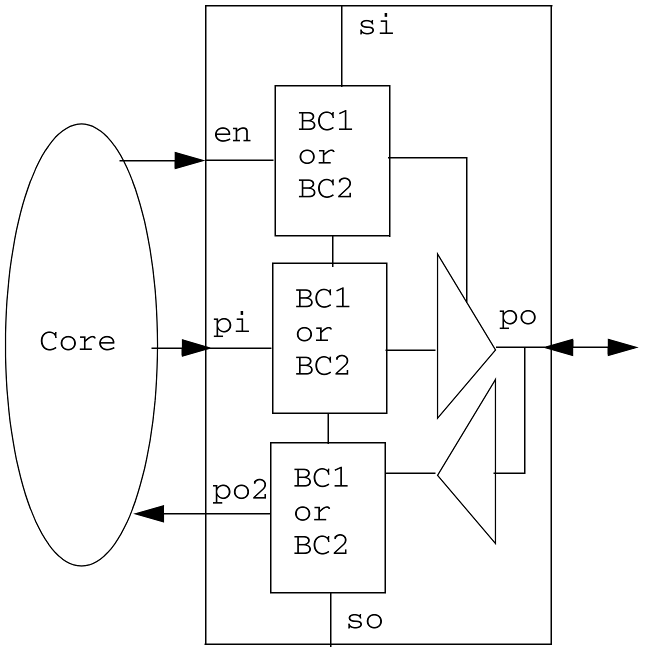


Figure 2-7 on page 2-12 and the following script show a bidirectional pad cell with control and separate input and output BSR cells.

```
define_dft_design -design bidi_pad2 -type PAD \
     -interface {data_in pi h data_out po2 h port po h  \
      enable en h si si h so so h capture_clk cclk  \
      h update_clk uclk h mode_out mode1 h \
      mode_in mode2 h}
     -params {$pad_type$ string bidirectional \
      $bsr_segment$ list string "0 BC2 en po 1" \
      "1 BC1 pi po Z 0" "2 BC2 po po2 X -" $end_list$}
```

Figure 2-7 Bidirectional Pad With Separate Input and Output BSR Cells



The following examples demonstrate complex BSR segment pad designs assumed to use synchronous-style BSR cells.

Figure 2-8 on page 2-13 and the following script show an input differential pad cell that has an observe-only BSR cell for each leg of the pad to observe the test receiver and a control-and-observe BSR cell.

```
define_dft_design -design diff_pad1 -type PAD \
     -interface {port ppi h port_inverted npi h data_out \
      po h si si h so so h capture_clk cclk h capture_en cen h \
      update_clk uclk h update_en uen h mode_in mode2 h } \
     -params {$pad_type$ string input $differential$ string \
       true $bsr_segment$ list string "0 BC4 ppi - X"  \
      "1 BC2 ppi po X" "2 BC4 npi - X -" $end_list$}
```

Figure 2-8 Input Differential Pad Cell

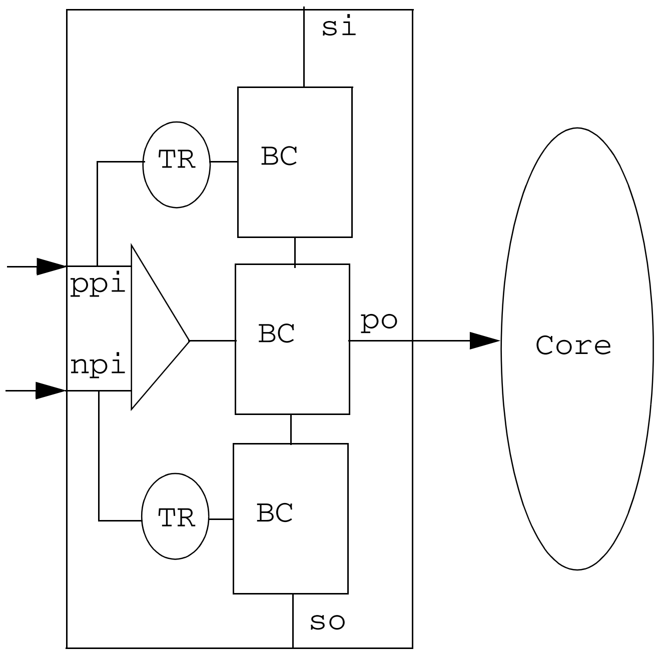


Figure 2-9 on page 2-14 and the following script show a tristate output pad cell that has a control BSR cell and AC BSR cells for data and ac/bc selection. Such pads are typically used in IEEE Std 1149.6 designs.

```
define_dft_design -design diff_pad2 -type PAD \
     -interface {port ppo h port npo 1 data_in \
       pi h enable en h si si h so so h capture_clk cclk h \
      capture_en cen h update_clk uclk h update_en uen h \
      mode_out mode1 h ac_test act h ac_mode acm h} \
     -params {$pad_type$ string output \
      $differential$ string true \
      $bsr_segment$ list string "0 BC2 en ppo 1"  \
      "1 AC1 pi ppo Z 0" "2 AC_SelU - ppo X -" $end_list$}
```

Figure 2-9 Tristate Output Differential Pad Cell With AC BSR Cells

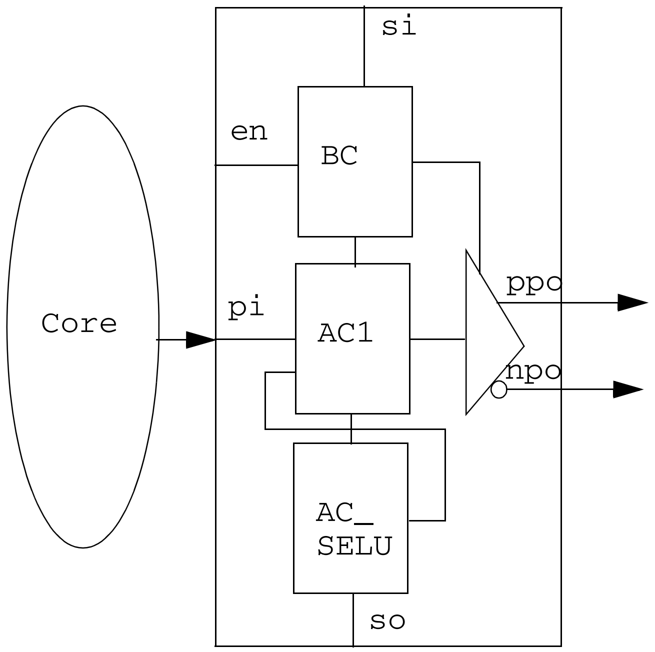


Figure 2-10 on page 2-15 and the following script show a bidirectional differential pad cell that shows multiple BSR cells.

```
define_dft_design -design diff_pad2 -type PAD \
     -interface {port ppo h port npo 1 data_in \
      pi h data_out po2 h enable en h si si h so so h \
      capture_clk cclk h capture_en cen h update_clk uclk h \
      update_en uen h mode_out mode1 h mode_in mode2 ac_test act \
      h ac_mode acm h} \
     -params {$pad_type$ string bidirectional \
      $differential$ string true \
      $bsr_segment$ list string "0 BC2 en ppo 1"   \
      "1 AC1 pi ppo Z 0" "2 AC_SelU - ppo X"  \
      "3 BC4 npo - X" "4 BC2 ppo po2 X"  \
      "5 BC4 npo - X -" $end_list$}
```

Figure 2-10 Bidirectional Differential Pad Cell With AC BSR Cells

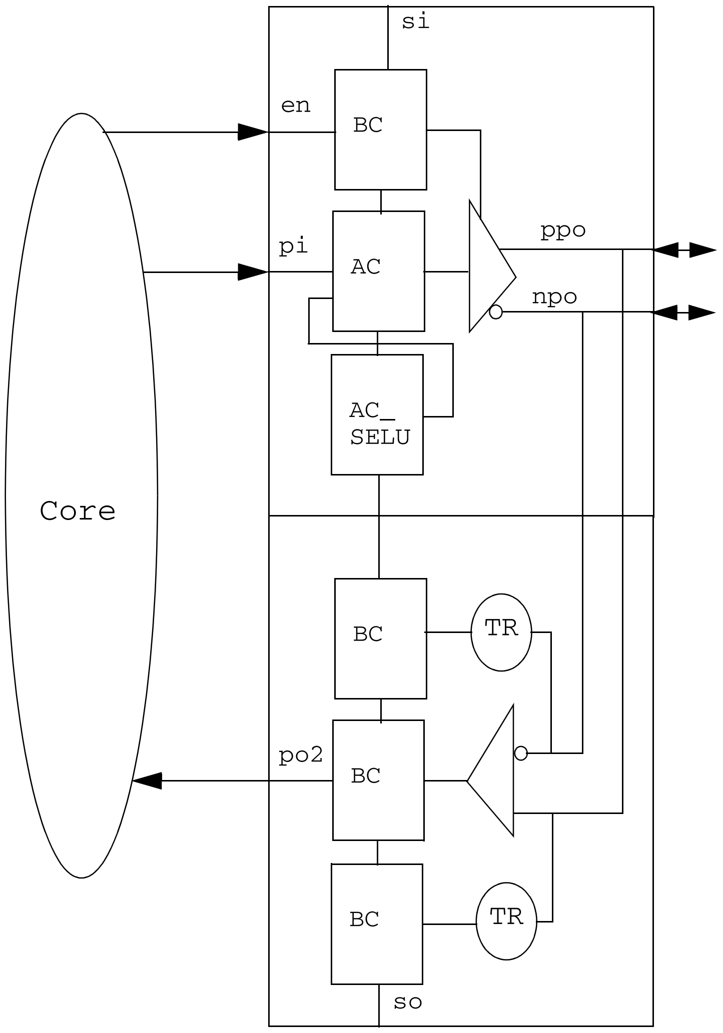


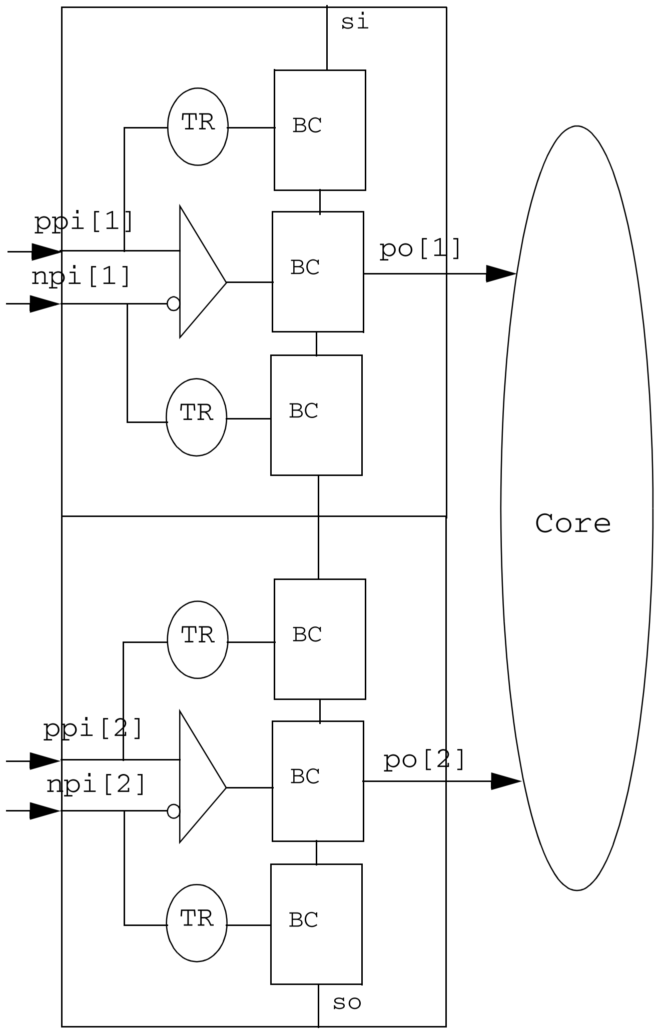
Figure 2-11 on page 2-16 and the following two scripts show a multibit differential input pad cell block with BSR cells associated with each bit of the pad cell.

```
define_dft_design -design diff_pad3 -type PAD \
     -interface {port ppi h port npi 1 data_out \
      po h si si h so so h \
      capture_clk cclk h capture_en cen h update_clk uclk h  \
      update_en uen h mode_in mode2 h} \
     -params {$pad_type$ string input \
      $differential$ string true \
      $bsr_segment$ list string "0 BC4 ppi[1:2]- X" \
      "1 BC2 pp1[1:2] po[1:2] X" "2 BC4 npi[1:2] - X -" $end_list$}
```

and

```
define_dft_design -design diff_pad3 -type PAD \
     -interface {port ppi h port npi 1 data_out \
      po h si si h so so h \
      capture_clk cclk h capture_en cen h update_clk uclk h  \
      update_en uen h mode_in mode2 h} \
     -params {$pad_type$ string input \
      $differential$ string true \
      $bsr_segment$ list string "0 BC4 ppi[1]- X"  \
      "1 BC2 ppi[1] po[1] X" "2 BC4 npi[1] - X"  \
      "3 BC4 ppi[2] - X" "4 BC2 ppi[2] po[2] X"  \
      "5 BC4 npi[2] - X -" $end_list$}
```

Figure 2-11 Multibit Input Differential Pad Cell



Multiport and Multipad Support for the $bsr_segment$ Parameter

BSD Compiler supports the `$bsr_segment$` parameter for multiports and multipads as well as multibit pad support.

Note that this terminology is used in the following way:

- Multiport refers to a design with many ports
- Multipad refers to the pads associated with the multiple ports
- Multibit refers to a multiport specification as a bus array [0:3]

The hybrid pad_type is added as part of the `$bsr_segment$` parameters specification of the `define_dft_design` command. Also, to specify the multiple ports for multibit differential pad type hybrids such as Peripheral Component Interconnect Express (PCIe) and Serialize/Deserialize (SerDes) pads, use the option `-param $diff_port_pairs$` of the command. The port information specified for each `$bsr_segment$` parameter is used to identify the port associated to the BSRs in the `$bsr_segment$` parameter descriptions.

Example 2-1 shows a script for a multiport `$bsr_segment$` parameter that uses the hybrid pad type with five ports associated to a bsr segment. Each port has its own BSR cell attached to the pad with the respective `$bsr_segment$` parameter specifications.

In this example, the `-interface` signals for multiport/multipad specification include only the TAP FSM signals that are to be hooked up automatically to the top level BSR chain. The `-params` specification includes the pad type option hybrid, indicating that the design is a multiport-multipad bsr segment, followed by the `$bsr_segment$` parameter specification for each pad associated to a port.

Example 2-1 Script for a Multiport With a 5-Pads $bsr_segment$

```
define_dft_design -design_name MY_BIP_DES -type PAD \
     -interface {\
          capture_clk capture_clk h \
          capture_en capture_en h \
          update_clk update_clk l \
          update_en update_en h \
          shift_dr shift_dr h \
          si si h \
          so so h \
          mode1_inout mode1 h \
          mode2_inout mode2 h \
          mode_in          mode_in h \
          mode_out       mode_out h \
         }\
     -params {$pad_type$ string hybrid \
          $bsr_segment$ list string \
          "0 BC1 bip_addr_[2] bip_addr[2] X -" \
          "1 BC1 bip_addr_[1] bip_addr[1] X -" \
          "2 BC1 bip_addr_[0] bip_addr[0] X -" \
          "3 BC2 rstn rstn_ X -" \
          "4 BC1 oen dinout 1 -" \
          "5 BC7 data dinout Z 4" $end_list$ }
```

Example 2-2 shows a script for a 4-bit multibit PCIe differential pad with the respective bsr segment attached to the differential SerDes of the pad. The `-interface` signals for the multiport specification include only the TAP FSM signals that are to be hooked up automatically to the top-level BSR chain. The `-params` specification includes the pad type option hybrid, indicating that the design is a multibit bsr segment, followed by the `$bsr_segment$` parameter specification for each port bit. Note that the script uses `-param $diff_port_pairs$` to specify the ports associated to the SerDes differential pads of the differential PCIe pad. Each bit of the multibit has its own bsr segment associated to the differential port.

Example 2-2 Script for a 4-Bit Multibit Differential PCIe Bus Interface bsr_segment

```
## 4 bit PCIe DIFF_RX_TX bsr_segment
define_dft_design -type PAD \
-design_name DIFF_RX_TX_PCIe \
-interface { \
            si          tt_si H \
            so          tt_so H \
            capture_clk tt_capture_clk H \
            capture_en  tt_capture_en H \
            update_clk  tt_update_clk H \
            update_en   tt_update_en  H \
            highz       tt_highz H \
            shift_dr    tt_shift_dr H \
            mode_out    tt_mode_o H \
            ac_init_clk tt_ac_init_clk H \
            ac_mode     tt_AC_Mode H   \
            ac_test     tt_AC_test H } \
-params { $pad_type$ string hybrid $differential$ boolean true \
      $diff_port_pairs$ list string \
            "tt_diffrx_in_pos[0]  tt_diffrx_in_neg[0]" \
            "tt_diffrx_in_pos[1]  tt_diffrx_in_neg[1]" \
            "tt_diffrx_in_pos[2]  tt_diffrx_in_neg[2]" \
            "tt_diffrx_in_pos[3]  tt_diffrx_in_neg[3]" \

            "tt_difftx_out_pos[0] tt_difftx_out_neg[0]" \
            "tt_difftx_out_pos[1] tt_difftx_out_neg[1]" \
            "tt_difftx_out_pos[2] tt_difftx_out_neg[2]" \
            "tt_difftx_out_pos[3] tt_difftx_out_neg[3]" \
$end_list$ \
$bsr_segment$ list string \
          "0 BC4 tt_diffrx_in_pos[3] - X -" \
          "1 BC4 tt_diffrx_in_neg[3] - X -" \
          "2 BC1 tt_difftx_oe_in[3]    tt_difftx_out_pos[3] 1 -" \
          "3 AC1 tt_difftx_data_in[3]  tt_difftx_out_pos[3] Z 2" \

          "4 BC4 tt_diffrx_in_pos[2] - X -" \
          "5 BC4 tt_diffrx_in_neg[2] - X -" \
          "6 BC1 tt_difftx_oe_in[2]   tt_difftx_out_pos[2] 1 -" \
          "7 AC1 tt_difftx_data_in[2] tt_difftx_out_pos[2] Z 6" \

          "8 BC4 tt_diffrx_in_pos[1] - X -" \
          "9 BC4 tt_diffrx_in_neg[1] - X -" \
          "10 BC1 tt_difftx_oe_in[1]   tt_difftx_out_pos[1] 1 -" \
          "11 AC1 tt_difftx_data_in[1] tt_difftx_out_pos[1] Z 10" \

          "12 BC4 tt_diffrx_in_pos[0] - X -" \
          "13 BC4 tt_diffrx_in_neg[0] - X -" \
          "14 BC1 tt_difftx_oe_in[0]    tt_difftx_out_pos[0] 1 -" \
          "15 AC1 tt_difftx_data_in[0]  tt_difftx_out_pos[0] Z 14"
$end_list$ }
```

Netlist and Script Examples

Example 2-3 describes BSR embedded pad cells within a design named top.v1.

Example 2-3 Example for BSR Embedded Pad Cells

```
module UP_CORE ( i_clk,i_tm,i_en,i_in0,i_in1,i_in2,o_en0,
                 o_en1,o_en2,o_out0,o_out1,o_out2);
input i_clk,i_tm,i_en,i_in0,i_in1,i_in2;
output o_en0,o_en1,o_en2,o_out0,o_out1,o_out2;
assign i_en = o_en2;
endmodule

module TOP (clk,tm,en,tck,tms,tdi,trst_n,tdo, in0, in1,
            in2, out0,out1,out2 ); input
            tm,en,in0,in1,in2,clk,tck,tms,tdi,trst_n;
            output tdo,out0,out1,out2;

wire i_clk,i_tm,i_en,i_in0,i_in1,i_in2,o_en0,o_en1,
     o_en2,o_out0,o_out1,o_out2;

//wire t_si, t_so, t_shift_dr, t_capture_clk, t_update_clk,
       t_mode;

 IPAD U1 ( .A(trst_n),.Z() );
 IPAD U2 ( .A(tdi),.Z() );
 IPAD U3 ( .A(tms),.Z() );
 TRIOPAD U4 ( .A(),.E(), .PAD(tdo) );
 IPAD U5 ( .A(in0), .Z(i_in0) );
 IPAD U6 ( .A(tck),.Z() );
//IPAD U7 ( .A(in1), .Z(i_in1) );
 t_bc1 U7
 ( .t_in (in1),// bsrinpad_Input
 .t_capture_en (), // auto Stitch
 .t_capture_clk (), // auto Stitch
 .t_update_en (), // auto Stitch
 .t_update_clk (), // auto Stitch
 .t_shift_dr (), // auto Stitch
 .t_mode (), // auto Stitch
 .t_si (), // auto Stitch
 .t_so (), // auto Stitch
 .t_data_out (i_in1) // bsrinpad_Output
 );

// IPAD U8 ( .A(in2), .Z(i_in2) );
 t_bc2 U8
 ( .t_in (in2),// bsrinpad_Input
 .t_capture_en (), // auto Stitch
 .t_capture_clk (), // auto Stitch
 .t_update_en (), // auto Stitch
 .t_update_clk (), // auto Stitch
 .t_shift_dr (), // auto Stitch
 .t_mode (), // auto Stitch
 .t_si (), // auto Stitch
 .t_so (), // auto Stitch
 .t_data_out (i_in2) // bsrinpad_Output
 );

// IPAD U9 ( .A(clk), .Z(i_clk) );
 t_bc4 U9
 ( .t_in (clk),// bsrinpad_Input
 .t_capture_en (), // auto Stitch
 .t_capture_clk (), // auto Stitch
 .t_shift_dr (), // auto Stitch
 .t_si (), // auto Stitch
 .t_so (), // auto Stitch
 .t_data_out (i_clk) // bsrinpad_Output
 );

 IPAD U10 ( .A(tm), .Z(i_tm) );
 IPAD U11 ( .A(en), .Z(i_en) );

 TRIOPAD U12 ( .A(o_out0), .E(o_en0), .PAD(out0) );
 TRIOPAD U13 ( .A(o_out1), .E(o_en1), .PAD(out1) );
 TRIOPAD U14 ( .A(o_out2), .E(o_en2), .PAD(out2) );

 UP_CORE U15
(i_clk,i_tm,i_en,i_in0,i_in1,i_in2,o_en0,o_en1,
 o_en2,o_out0,o_out1,o_out2);

endmodule
```

```
=======================================

## Module t_bc1.v

module t_bc1 ( t_in, t_capture_clk, t_capture_en,
t_update_clk, \
 t_update_en, t_shift_dr, t_mode, t_si, t_data_out, t_so );

 input t_in, t_capture_clk, t_capture_en, t_update_clk,
  t_update_en, t_shift_dr, t_mode, t_s i; output t_data_out,
 t_so;

 wire i_data_in;

 IPAD UU1 ( .A(t_in), .Z(i_data_in) );

 DW_bc_1 my_bc1_inst ( .capture_clk(t_capture_clk),
.update_clk(t_update_clk),
 .capture_en(t_capture_en), .update_en(t_update_en),
.shift_dr(t_shift_dr), \
 .mode(t_mod e), .si(t_si), .data_in(i_data_in),
.data_out(t_data_out), \
 .so(t_so) );
endmodule

========================================

## Module t_bc2.v

module t_bc2 ( t_in, t_capture_clk, t_capture_en,
 t_update_clk, t_update_en,
 t_shift_dr, t_mode, t_si, t_data_out, t_so );

 input t_in, t_capture_clk, t_capture_en, t_update_clk,
 t_update_en,
 t_shift_dr, t_mode, t_s i; output t_data_out, t_so;

 wire i_data_in;

 IPAD UU2 ( .A(t_in), .Z(i_data_in) );

 DW_bc_2 my_bc2_inst ( .capture_clk(t_capture_clk),
.update_clk(t_update_clk),
 .capture_en(t_capture_en), .update_en(t_update_en),
.shift_dr(t_shift_dr),
 .mode(t_mod e), .si(t_si), .data_in(i_data_in),
.data_out(t_data_out),
 .so(t_so) );
endmodule
```

```
==========================================

## Module t_bc4.v

module t_bc4 ( t_in, t_capture_clk, t_capture_en,
.data_out(t_data_out), t_data_out );
 input t_in, t_capture_clk, t_capture_en, t_shift_dr, t_si;
 output t_so, t_data_out;

 wire i_data_in;

 IPAD UU4 ( .A(t_in), .Z(i_data_in) );

 DW_bc_4 my_bc4_inst ( .capture_clk(t_capture_clk),
.capture_en(t_capture_en),
 .shift_dr(t_shift_dr), .si(t_si), .data_in(i_data_in),
.so(t_so),
 .data_out(t_data_out) );
endmodule
```

The script in Example 2-4 applies to the netlist for top.v1.

Example 2-4 Example Netlist Script

```
## BSR-in-pads specification
# Input pad t_bc1
define_dft_design -type PAD \
 -design_name t_bc1 \
 -interface {port in1 high \
 data_out t_data_out high \
 si t_si h \
 so t_so h \
 mode_in t_mode low \
 capture_clk t_capture_clk high \
 capture_en t_capture_en low \
 update_clk t_update_clk high \
 update_en t_update_en high \
 shift_dr t_shift_dr h} \
 -params {$pad_type$ string input $bsr_segment$ list \
 string "0 BC1 t_in - X -" $end_list$}

## Input pad t_bc2
define_dft_design -type PAD \
 -design_name t_bc2 \
 -interface {port in2 high \
 data_out t_data_out high \
 si t_si h \
 so t_so h \
 mode_in t_mode low \
 capture_clk t_capture_clk high \
 capture_en t_capture_en low \
 update_clk t_update_clk high \
 update_en t_update_en high \
 shift_dr t_shift_dr h} \
 -params {$pad_type$ string input $bsr_segment$ list \
 string "0 BC2 t_in - X -" $end_list$}

## Input pad t_bc4
define_dft_design -type PAD \
 -design_name t_bc4 \
 -interface {port clk high \
 data_out t_data_out high \
 si t_si h \
 so t_so h \
 capture_clk t_capture_clk high \
 capture_en t_capture_en low \
 shift_dr t_shift_dr h} \
 -params {$pad_type$ string input $bsr_segment$ list \
 string "0 BC4 t_in - X -" $end_list$}
```

Ordering the Boundary-Scan Registers

BSD Compiler automatically sorts the ports alphabetically to place the BSRs. If you require the ports be placed in a specific order, use the port-to-pin map. See "Reading the Port-to-Pin Mapping File" on page 5-10 for more information on how to order your ports with the port-to-pin map. After you specify the order of your ports using the port-to-pin map, you can fine-tune the order of the BSRs using the `set_scan_path` command.

Setting Boundary-Scan Specifications

Boundary-scan specifications define parameters of boundary-scan elements such as TAP signals, types of boundary-scan cells, test data registers, and the boundary-scan instruction set.

Defining IEEE Std 1149.1 Test Access Ports (TAPs)

The `set_dft_signal` command identifies IEEE Std 1149.1 test access ports (TAPs) by placing an attribute on the specified port. The attribute value is the same as the signal type keyword.

Use the `set_dft_signal` command to define each TAP you intend to create in your boundary-scan design.

Run the `set_dft_signal` command using the following options:

```
set_dft_signal -view [spec|existing_dft] -type signal_type \
               -port port_list -hookup_pin pad_output_name
```

The command has the options shown in Table 2-2.

Table 2-2 set_dft_signal Command Options for TAPs

Option	Description
`-type` *signal_type*	Keyword that specifies the type of IEEE Std 1149.1 test signal you are defining. Table 2-3 shows the valid signal type of keywords.
`-port` *port_list*	Name of the design port driving the IEEE Std 1149.1 test signal.
`-hookup_pin` **pad_output_pin**	Identifies the pad output pin to hook up the TAPs in the design.
`-hookup_sense` *inverted* | *non_inverted*	Specifies the sense of the hookup port; tool issues a warning if sense does not match observed. Default is non_inverted.
`-active_state` *active_state*	Active state can be 0 or 1 and specifies the active sense of the port (high or low); for a TRST port the active state is 0, and for TCK, TMS, TDI, and TDO ports, the active state is 1.
`-view` *spec*	The design must change as a result of the specification.
`-view` *existing_dft*	The design is not changed by the specification.

Test signal names defined with the `set_dft_signal` command are listed in Table 2-3.

Table 2-3 Test Signal Names

Signal type keyword	Description
tck	test clock
tdi	test data in
tdo	test data out
tdo_en	test data out enable
tms	test mode select

Table 2-3 Test Signal Names (Continued)

Signal type keyword	Description
trst	asynchronous test reset (optional)

Specifying Hookup Pins for Test Access Ports (TAPs)

All hookup pins are specified by using the `-hookup_pin` option in the `set_dft_signal` command.

The following example shows how to use the `-hookup_pin` option for TAPs:

```
set_dft_signal -view spec -type tdi -port my_tdi \
               -hookup_pin /io/pad_output
set_dft_signal -view spec -type tms -port my_tms \
               -hookup_pin /io/pad_output
set_dft_signal -view spec -type trst -port my_trst \
               -hookup_pin /io/pad_output/E
set_dft_signal -view spec -type tdo -port my_tdo \
               -hookup_pin {/io/pad_output/Z}
set_dft_signal -view spec -type tdo_en -port my_tdo \
               -hookup_pin {/io/pad_output/E}
```

Global Hook Up of Test Access Port FSM State Signals by Pin Name

You can hook up any of the sixteen TAP FSM state signals to a test data register pin or core level pin by using the `set_dft_signal` command and specifying both the signal type with the `-type` option and the pin path with the `-hookup_pin` option. The signal types that you specify with the `set_dft_signal` command and their corresponding TAP FSM signals are listed in Table 2-4.

Table 2-4 The set_dft_signal Command Signal Types for the TAP FSM State Signals

Signal type	TAP FSM signal
bsd_test_logic_reset	TAP FSM TLR
bsd_run_test_idle	TAP FSM RTI
bsd_select_dr_scan	TAP FSM Select-DR
bsd_capture_dr	TAP FSM Capture-DR
bsd_shift_dr	TAP FSM Shift-DR
bsd_exit1_dr	TAP FSM Exit1-DR

Table 2-4 The set_dft_signal Command Signal Types for the TAP FSM State Signals (Continued)

Signal type	TAP FSM signal
bsd_pause_dr	TAP FSM Pause-DR
bsd_exit2_dr	TAP FSM Exit2-DR
bsd_update_dr	TAP FSM Update-DR
bsd_select_ir_scan	TAP FSM Select-IR
bsd_capture_ir	TAP FSM Capture-IR
bsd_shift_ir	TAP FSM Shift-IR
bsd_exit1_ir	TAP FSM Exit1-IR
bsd_pause_ir	TAP FSM Pause-IR
bsd_exit2_ir	TAP FSM Exit2-IR
bsd_update_ir	TAP FSM Update-IR

Note:
You must specify the signal types and pin paths *before* you run the `insert_dft` command.

In the following example, the BSD Compiler TAP FSM state signals are hooked up to the module path pin U1/*pin_name*, using the option arguments of the `set_dft_signal` command:

```
...
set_dft_configuration -bsd enable -scan disable
set_dft_signal -view spec -type  bsd_test_logic_reset  -hookup_pin  u1/reset
set_dft_signal -view spec -type  bsd_run_test_idle     -hookup_pin  u1/run_test_idle
set_dft_signal -view spec -type  bsd_select_dr_scan    -hookup_pin  u1/select_dr_scan
set_dft_signal -view spec -type  bsd_capture_dr        -hookup_pin  u1/capture_dr
set_dft_signal -view spec -type  bsd_shift_dr          -hookup_pin  u1/shift_dr
set_dft_signal -view spec -type  bsd_exit1_dr          -hookup_pin  u1/exit1_dr
set_dft_signal -view spec -type  bsd_pause_dr          -hookup_pin  u1/pause_dr
set_dft_signal -view spec -type  bsd_exit2_dr          -hookup_pin  u1/exit2_dr
set_dft_signal -view spec -type  bsd_update_dr         -hookup_pin  u1/update_dr
set_dft_signal -view spec -type  bsd_select_ir_scan    -hookup_pin  u1/select_ir_scan
set_dft_signal -view spec -type  bsd_capture_ir        -hookup_pin  u1/capture_ir
set_dft_signal -view spec -type  bsd_shift_ir          -hookup_pin  u1/shift_ir
set_dft_signal -view spec -type  bsd_exit1_ir          -hookup_pin  u1/exit1_ir
set_dft_signal -view spec -type  bsd_pause_ir          -hookup_pin  u1/pause_ir
set_dft_signal -view spec -type  bsd_exit2_ir          -hookup_pin  u1/exit2_ir
set_dft_signal -view spec -type  bsd_update_ir         -hookup_pin  u1/update_ir
...
insert_dft
```

Setting Boundary-Scan Test Port Attributes

To use the `set_dft_signal` command, first identify the test signal ports on your design and then assign port signal attributes to those ports.

Assume, for example, that your design has the following IEEE Std 1149.1 test signal ports:

- The `my_tck` port drives the test clock signal.
- The `my_tdi` port drives the test data in signal.
- The `my_tdo` port drives the test data out signal.
- The `my_tms` port drives the test mode select signal.
- The `my_trst` port drives the asynchronous test reset signal.

You would use the following command sequence to identify your IEEE Std 1149.1 test signals:

```
dc_shell> set_dft_signal -view spec -type tck -port my_tck
dc_shell> set_dft_signal -view spec -type tdi -port my_tdi
dc_shell> set_dft_signal -view spec -type tdo -port my_tdo
dc_shell> set_dft_signal -view spec -type tms -port my_tms
dc_shell> set_dft_signal -view spec -type trst -port my_trst
```

Note:
Use the `-view existing` specification with the TAP port signals for a verification only flow.

Reporting Boundary-Scan Test Port Attributes

If you want to verify that the TAP port attributes have been set correctly, you can use the `preview_dft -bsd all` command to identify the IEEE Std 1149.1 test signal ports with IEEE Std 1149.1 test signal attributes.

You can also the `report_dft_signal` command to generate a report of all the TAP DFT signals.

```
dc_shell> preview_dft -bsd all
```

Note:
When TAP port signals are specified in `existing_dft` view, BSD preview/insertion exits with a message that TAP ports are not found. BSD preview/insertion only looks at view `spec` for TAP port signals and `-hookup_pin` is not supported for this view for TAP ports. When `-hookup_pin` is specified for view `spec` for TAP ports, the specification is rejected with an error message.

For more information about `preview_dft`, see "Previewing the Boundary-Scan Design" on page 2-64.

Removing Boundary-Scan Test Port Attributes

If you want to remove TAP port attributes, use the `remove_dft_signal` command, which has the following syntax:

```
remove_dft_signal -port [port1 port2 ...]
```

The command has the option shown in Table 2-5.

Table 2-5 remove_dft_signal Command Option

Option	Description
`-port` *port_list*	Name of the IEEE Std 1149.1 test port signal.

Identifying Linkage Ports

You can avoid putting boundary-scan cells on some ports in your design by using the `set_bsd_linkage_port` command. This command identifies the ports of the current design to be considered as linkage ports. Such ports might be analog, power or ground, or any driven by a black box cell that you do not want to consider for boundary-scan insertion and compliance checking. The linkage ports are listed as linkage bits in the BSDL. The syntax of the command is

```
set_bsd_linkage_port -port_list [list_of_ports]
```

The command has the option shown in Table 2-6.

Table 2-6 set_bsd_linkage_port Command Option

Option	Description
`-port_list` *list_of_ports*	Identifies ports for insert_dft to ignore.

If you want to identify ports that are ignored by BSD, use the following command:

```
dc_shell> set_bsd_linkage_port -port_list {port1 port2 ...}
```

Identifying Clock Signals

To better optimize your design, you should specify clock ports. Clock ports, unlike other boundary-scan ports, should be observe-only ports (such as a BC4 cell). These clock ports control timing for internal logic, such as user-defined test data registers. Board-level testing should have no control over their function.

Use the `create_clock` command to identify all clocks in the design. For more information about the `create_clock` command, see Appendix B, “Setting Timing Attributes,” and the *Design Compiler Reference Manual.*

You can identify clock signals by using the following command:

```
create_clock -period 100 -waveform {0 50} clk
```

Use the `set_dft_signal` command when you want to modify the TCK default period. When setting ports as system or test clock ports, use `set_dft_signal` or `create_clock` commands explicitly.

The syntax for the `set_dft_signal` command is

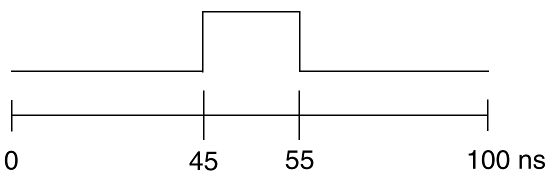
```
set_dft_signal -view [existing_dft | spec]
               -type clock_type2
               -port port_list -timing timing_list
```

Note:
The `set_dft_signal` command has a period and a strobe associated with it. They need to be identical to the `test_default_period` and `test_bsd_default_strobe` values, respectively. If you modify either value, you must adjust the other also. For more information on timing attributes, see Appendix B, “Setting Timing Attributes”.

Multiplexed Flip-Flop Design Example

The clock timing for the multiplexed flip-flop design example shown in Figure 2-12 is a positive pulse clock with a period of 100.0 ns. The rising edge occurs at 45.0 ns and the falling edge occurs at 55.0 ns. This clock waveform is shown in Figure 2-12.

Figure 2-12 Default Clock Timing for Multiplexed Flip-Flop Example



The `check_bsd` command automatically infers this clock timing because the design contains an edge-triggered, active rising clock. You can explicitly specify this clock waveform using the following `set_dft_signal` command:

```
dc_shell> set_dft_signal -view existing_dft -type TCK
              -port my_tck -timing {45.0 55.0}
```

If a return-to-one clock is required instead of the default clock, you can use the following `set_dft_signal` command:

```
dc_shell> set_dft_signal -view existing_dft -type TCK
              -port my_tck -timing {55.0 45.0}
```

For more information on setting timing attributes, see Appendix B, "Setting Timing Attributes."

Selecting the Boundary-Scan Configuration

You can define the boundary-scan instruction encoding, the reset configuration of your TAP, and the pin-mapped package you intend to use with the `set_bsd_configuration` command.

The syntax of the command is

```
set_bsd_configuration
          [-asynchronous_reset true | false]
          [-check_pad_designs none | all | pad_design_list]
          [-control_cell_max_fanout max_fanout]
          [-default_package package_name]
          [-instruction_encoding binary | one_hot]
          [-ir_width instruction_register_width]
          [-std ieee1149.1_1993 | ieee1149.1_2001 |ieee1149.6_2003]
          [-style synchronous | asynchronous]
```

The options for `set_bsd_configuration` are shown in Table 2-7.

Table 2-7 set_bsd_configuration Command Options

Option	Description	Choices
`-asynchronous_reset`	Sets the type of reset implemented in the TAP controller	The value true (default) puts an asynchronous reset in the TAP controller. The value false puts no reset in the TAP controller; you must provide a power-up mechanism.
`-check_pad_designs`	Specifies the kind of validation of the soft macro pads.	The value none causes all the pads in the pad_design_list to be bypassed during preview_dft validation. Validation of all pads is still done in check_bsd. The value all (default) causes all pads in the pad_design_list to be validated including open_source and open_drain pad types.
`-control_cell_max_fanout`	Specifies the maximum number of fanouts allowed for the control cell. To overwrite this behavior use the `set_boundary_cell` command.	
`-default_package`	Sets the default package to be used.	package_name is the name of the package you provide.
`-instruction_encoding`	Sets the implementation for the instruction register.	The value binary causes the instruction register implementation to be binary. The value one_hot causes the instruction register implementation to be one_hot.

Table 2-7 set_bsd_configuration Command Options (Continued)

Option	Description	Choices
`-ir_width`	Specifies the width of the instruction register for the boundary-scan design.	2 < ir_bit_width < 32 ir_bit_width must be a positive integer representing the number of bits in the instruction register.
`-std`	Specifies the IEEE Std 1149.X for the BSD Compiler design flow.	By default BSD Compiler runs in the IEEE1149.1-2001 standard mode. When set to -std ieee1149.6_2003, BSD Compiler runs in the IEEE1149.1-2001 and the IEEE1149.6-2003 standard modes. When set to -std ieee1149.1_1993 ieee1149.6_2003, BSD Compiler runs in the IEEE1149.1-1993 and the IEEE1149.6-2003 standard modes.
`-style`	Sets the TAP controller TCK routing implementation.	The value synchronous (default) causes synchronous TAP controller signals, sync_capture_en, sync_update_dr, and TCK to be connected to the boundary-scan register cells. The value asynchronous causes asynchronous TAP controller signals, clock_dr and update_dr to be connected to the boundary-scan register cells.

Note:
The default package must be the package name defined in the port-to-pin mapping file you use. Use the `-default_package` option to define the default package when you are reading multiple packages into the design.

To select the boundary-scan configuration for the TAP controller in your design, you can use the following command:

```
set_bsd_configuration -style synchronous \
     -instruction_encoding binary -ir_width 4 \
     -asynchronous_reset true -default_package my_package
```

Selecting the Boundary-Scan Location

By default, when you use the `set_dft_location` command, the TAP controller design module and the BSR design module are inserted at the top-level design. However, you can use the `-include` option of this command to specify any location within the design hierarchy where you want the TAP or BSR module inserted. With respect to these modules the command syntax is

```
set_dft_location -include [TAP|BSR] <specified_design_hierarchy_path>
```

If you use this command and do not specify a path, the TAP or BSR is inserted at the top level.

As an example, to specify that the BSR is to be inserted at the existing I/O design module level, use the command

```
set_dft_location -include BSR top/io
```

Similarly, as an example, to specify that the TAP is to be inserted at the existing core design module level, use the command

```
set_dft_location -include TAP top/core
```

Note:
To review all the `set_dft_location` command options, see the man page.

Selecting the TAP Controller Reset Configuration

The TAP controller can be reset (initialized), based on the IEEE 1149.1 Std, in one of several ways:

- Initializing the TAP with asynchronous reset using TRST
- Initializing the TAP with asynchronous reset using a power-up-reset (PUR) cell
- Initializing the TAP with asynchronous reset using a PUR cell and TRST
- Initializing the TAP with synchronous reset holding TMS to logic1 and at least 5 TCK clock cycles

Initializing the TAP With Asynchronous Reset Using TRST

To initialize your TAP controller to reset asynchronously, select the TAP port TRST* when setting your BSD specifications. Then select asynchronous as the type of reset to be implemented.

To put the TAP controller into the Test-Logic-Reset state, use the following commands:

```
...
set_dft_signal -view spec -type trst -port my_trst
...
set_bsd_configuration -asynchronous_reset true
...
```

Implementing this command causes the BSD Compiler to add the TRST port and to implement the asynchronous reset for the TAP controller.

Initializing the TAP With Asynchronous Reset Using a PUR Cell

To initialize your TAP controller to reset using power-up reset, omit the TAP port TRST* and use the PUR cell as part of your BSD specification. Note that if you do not have a TRST* pin in your design, you must use the PUR cell.

The `set_bsd_power_up_reset` command specifies and characterizes the power-up reset cell for the current design. To specify the details of the power-up cell to implement the power-up reset, use the following command:

```
dc_shell> set_bsd_power_up_reset -cell_name cell_name \
               -reset_pin_name pin_name -active_state [high | low] \
               -delay power_up_reset_delay
```

The options for `set_bsd_power_up_reset` are shown in Table 2-8.

Table 2-8 set_bsd_power_up_reset Command Options

Option	Description	Choices
`-cell_name cell_name`	Specifies instance name of power-up reset cell.	
`-reset_pin_name pin_name`	Specifies the pin name of the power-up-reset cell at which a reset pulse is generated upon power on.	
`-active_state`	Specifies whether the active state of the power-up-reset pulse is high or low.	`high` `low`
`-delay power_up_reset_delay`	Specifies the initial power-up-reset delay, which is measured from when the time power is switched on to the time the TAP controller resets to the test-logic-reset state.	Specify an integer.

If you want to specify a PUR cell with a delay of 130 ns to the power-up reset of your TAP controller, use the following command:

```
set_bsd_power_up_reset -cell_name POR_INST \
     -reset_pin_name reset -active_state high -delay 130
```

BSD Compiler adds an inverter during synthesis when the PUR cell is set to `-active_state high`. For a verification-only flow, confirm that the TAP controller `trst_n` pin gets pulsed with active low upon PUR reset.

When using the `set_bsd_power_up_reset` command to reset the TAP controller, the BSD patterns wait for the amount of time specified by the `-delay` option before starting the JTAG simulation. For example, suppose the power-up reset is defined with a 2000 ns delay:

```
set_bsd_power_up_reset -cell_name my_por_inst \
     -reset_pin_name pwruprst -active_state high -delay 2000
```

Then, with a 100 ns TCK clock period, the test patterns wait 20 TCK cycles before starting the simulation. This behavior is captured in the BSDL file as well as in the BSD STIL patterns and the Verilog testbench.

For the STIL patterns, this delay is accomplished by adding a Loop statement just after the first initialization pattern and before pattern 0, as shown in the following example:

```
Pattern "_BSD_block_" {
    W "BSD_WFT";
    C {
        "all_inputs" = \r8 0;
        "all_outputs" = XXX;
        "all_bidirectionals" = Z;
    }

    Loop 20 {  V {
        "all_inputs" = NNNNNNNN;
        "all_outputs" = XXX;
        "all_bidirectionals" = X;
    }
  }

    "pattern 0 Sync Reset Vectors" : V {
        "all_inputs" = NNNNPN1N;
        "all_outputs" = XXT;
        "all_bidirectionals" = X;
    }
```

With this Loop statement added, the simulation waits the required 20 TCK cycles (2000 ns) before continuing the simulation.

Similarly, a waiting time is added to the Verilog testbench as follows:

```
initial begin
_failed = 0;

/* Initialization vectors. */
_bi=1'bZ;
assign _ck=1'b0;
assign _pi=7'bXXXXXXX;

 #2000 ;
assign _e_po=3'bXXX;
assign _m_po=3'b111;
-> _check_po;
```

You can model the PUR reset pulse in the simulation library for a design without a TRST port by using the power-up reset cell PUR_CELL (instance POR_INST) with an active high RESET pin that is pulsed 150 ns after a delay of 850 ns, as shown in Example 2-5.

Example 2-5 PUR Simulation Model

```
module pur_cell (VDD, Z)
      input VDD;
      output Z;
      reg Z;
      initial begin
            assign Z = 1’b0;
            #850 assign Z = 1’b1;
            #150 assign Z = 1’b0;
      end
endmodule
```

The corresponding BSD Compiler commands are as follows:

```
...
set_bsd_configuration -asynchronous_reset false
...
set_bsd_power_up_reset -cell_name POR_INST \
     -reset_pin_name Z -active_state high -delay 1000
```

You can specify a black box cell for your PUR cell if the black box has the output reset pin defined correctly. Although BSD Compiler does not use the function from the synthesis library for the synchronous reset, the simulation library should describe the PUR reset function correctly, as shown by the preceding commands. See Step 6, “Simulating BSD Patterns With VCS” on page 5-30.

Because the power-up-reset behavior cannot be simulated in TetraMAX, the PUR model relies on virtually disabling the power-up-reset and using the synchronous reset protocol to initially reset the TAP controller.

For TetraMAX, the following PUR model and methodology is recommended for a BSD Compiler inserted design:

- Use a different cell, PUR_tmax, as the power-up-reset cell in the TetraMAX flow. The PUR_tmax cell has a simulation model in which the power-up-reset pin is tied permanently to an inactive value, as shown in the following TetraMAX simulation model:

  ```
  module PUR_tmax(VDD, RESET);
  output RESET;
  input VDD;
  assign RESET = 1'b0;
  endmodule // PUR_tmax
  ```

  This change of the PUR reference cell can be done automatically within the synthesis script by using the `change_link` command, as shown in the following example:

  ```
  ...
  set_dont_touch find(cell, "POR_INST")
  insert_dft
  write -f verilog -h -o DESIGN_bsd.v
  change_link POR_INST pur/PUR_tmax
  write -f verilog -h -o DESIGN_tmax.v
  check_bsd -verbose
  create_bsd_patterns
  ...
  ```

- Use the following TetraMAX command to specify a valid random state (0 or 1) on all state elements:

  ```
  set drc -initialize_dff_dlat random
  ```

  Note that this command and option applies to the TAP controller state elements as well.

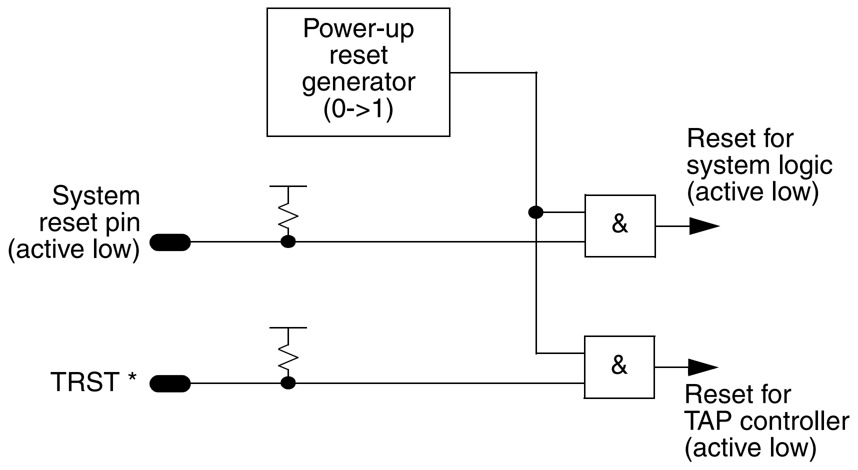
- The synchronous reset sequence generated at the beginning of the external vectors resets the TAP controller after the controller has been brought to a valid random state by the preceding command.

Initializing the TAP With Asynchronous Reset Using a PUR Cell and TRST

The IEEE Std 1149.1 allows the use of both your TRST and PUR cell to reset the TAP controller; the TRST cannot be used for system reset at any time. When using the TRST pin and the PUR cell, you must specify both commands as described previously in the chapter.

Figure 2-13 shows how the TAP initialization can be implemented.

Figure 2-13 Use of Power-Up Reset for System and Test Logic


Power-up reset generator (0->1)
System reset pin (active low)
TRST *
&
&
Reset for system logic (active low)
Reset for TAP controller (active low)

Initializing the TAP With Synchronous Reset Holding TMS to Logic 1 and at Least 5 TCK Clock Cycles

The IEEE Std 1149.1 allows the use of TMS and TCK to initialize the TAP FSM, but this methodology has a risk involved since the TAP FSM weakens the power-up-reset at an unknown state while TCK is clocking.

If BSD Compiler does not find a TRST pin nor a PUR specification, it continues to initialize the TAP FSM using TMS and TCK. This is done by holding TMS high for at least 5 TCK clock cycles moving the TAP FSM to TLR state. No TRST pin or PUR cell specifications are required for this.

Configuring the Device Identification Register

You must define the device identification register (DEVICE_ID) to be used in your boundary-scan design if you intend to use the optional IDCODE instruction. See “Implementing the IEEE Std 1149.1 Instructions” on page 2-50.

The IEEE Std 1149.1 specifies only one device ID register for the TAP controller. For more information about the device identification register, including its structure, see the IEEE Std 1149.1 document.

BSD Compiler allows you to define the identification code in conformance with IEEE Std 1149.1. The code is composed of three configurable fields:

1. The version, which is 4 bits wide
2. The part number, which is 16 bits wide
3. The manufacturer identity, which is 11 bits wide

The device identification register's least significant bit is always set to 1 by BSD Compiler and is not configurable.

You can specify the device ID code by using the `set_bsd_instruction` command, as follows:

```
set_bsd_instruction IDCODE \
     -register DEVICE_ID \
     -capture_value value \
     -code opcode
```

Use the `-capture_value` option to define the binary code that represents the device identification register. This code contains 32 digits, ordered in the way previously described.

Ensure that the least significant bit (LSB) of the capture value is set to 1; if you do not, you see the following error:

```
Error: Invalid LSB for IDCODE capture value. (UIT-545)
Discarded bsd instruction specification.
0
```

When you get this error, change the last bit from 0 to 1.

Example 2-6 Defining the IDCODE Instruction

```
set_bsd_instruction IDCODE -code 0011 -register DEVICE_ID \
        -capture_value {32'b00000000000000000000000000000011}
```

The device identification register is not implemented until you specify the IDCODE instruction. For more about specifying instruction registers, see "Implementing the IEEE Std 1149.1 Instructions" on page 2-50.

USERCODE and Flexible IDCODE Support

When implementing the flexible IDCODE and USERCODE optional instructions with BSD Compiler, the signals holding the capture values for the DIR IDCODE and USERCODE are available at the interface of DW TAP component inserted in your design. You can modify these capture values manually for additional revisions of the IDCODE capture value or the USERCODE capture value. The flexible IDCODE and USERCODE allows you to revise the capture value of the IDCODE and USERCODE by replacing only one mask plate of the JEDEC source rather that of the full mask set.

USERCODE is an IEEE Std 1149.1 optional instruction that allows you to load and shift a user-programmable identification code out of the device identification register (DIR) for examination. You can program the 32-bit binary USERCODE value in the field or choose it during component design.

Example 2-7 Defining the USERCODE Instruction

```
set_bsd_instruction USERCODE -code {1110} -capture_value 32'hfabd0001
```

To instruct BSD Compiler to use the USERCODE instruction when inserting boundary scan, use the `-capture_value` option with the `set_bsd_instruction` command. You can specify a 32-bit binary or hexadecimal value, design pins, or a mix of bits and design pins.

IDCODE is an IEEE Std 1149.1 optional instruction. BSD Compiler supports a flexible IDCODE implementation, which allows you to implement IDCODE with a flexible opcode for the instruction register (IR) and revise the identification code parameters after synthesis.

Example 2-8 Defining the Flexible IDCODE Instruction

```
set_bsd_instruction USERCODE -code {1110} \
               -capture_value {4'h1, 16'b1111111110000000, 12'h2ab} \
               -register DEVICE_ID
```

For example, specify binary or hexadecimal-bit patterns using the following sized-constant Verilog syntax:

number of bits in the binary representation [b|h]’value

Specify a bus representing a collection of pins using the following syntax:

name of the bus [upper bit : lower bit]

The DIR identification code bits are listed in the boundary-scan inserted netlist for post-processing of the DIR parameters.

You can include any valid and unreserved opcode for the IDCODE instruction with the DW_tap_uc TAP controller. Use the opcode 0001 for the DW_tap IDCODE instruction.

See the section “Implementing USERCODE and Flexible IDCODE” on page 2-41 for a example run script that implements USERCODE and flexible IDCODE within BSD Compiler. See Appendix A, “Custom Boundary-Scan Design,” for information on how to use a custom TAP controller with USERCODE and flexible IDCODE instructions.

Custom TAP Interface to Support USERCODE and Flexible IDCODE

To use a TAP controller with USERCODE and flexible IDCODE instructions, you need to set the TAP controller access interface to the correct signal types. Setting the TAP element characterizes a design cell as a boundary-scan TAP controller for use during boundary-scan insertion.

Use the command `define_dft_design -interface` with the appropriate signal types. The following signal types support USERCODE and flexible IDCODE:

```
device_id_sel    : scalar
user_code_sel    : scalar
user_code_val    : 32-bit bus
ver              : 4-bit bus
ver_sel          : scalar
part_num         : 16-bit bus
part_num_sel     : scalar
mnfr_id          : 11-bit bus
mnfr_id_sel      : scalar
```

The `instructions` signal type is mandatory for the access interface, while `sample` and `extest` are optional.

The following example inserts a custom TAP controller with USERCODE and flexible IDCODE instructions:

```
define_dft_design -type TAP_UC
-interface {tck tck h trst_n h trst_n h \
      tms tms h tdi tdi h so so h  \
      bypass_sel bypass_sel h  \
      shift_dr shift_dr h tdo tdo h \
      tdo_en tdo_en h \
      instructions instructions h \
      sync_capture en_sync h capture_en  \
      sync_update h dr_sync update_dr h \
      device_id _sel h device_id_sel h  \
      user_code_sel user_code_sel h  \
      user_doce_val user_code_val h}
-design_name my_tap
set_bsd_instruction {IDCODE} -code {1001} \
      -capture_value {32'b10000001000111010111000011110111}
set_bsd_instruction {USERCODE} -code {1100} -reg DEVICE_ID \
      -capture_value {32'b00011100111110000000010101010010}
```

Implementing USERCODE and Flexible IDCODE

BSD Compiler supports Scan Register and Shift Register as UTDR; before implementing a user-defined instruction, you must define the test data register (UTDR) to which it is to be associated by using the `set_dft_signal` and `set_scan_path` commands, as follows:

The `insert_dft` command implements the logic for the USERCODE instruction as follows:

- Connect the specified capture value bits or driver pins to the TAP controller `user_code_val` signal.
- Connect the decoded USERCODE instruction to the `user_code_sel` signal.
- Connect the decoded IDCODE ORed with the USERCODE instruction to the `device_id_sel` signal.

The `insert_dft` command implements any specified opcode for the IDCODE instruction as well as the specified capture values through the following TAP access signals:

```
device_id_sel
ver
ver_sel
part_num
part_num_sel
mnfr_id
mnfr_id_sel
```

Implementing User-Defined Test Data Registers

Before implementing a user-defined instruction, you must define the test data register (UTDR) to which it is to be associated using the `set_dft_signal` and `set_scan_path` commands, as follows:

```
set_dft_signal -view spec -type signal_type \
               -hookup_pin {reg_cell_name/pin_name ...}

set_scan_path register_name -view spec
               -class bsd -hookup {reg_cell_name/pin_name ...} \
               -exact_length 10 \
               -bsd_style {synchronous, asynchronous, global}
```

First define the signals with the `set_dft_signal` command, and then declare the register associated with those signals with the `set_scan_path` command. Use a unique register name. Specifying the `-exact_length` option for the `set_scan_path` command is mandatory.

This register definition only creates the interface. The interface created by this command causes the boundary-scan design to connect the serial input and the serial output of the register between TDI and TDO. The connections from the boundary-scan logic to the design cell are made according to the access interface you specify.

Specify the full hierarchical pin names with both commands when specifying the registers. In addition, specify register access pin types as DFT signal types.

The following pin types listed in Table 2-9 are supported.

Table 2-9 Supported DFT Signal Types

Signal type	Description
`tdi`	Specifies TDI port or BSD register TDI access pin.
`tdo`	Specifies TDO port or BSD register TDO access pin.
`bsd_shift_en`	Specifies the register access pin to be hooked up to the TAP shift_dr pin when the instruction to select the register is active.
`bsd_capture_en`	Specifies the register access pin to be hooked up to the TAP sync_capture_en pin when the instruction that selects the register is active. This signal is also used in core integration.
`bsd_capture_dr`	Specifies the register access pin to be hooked up to the TAP Capture-DR state on the negative edge of TCK, when the instruction that selects the register is active.
`bsd_update_en`	Specifies the register access pin to be hooked up to the TAP sync_update_dr pin when the instruction that selects the register is active. This signal is also used in core integration.
`bsd_update_dr`	Specifies the register access pin to be hooked up to the TAP Update-DR state on negative edge of TCK when the instruction that selects the register is active.
`capture_clk`	Specifies the register access pin to be hooked up to the TCK/clock_dr pin when the instruction that selects the pin is active.
`update_clk`	Specifies the register access pin to be hooked up to the TCK/update_dr pin when the instruction that selects the register is active.
`bsd_reset`	Specifies the register access pin to be hooked up directly to the TAP Test-Logic-Reset state. When the bsd_reset signal is part of the UTDR `set_scan_path` hookup specification, it is gated by the UTDR instruction.
`inst_enable`	Specifies the register access pin to be held active when the instruction that selects the register is active.
`bist_enable`	Specifies the register access pin to be hooked to TCK when the instruction that selects the register is active and TAP is in Run-Test-Idle (RTI) state. This signal is also used in core integration.

Note:

Inversion of these signals is supported by using the `-active_state 0` option, as shown in Example 2-9.

The `set_scan_path` command requires the option `-exact_length number_of_reg_elements`.

DFT Signal Polarity: Unspecified or High

The signal remains active high while the instruction is active and TAP is in <state>. The signal remains low while the instruction is not active.

DFT Signal Polarity: Low

The signal remains active low while the instruction is active and TAP is in <state>.

When the instruction is not active, the signal is held at high if the pin is not driven functionally or tied to 0 before BSD insertion, which is otherwise driven from the functional driver.

Example 2-9 Supporting Signal Inversion by Specifying the Option -active_state 0

```
set_dft_signal -view spec -type bsd_capture_en \
    -hookup_pin UTDR/U_apg_0/cap_en_n \
    -active_state 0
```

The cell you specify can be empty or unmapped. Its content is not checked before boundary-scan insertion. After the design is mapped, you can check the complete function with the `check_bsd` command.

The minimum interface required for a user-defined register is

- tdi
- tdo
- capture_clk
- bsd_shift_en

To specify a user test data register, use commands similar to those in Example 2-10.

Example 2-10 User Test Data Register

```
set_dft_signal -view spec -type tdi -hookup_pin BIST/WRAPPER_0/debug_in
set_dft_signal -view spec -type tdo -hookup_pin BIST/WRAPPER_0/debug_out
set_dft_signal -view spec -type bsd_shift_en -hookup_pin \
     BIST/WRAPPER_0/debug_en
set_dft_signal -view spec -type capture_clk  -hookup_pin \
     BIST/WRAPPER_0/clk

set_dft_signal -view spec \
           -type bsd_reset \
           -hookup_pin BIST/WRAPPER_0/debug_reset \
           -active_state 0
set_scan_path DEBUG_reg -class bsd \
                        -view spec \
                        -hookup { BIST/WRAPPER_0/debug_in \
                                  BIST/WRAPPER_0/debug_out \
                                  BIST/WRAPPER_0/debug_en \
                                  BIST/WRAPPER_0/clk} \
                        -exact_length 10
set_bsd_instruction DEBUG -code 1010 \
                          -register DEBUG_reg \
                          -input_clock_condition TCK \
                          -output_condition BSR
```

Connecting the TAP Reset to a UTDR Reset

To connect the TAP controller `bsd_reset` signal to the UTDR reset signal, use the `-type bsd_reset` option in the `set_dft_signal` command and exclude the `bsd_reset` signal from the hookup specification of the UTDR in the `set_scan_path` command. The tool then routes the `bsd_signal` directly to the UTDR reset signal.

Example 2-11 Connecting the TAP Controller bsd_reset Signal to the UTDR Reset Signal

```
set_dft_signal -view exist -type bsd_reset \
     -hookup_pin core_utdr_reg/utdr_rst_enable \
     -active_state 0
set_dft_signal ... # all remaining UTDR signals are controlled by the TAP

# Exclude bsd_reset signal from the set_scan_path command hookup
# spec of the UTDR

set_scan_path -class bsd <utdr_reg_name> -hookup {HPINS}
#NOTE: HPINS should not contain bsd_reset signals
```

It the UTDR `bsd_reset` signal is included in the `set_scan_path` hookup specifications of the UTDR, the UTDR reset signal is gated by the UTDR instruction, which is a less practical approach.

Implementing Instructions

BSD Compiler implements the standard set of IEEE Std 1149.1 instructions, and it allows you to implement user-defined instructions as well. You specify boundary-scan instructions by using the `set_bsd_instruction` command.

The use of this command and its options for implementing the standard set of instructions is described in the section "Implementing the IEEE Std 1149.1 Instructions" on page 2-50. The use of this command and its options for implementing user-defined instructions is described in the section "Implementing User-Defined Instructions" on page 2-52.

Note:

> Before you implement a user-defined instruction, you must first define the register to which you want it associated using the `set_scan_path` command. For more information, see "Implementing User-Defined Test Data Registers" on page 2-42.

Each time you specify an instruction using the `set_bsd_instruction` command with the same code op-code, the previous instruction is overwritten with the latest specification.

The syntax of the command is:

```
set_bsd_instruction instruction_list
     [-view spec | existing_dft]
     [-code inst_code_list]
     [-register register_name]
     [-input_clock_condition clock_conditioning]
     [-output_condition BSR | HIGHZ | NONE]
     [-capture_value capture_value_list]
     [-private]
     [-clock_cycles clock_cycle_list]
     [-signature signature]
     [-high pin_list_name]
     [-low pin_list_name]
     [-internal_scan pin_name]
     [-time real_time]
     [-excluded_bsr_condition CLAMP | NONE]
```

Note:

> `-high`, `-low`, and `-internal_scan` are options to connect signals controlled by the TAP controller.

Table 2-10 shows the `set_bsd_instruction` command options.

Table 2-10 The set_bsd_instruction Command Options

Option	Description	Choices
`-view`	Specifies the view to which the specification applies. Valid views are existing_dft and spec. Setting the value to existing_dft indicates that the specification refers to the existing instructions in the design. This is used when working with designs where BSD logic is inserted. Setting the value to spec, the default, indicates that the specification refers to instructions to be designed by insert_dft. This is used when inserting BSD logic.	spec existing_dft
`-code` *`inst_code_list`*	Specifies the list of binary codes corresponding to the instructions specified with instruction_list. This argument is required when setting user-defined instructions. Note that an opcode must be associated with a single instruction. That means instruction_list must contain a single identifier if the -code inst_code_list is used. The opcode must have a length equal to the number set by set_bsd_configuration -ir_width.	
`-register`	Selects a standard data register or a user-defined register, previously declared with the set_bsd_signal or set_scan_path command to be connected for serial access between TDI and TDO when the instruction is active.	BOUNDARY BYPASS *user_defined*
`-input_clock_condition`	Specifies the value that drives the clock signal going into the system when the instruction is active. If clocking the TDR with bist_clk use PI, and for TCK use capture_clk.	PI (default) TCK
`-output_condition`	Specifies how the system outputs are driven when a given instruction (BSR, HIGHZ, or NONE) is active.	
	Puts the boundary scan register into EXTEST mode.	BSR
	Puts the output pins into high impedance mode.	HIGHZ
	Puts the boundary scan register into transparent mode.	NONE (default)
`-capture_value` *`capture_value_list`*	Defines the capture value of the DEVICE-ID register for the IDCODE instruction. This capture value contains 32 digits. Ensure that the least significant bit (LSB) of the capture value is set to 1 for the IDCODE capture value.	

Table 2-10 The set_bsd_instruction Command Options (Continued)

Option	Description	Choices
`-private`	Specifies that the instructions are considered private. Private instructions have the INSTRUCTION_PRIVATE attribute in the BSDL file generated by the tool. The data registers of private instructions are not shown in the BSDL file.	
`-clock_cycles` *`clock_cycle_list`*	List of clock port and integer pairs that specify the minimum number of clock cycles on the specified clock port required for the design to stay in the Run-Test/Idle TAP controller state to ensure completion of the INTEST or the RUNBIST instruction.	
`-signature` *`pattern`*	Specifies the state of the boundary-scan register after execution of the RUNBIST instruction. The pattern variable correlates to the <det pattern> argument defined in section B.8.15 of the *Supplement to IEEE Std 1149.1*.	
`-high`	Specifies a list of pins that need to be in active high state when the specified instructions are selected. The pin_name_list should contain the hierarchical names of the pins.	
`-low`	Specifies a list of pins that need to be in active low state when the specified instructions are selected. The pin_name_list should contain the hierarchical names of the pins.	
`-internal_scan`	Specifies a hierarchical pin in the design to which the TAP controller's Shift-dr signal gated with the decoded instruction is connected.	
`-time real_time`	A real number that specifies the time duration in nanoseconds for the EXTEST_TRAIN and EXTEST_PULSE instructions in the Run-Test-Idle (RTI) state, as well as for INTEST and RUNBIST. The tool returns an error if this option is used with other instructions	

Table 2-10 The set_bsd_instruction Command Options (Continued)

Option	Description	Choices
`-excluded_bsr_condition`	Specify conditioning of BSR cells excluded from the specified short BSR chain. Note: The error, Invalid value%s specified for option -excluded_bsr_condition., occurs when a value other than CLAMP or NONE is used with this option.	CLAMP - all BSR cells that are not part of the specified BSR chain are conditioned as if CLAMP instruction is active. NONE - all BSR cells that are not part of the specified BSR chain are kept transparent while the user instruction is active.

BSD synthesis handles Active High and Active Low instruction Enable pins. The `set_bsd_instruction` command supports the following options.

```
-high list_of active_high_instruction_enable_pins
-low list_of active_low_instruction_enable_pins
```

Example 2-12 The set_bsd_instruction Command for Active High Instruction Pin

```
set_bsd_instruction MY_INSTR -code 1010 -register MY_BSR_REG \
     -high pad1/tm
```

Implementing User Instructions With HIGHZ Behavior

If a design has an Output2 port as well as an Output3 port, you can only specify a user-defined instruction for the Output3 port to behave as the HIGHZ instruction. This port is in HIGHZ when the instruction is active. The HIGHZ instruction cannot be specified for Output2 ports according to the IEEE Std 1149.1 rule.

Note:
Here, Output2 signifies an output buffer and Output3 signifies a tristate output buffer.

See the following example:

```
set_bsd_instruction UDI_HIGHZ -code 0100 -register BYPASS \
-output_condition HIGHZ
```

This instruction implements a user-defined instruction `UDI_HIGHZ` with the same behavior as the HIGHZ instruction to the I/O ring.

Implementing the IEEE Std 1149.1 Instructions

BSD Compiler allows you to implement the mandatory and optional set of IEEE Std 1149.1 instructions as well as RUNBIST, INTEST, and user-defined instructions.

The following instructions, which are mandated by IEEE Std 1149.1, are generated automatically:

- BYPASS
- EXTEST
- SAMPLE
- PRELOAD

You need not do anything to implement these instructions in their default form. If you want to assign multiple registers to address any of these instructions, see the section "Implementing User-Defined Test Data Registers" on page 2-42. To see the binary code assignments for the default instructions, see the information about binary code assignments in the *BSD Compiler Reference Manual*.

If you want to implement the following optional instructions, you must specify:

- IDCODE
- USERCODE
- HIGHZ
- CLAMP
- INTEST
- RUNBIST
- user-defined-instructions

For example, the following command implements the HIGHZ and CLAMP instructions:

```
dc_shell> set_bsd_instruction -view spec {HIGHZ CLAMP}
```

See section "USERCODE and Flexible IDCODE Support" on page 2-39 for further information about using the IDCODE and USERCODE instructions.

Implementing the INTEST Instruction

You can set the INTEST instruction by using the `set_bsd_instruction` command to specify the instruction name and the output clock conditioning. The syntax of the command for implementing the INTEST instruction is shown here:

```
set_bsd_instruction INTEST -view spec
     [-clock_cycles clock_cycle_list]
     [-input_clock_condition input_clock_conditioning]
     [-output_condition output_conditioning]
     [-time real_time]
```

The options you use to set the INTEST instruction are shown in Table 2-10 on page 2-47.

Note:
INTEST instructions using BC4 cells on input ports and BC2 cells on output ports are not supported by the IEEE Std 1149.1.

In the following example, the INTEST instruction specifies 5000 ns wait time at the RTI state with PI driving the input and HIGHZ driving the output.

```
set_bsd_instruction INTEST -view spec \
     [-time 5000] \
     [-input_clock_condition PI -output_condition HIGHZ]
```

Note:
You must select either the `-time` option for real-time specifications or the `-clock_cycles` option for clock-cycles specification but not both.

For additional information on the INTEST instruction, see the *BSD Compiler Reference Manual*.

Implementing the RUNBIST Instruction

You can set the RUNBIST instruction by using the `set_bsd_instruction` command to specify the instruction name, the input clock conditioning, and the output conditioning.

The syntax of the command for implementing the RUNBIST instruction is shown here:

```
set_bsd_instruction RUNBIST -view spec
     [-clock_cycles clock_cycle_list]
     [-input_clock_condition clock_conditioning]
     [-output_condition output_conditioning]
     [-signature pattern]
     [-time real_time]
```

The options you use to set the RUNBIST instruction are shown in Table 2-10 on page 2-47.

In the following example, the RUNBIST instruction specifies 5000 ns wait time at the Test-Idle state with HIGHZ as the output condition.

```
set_bsd_instruction RUNBIST -time 5000\
     -output_condition HIGHZ\
     -signature 10101010\
```

Note:

You must select either the `-time` option for real-time specifications or the `-clock_cycles` option for clock-cycles specification but not both.

Also, you must set your RUNBIST_reg test data register before specifying the RUNBIST instruction. For more information on setting test data registers, see "Implementing User-Defined Test Data Registers" on page 2-42. If, when using RUNBIST_reg, you get the TEST-437 error message, you must validate it to make sure it shifts properly. You can test shifting using the `dft_drc` command in DFT Compiler. See DFT Compiler documentation for information on validating shift.

Implementing User-Defined Instructions

You can further customize your boundary-scan implementation by creating user-defined instructions. You can use BSD Compiler to address individual instructions with multiple opcodes. With BSD Compiler, you can implement instructions that select user-defined test data registers. These user-defined test data registers can function as a test access mechanism to structure internal core logic.

When you want to implement a user-defined instruction, use the `set_bsd_instruction` command to identify the instruction name, as well as the register and the opcode with which it is to be associated. You must declare the user data registers referenced by the user instruction declarations.

Setting User-Defined Instructions

After you define registers with the `set_dft_signal` and `set_scan_path` commands, use the `set_bsd_instruction` command to implement user-defined instructions. These user-defined instructions select their corresponding user-defined register to be put between TDI and TDO.

The syntax for the `set_bsd_instruction` command when you are setting user-defined instructions is

```
set_bsd_instruction -view spec instruction_list
     [-code inst_code_list]
     [-register register_name]
     [-input_clock_condition clock_conditioning]
     [-output_condition output_conditioning]
     [-internal_scan pin_name]
     [-high pin_list_name]
     [-low pin_list_name]
```

The options you use to create user-defined instructions with the `set_bsd_instruction` command are shown in Table 2-10 on page 2-47.

In the following example, `set_bsd_instruction` implements a user-defined instruction named UDI1, whose binary code is 1010. The command selects the boundary-scan register to be connected for serial access TDI and TDO with high impedance.

Note:
When using the `set_bsd_instruction` command to select a user-defined instruction, you must specify your particular register. BSD Compiler defaults to BYPASS if you selected BYPASS, CLAMP, and HIGHZ instructions; however, there is no default register for user-defined instructions. If you forget to specify a register, BSD Compiler issues an error and rejects the specification.

Example 2-13 set_bsd_instruction Command

```
dc_shell> set_bsd_instruction -view spec {UDI1} \
            -register BOUNDARY -code {1010} \
            -output_condition HIGHZ
```

Setting Private Instructions

Private instructions are user-defined instructions that are restricted from general use. These instructions might be unsafe for use by other than the manufacturer, and their effects are undefined to the general user.

For private instructions, the opcodes for the instructions are shown in the BSDL file, and the following attribute appears:

```
attribute INSTRUCTION_PRIVATE of M1: entity is
  MY_PRIVATE_INSTR1,MY_PRIVATE_INSTR2;
```

The Test Data Registers for private instructions do not appear in the BSDL file. By default, no patterns are generated.

To generate patterns for private instructions, set the `test_bsdvec_create_patterns_for_private_instructions` variable to true.

Example 2-14 shows how to specify a private instruction and generate BSDL and patterns for private instructions.

Example 2-14 Generating BSDL and Patterns for Private Instructions

```
...
set_bsd_instruction -private {PDI_2} -code {0010} -register BOUNDARY
preview_dft -bsd all
insert_dft
check_bsd
write_bsdl -naming_check BSDL -output private.bsdl

# Generated patterns include private instr. patterns
set test_bsdvec_create_patterns_for_private_instructions TRUE
create_bsd_patterns
write_test -format stil -output private
```

```
create_bsd_patterns
write_test -format stil_testbench -output all_priv.v
```

Writing the STIL Protocol File for Instructions

BSD Compiler supports writing the STIL protocol file for the instruction that follows:

```
write_test_protocol -instruction MY_INST -output my_inst.spf
```

The SPF file can be generated as soon as the `insert_dft` or the `check_bsd` command is completed.

Implementing IEEE Std 1149.6-2003 Instructions

The following instructions are generated automatically if IEEE Std 1149.6-2003 is specified with the `set_bsd_instruction` command.

- EXTEST_PULSE
- EXTEST_TRAIN

See Chapter 3, "Inserting Boundary-Scan Components for IEEE Std 1149.6-2003."

Implementing Scan-Through-TAP

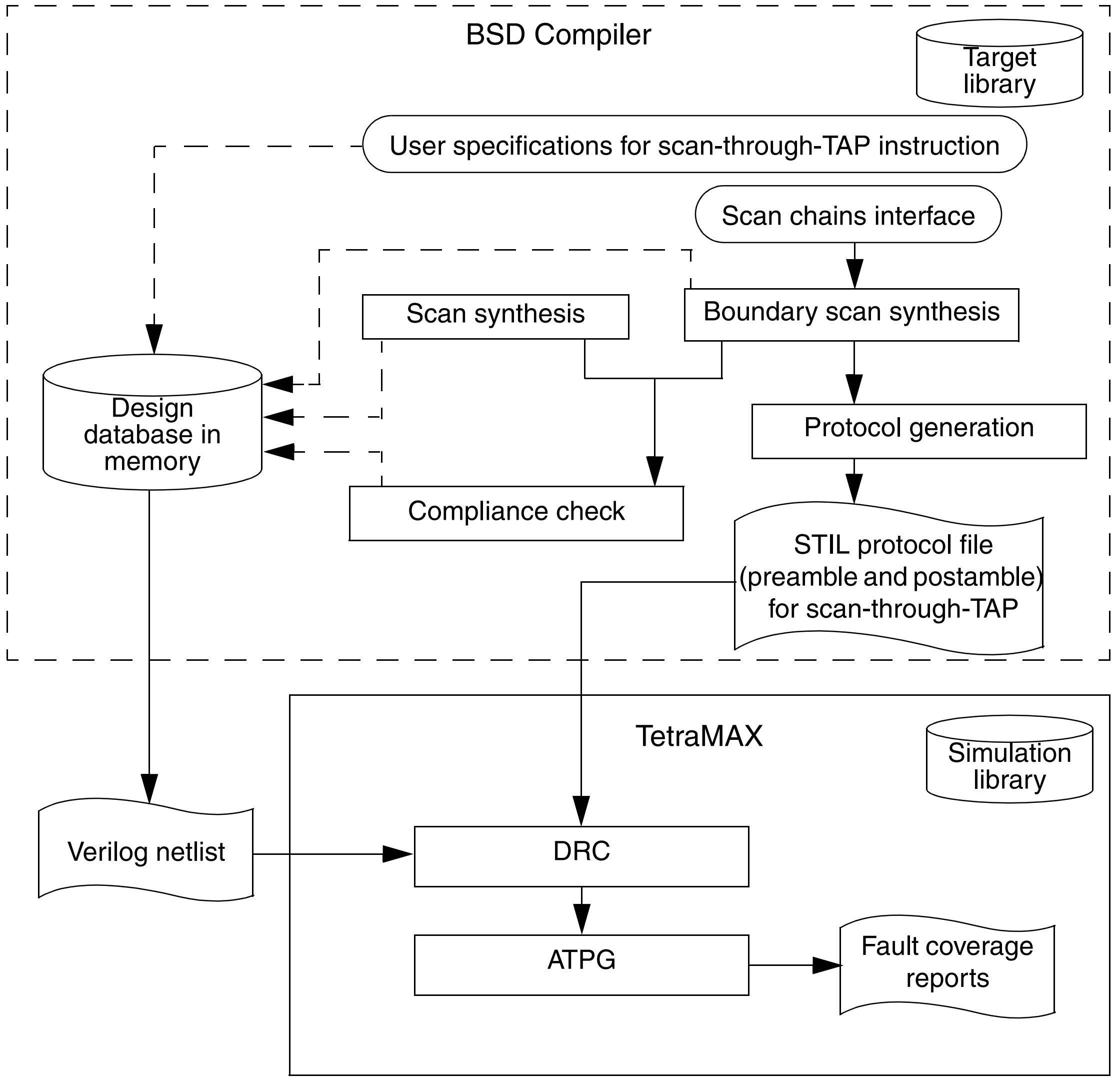
The scan-through-TAP capability in BSD Compiler allows you to daisy chain the scan chains with the BSR chain to become a single chain between TDI and TDO. This results in saving scan-in, scan-out, and scan-enable pins. You implement scan-through-TAP using the IEEE Std 1149.1 user-defined instruction, which selects the scan chains between TDI and TDO.

Using scan-through-TAP, you can automate the process of

- Specifying the scan-through-TAP register containing multiple scan-chains and boundary-scan registers
- Daisy-chaining the specified scan chains or the BSRs in the scan-through-TAP register
- Synthesizing the circuitry to control the scan-chain operation by IEEE Std 1149.1 instruction
- Generating STIL protocol file for TetraMAX ATPG

Figure 2-14 shows the scan-through-TAP flow through BSD Compiler to TetraMAX.

Figure 2-14 Scan-Through-TAP Flow



By using the scan-through-TAP flow provided in this section, you can specify the scan-through-TAP register, specify the necessary instructions, implement scan-through-TAP, and generate the appropriate protocol file.

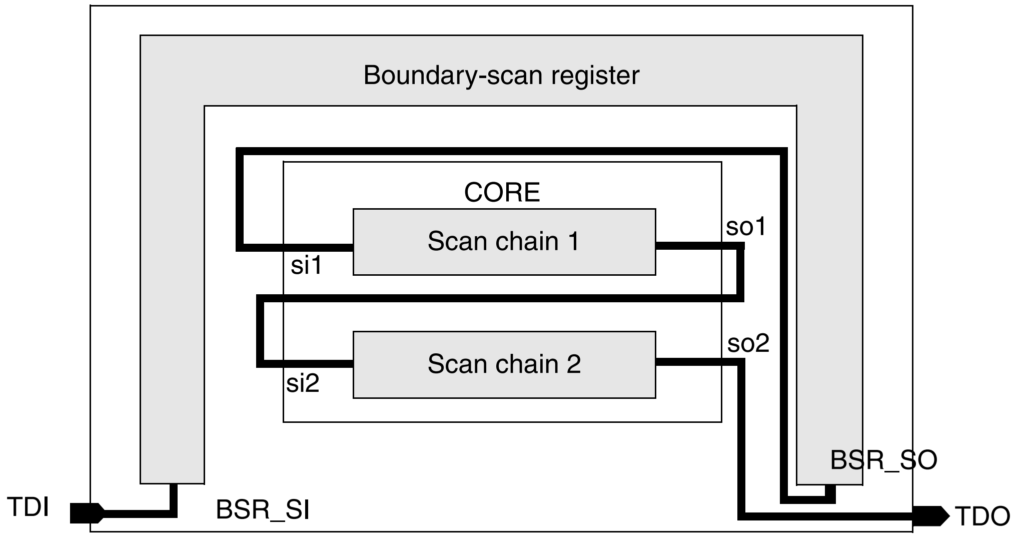
Specifying the Scan-Through-TAP Register

Use the `set_dft_signal` and `set_scan_path` commands to specify the STT register. To specify the appropriate scan-in and scan-out pins, use the signal types TDI and TDO. These signal types must appear alternately and in the exact order in which the scan chains have been connected in the register. Use the reserved keywords BSR_SI nd BSR_SO to specify the boundary-scan register (BSR) serial input and output. Use the reserved keyword STT_REG to specify the scan chain's scan in and scan out pins.

You can use the following commands to construct a scan-through-TAP register as shown in Figure 2-15.

```
set_dft_signal -view spec -type tdi \
               -hookup_pin core/si1 \
set_dft_signal -view spec -type tdo \
               -hookup_pin core/so1 \
set_dft_signal -view spec -type tdi \
               -hookup_pin core/si2 \
set_dft_signal -view spec -type tdo \
               -hookup_pin core/s02 \
set_dft_signal -view spec -type bsd_shift_en \
               -hookup_pin core/se \
set_dft_signal -view spec -type capture_clk \
               -hookup_pin core/clk \
set_scan_path STT_REG -class bsd -view spec \
     -hookup {BSR_SI BSR_SO core/si1 core/so1 core/si2 \
core/so2 core/se core/clk} -exact_length 50
set_bsd_instruction STT -register STT_REG -code 1101
```

Figure 2-15 Scan-Through-TAP Register



Note:
Multiplexers are added to the scan input ports for functional mode.

Specifying the Scan-Through-TAP Instruction

Use the `set_bsd_instruction -view spec STT` command to identify the desired instruction (STT_REG) as the scan-through-TAP instruction, and implement the required logic. Only one scan-through-TAP instruction can be implemented at a time.

You can use the following command to implement the scan-through-TAP instruction.

```
set_bsd_instruction -view spec {my_stt_instruction] \
     -register STT_REG -code {1101]
```

Writing the STIL Protocol File for STT

BSD Compiler supports writing the STIL protocol file for the Scan-Through-TAP instruction as follows:

```
write_test_protocol -instruction STT -output bsd_stt.spf
```

After you specify the scan-through-TAP register and instruction, `insert_dft` synthesizes the logic for both synchronous and asynchronous implementation. When you run `preview_dft`, it treats and reports the scan-through-TAP instruction as it would any other user-defined instruction and the STT_REG as any other user test data register (TDR).

The `insert_dft` command needs only the scan register interface to create the necessary logic. You can add your scan chains either before or after you run `insert_dft`, but you must insert them before running `check_bsd`. BSD Compiler connects the asynchronous reset pins in the same manner as it would for other TDRs.

Implementing a User Test Data Register Controlled by TAP

Implementing a Test Data Register control by TAP follows a design flow similar to the STT design flow. However, in this case you do not have to specify the BSR_SI and the BSR_SO signals as they are not a part of the TDR controlled by TAP. You do need to specify the TDR register name and the user-defined instruction name. The STIL protocol file is supported.

Example 2-15 is an example script for this flow.

Example 2-15 Script for Implementing a User Test Data Register Controlled by TAP

```
##### Define UTDR Register ######
set_dft_signal -view spec -type tdi -hookup_pin i_tm_reg/TM_TDI
set_dft_signal -view spec -type tdo -hookup_pin i_tm_reg/TM_TDO
set_dft_signal -view spec -type bsd_shift_en -hookup_pin i_tm_reg/ \
   TM_SHIFT_ENABLE
set_dft_signal -view spec -type capture_clk -hookup_pin i_tm_reg/ \
   TM_CAPTURE_CLK
set_dft_signal -view spec -type bsd_reset -hookup_pin i_tm_reg/ \
   TM_RESETN -active_state 0
set_scan_path TM_REG -class bsd -view spec \
   -hookup [list i_tm_reg/TM_TDI i_tm_reg/TM_TDO i_tm_reg/ \
   TM_SHIFT_ENABLE i_tm_reg/TM_CAPTURE_CLK] -exact_length 25
set_bsd_instruction TESTMODE -code {1010} -register TM_REG
preview_dft -bsd all
insert_dft
...
```

Inserting Scan on Core Logic

Before inserting boundary-scan logic, you might want to insert scan on your core logic using DFT Compiler. You have two methods available to do this:

1. Run a 1-Pass scan synthesis and route your scan chains later. The 1-Pass scan synthesis works on the RTL netlist and only replaces nonscan cells with scan cells.
2. Run a simple compile and replace and route your scan chain at the gate level.

   Note:
   If you are implementing STT or UTDR controlled by TAP, the scan logic must be mapped to gates for BSD Compiler to validate the shift path of the scan logic.

For more details about scan insertion methodology, see the *DFT Compiler Scan Synthesis User Guide*.

Reducing the Number of Control Cells

By default, each tristate output has its own dedicated control BSR cell after `insert_dft` is run. If several tristate outputs are controlled by the same enabling signal, you can save some of the area by using one BSR cell to control them. The number of tristate outputs that can be controlled by one control BSR cell is defined by the command `set_bsd_configuration -control_cell_max_fanout` *`max_fanout`*.

The *`max_fanout`* is an integer that specifies the number of cells driven by a single BSR cell.

Note:
Select an optimum `max_fanout` number of controlled BSR cells for the tristate function of the bus to prevent ground bouncing on tristate output buses. See the *BSD Compiler Reference Manual* for more information on ground bounce issues.

Implementing the Short BSR Chain

BSD Compiler supports the partition of the BSR chain into multiple BSR chain segments. These segments are referred to known as short-BSR-chains.

With the support of Short-BSR-Chains, the BSR chain inserted by BSD Compiler can be partitioned in multiple BSR chains and access maintained to the full BSR chain. BSD Compiler adds MUXs as appropriate to support Short-BSR-Chain and the decoded logic to connect the Short-BSR-Chain one at a time in between TDI and TDO. The default is the full BSR chain.

This capability enables you to select a few BSR cells from the full BSR chain and preload values without the need to clock the full BSR chain, saving clock cycles for any ASIC.

As shown in Figure 2-16, the TDI port of the design is connected to all the Short-BSR-Chain TDI inputs. The TDO output port of the design has a MUX selector controlled by the TAP instruction enable to connect one BSR chain at a time between TDI and TDO.

Figure 2-16 Multiple BSR Chains

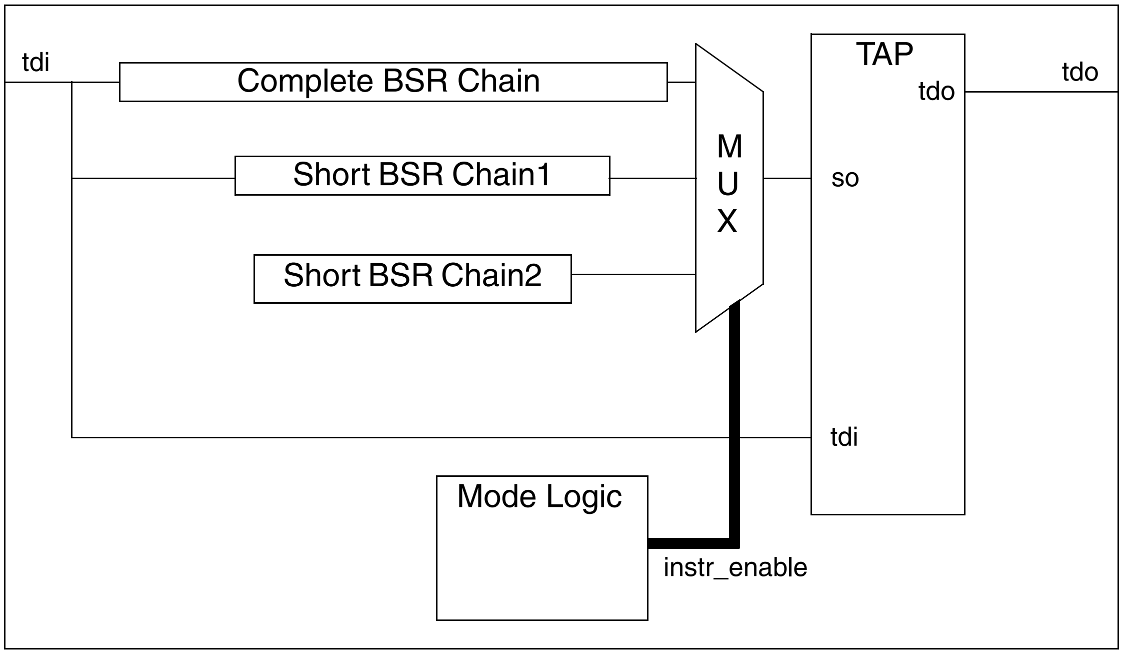
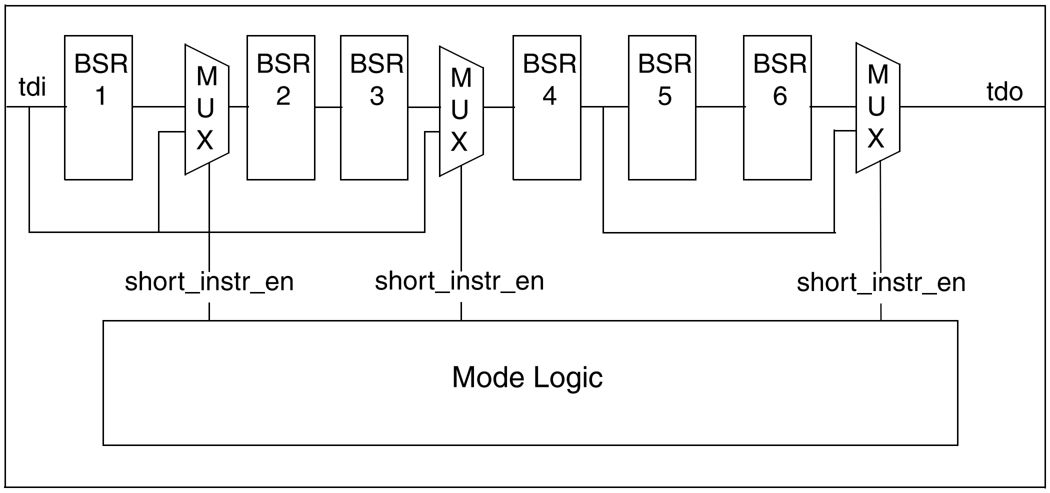


Figure 2-17 shows the reconfigurable MUXs added to the full BSR chain to form the individual short BSR chains in between TDI and TDO. The default BSR chain is the full BSR chain with all the six BSR cells, whereas the short BSR chain1 contains only BSR cells 2, 3, and 4, and the short BSR chain2 contains only BSR cells 4, 5, and 6.

Figure 2-17 Multiple BSR Chain MUX Logic



BSD Compiler allows the specification of BSR cells for a short BSR chain. The short BSR chain is constructed only from the specified BSR cells. The order of BSR cells in the short BSR chain matches the order in the specification. If a BSR cell of an embedded BSR in PAD (bsr_segment) is part of a short BSR chain, all other embedded BSR cells of the pad are also included in the short BSR chain.

Associating Short-BSR-Chains With User Instructions

BSD Compiler allows the association of short BSR chains with user instructions.

When user instructions that use short BSR chains are active, the associated short BSR chain is selected between TDI and TDO.

The `set_scan_path` command accepts short BSR chain descriptions. The chain name `boundary` option describes the default BSR chain. All other names specify the user-defined BSR chains. The chain name is used with the `set_bsd_instruction` to associate the BSR chain with a user specified instruction. You make no change to the `ordered_elements` option specified with the command.

The `set_bsd_instruction` command accepts the association of short BSR chains with user specified instructions, using the `-register` and `-exclude_bsr_condition` options.

Option	Description	Choices
`-register` *name*	Accepts BSR chain name as a valid register associated with the user instruction. If the name matches both a BSD register and a BSR chain, the BSD register is associated with the user instruction.	
`-excluded_bsr_condition CLAMP | NONE`	Specify conditioning of BSR cells excluded from the specified short BSR chain.	CLAMP - all BSR cells that are not part of the specified BSR chain are conditioned as if CLAMP instruction is active. NONE - all BSR cells that are not part of the specified BSR chain are kept transparent while the user instruction is active. Note: The error, Invalid value '%s' specified for option '-excluded_bsr_condition', occurs when a value other than CLAMP or NONE is used with this option.

The following `set_bsd_instruction` command options cannot be used when the TDR is a short BSR chain.

- `-input_clock_condition`
- `-output_condition`
- `-internal_scan`

If these options are used when the TDR is a short BSR chain, the following error message is displayed:

```
Error: Invalid option '%s' specified with the specified TDR.
```

The `-excluded_bsr_condition` option cannot be used with an instruction that does not use short BSR chain as TDR. If it is used, the following error message is displayed:

```
Error: Invalid option '%s' specified with the specified TDR.
```

When a specified short BSR chain is not associated with any user instructions, the following warning message is displayed:

```
Warning: Ignoring short BSR chain '%s'.
```

Because the short BSR chain is not used in any instruction, it is not implemented.

Short BSR Chain Instructions and EXTEST Instructions

When a short BSR chain instruction is active, the BSR cells of the short BSR chain behave as if EXTEST is selected; that is, conditioning for BSR cells is supported for short BSR chains.

For example, the mode pins of input BC1 BSR cells are connected to `(EXTEST | `*`short_bsr_chain_instr_en1`*` | `*`short_bsr_instr_en2`*` ...)`.

Similarly the mode pins output BC1 or BC2 BSR cells or mode1 pin of BC7 cells are connected to `(EXTEST | <short_bsr_instr_en1> | <short_bsr_instr_en2> ...)`.

BSD Compiler allows multiple user instructions that use short BSR chains as TDR.

Conditioning for Excluded BSR Cells

When the short BSR chain is selected as Test Data Register (TDR) between TDI and TDO, BSD Compiler allows the following types of conditioning for BSR cells excluded from a short BSR chain:

- CLAMP - All BSR cells excluded from the short BSR chain are conditioned as if CLAMP instruction is selected when the short BSR chain is selected.
- NONE - All BSR cells excluded from the short BSR chain are in transparent mode when the short BSR chain is selected.

Previewing Short BSR Chain Instructions

BSD preview shows the short BSR chain name as the TDR for the user instructions that use short BSR chains. BSD preview shows if a MUX has been added to the TDO of a BSR cell in the BSR cell description section. The default full BSR chain is used as the TDR for instructions EXTEST, SAMPLE, and PRELOAD. There is no change in conditioning logic added to (all) BSR cells for HIGHZ or CLAMP instructions.

Compliance Checking of Short BSR Chain Instructions

For short BSR chain instructions, no additional checks are done other than what is currently done for a non-BSR TDR. (Only shift path is validated.)

A compliance check shows the short BSR chain name as the TDR for the user instructions that use short BSR chains.

BSDL Support for Short BSR Chain Instructions

BSDL generated by BSD Compiler shows the short BSR chain name as the TDR for the instructions that select them.

BSD Vector Support for Short BSR Chain Instructions

No new UI commands have been created to support instructions specific short BSR chains.

Short BSR Chains Example Script

The Example 2-16 script uses the design M1, which has input port in1, two state output port out1, and tristate output port tri1, and a bidirectional port bidi1. This example shows how to specify Short-BSR-chains with CLAMP or NONE conditioning for excluded BSR cells.

Example 2-16 Script for Short BSR Chain

```
# read Verilog file and technology libraries
 current_design M1
 set_dft_signal -type tck -port tck
 set_dft_signal -type tdi -port tdi
 set_dft_signal -type tdo -port tdo
 set_dft_signal -type tms -port tms
 set_dft_signal -type trst -port trst
# Default BSR chain - all other BSR cells (tri1, bidi1) will be
# added to the full BSR chain
set_scan_path -class bsd boundary -ordered_elements {in1 out1}
# SHORT_BSR_CHAIN1 contains BSR cells for ports in1 and tri1.
set_scan_path SHORT_BSR_CHAIN1 -class bsd \
                               -ordered_elements {in1 tri1}
# SHORT_BSR_CHAIN2 contains BSR cells for ports out1 and bidi1
set_scan_path SHORT_BSR_CHAIN2 -class bsd \
              -ordered_elements {out1 bidi1}
# Instruction EXTEST_FLASH_CLAMP selects SHORT_BSR_CHAIN1 between
# TDI and TDO, BSR cells of ports out1, bidi1 are in CLAMP
# condition
set_bsd_instruction EXTEST_FLASH_CLAMP -register SHORT_BSR_CHAIN1 \
                    -excluded_bsr_condition CLAMP
# Instruction EXTEST_FLASH_TRANS selects SHORT_BSR_CHAIN2 between
# TDI and TDO, BSR cells of ports in1, tri1 are in transparent
# condition
set_bsd_instruction EXTEST_FLASH_TRANS -register SHORT_BSR_CHAIN2 \
                    -excluded_bsr_condition NONE
# preview shows SHORT_BSR_CHAIN1 as TDR for EXTEST_FLASH_CLAMP
# instruction, SHORT_BSR_CHAIN2 as TDR for EXTEST_FLASH_TRANS
# instruction.
preview_dft -bsd all
```

Previewing the Boundary-Scan Design

You can preview your boundary-scan design before you generate it by using the `preview_dft` command. This command generates a preview of the boundary-scan designs, including TAP ports, and it reports information about pad design (soft macro and library cells), test data registers, boundary-scan registers, and instructions.

The syntax of the command is

```
preview_dft -bsd tap
            -bsd cells
            -bsd data_registers
            -bsd instructions
            -script
```

The syntax to specify all the options is

```
preview_dft -bsd all
```

Example 2-17 shows a typical report generated by `preview_dft -bsd all`.

Example 2-17 Boundary-Scan Design Information Described by preview_dft -bsd all

```
****************************************
Preview bsd report
Design : TOP
Version: A-2007.12
Date   : Wed November 14 09:39:01 2007
****************************************
Number of TAP ports          : 5
port type     port name     pad pin(s)     package pin
---------     ---------     ------         -----------
TCK           tck           U8/Z           P3
TDI           tdi           U6/Z           P5
TDO           tdo           U4/A,U4/E'     P10
TMS           tms           U7/Z           P4
TRST          trst_n        U5/Z           P6

Test Logic Reset Method: Synchronous and Asynchronous(TRST)
Number of test data registers: 3
Mandatory:
Register      Length
--------      ------
BYPASS        1
BOUNDARY      6
Optional:
Register      Length
--------      ------
```

```
DEVICE_ID     32
Instruction Register Length  : 4
Instruction Encoding         : binary
Number of instructions       : 9
Instructions that select the register 'BYPASS':
BYPASS       1111
HIGHZ        0001
PRV1         0110
UDI1         0011
Instructions that select the register 'BOUNDARY':
EXTEST       1010
PRELOAD      1100
SAMPLE       1100
Instructions that select the register 'DEVICE_ID':
IDCODE       1011
USERCODE     1101
IDCODE capture value:
  Manufacturer id   : 3
  Part Number       : 2
  Version Number    : 1
Number of unused opcode(s) mapped to the BYPASS instruction: 8
Number of ports reduced or disabled: 0
Boundary Scan Register length: 6
index     port      pin(s)      package pin     function      type
impl        ccell     disval     rslt
-----     ----      ------      -----------     --------      ----
----        -----     ------     ----
5         clk       U10/Z       P1              clock         BC_4
DW_BC_4     -         -          -
4         in0       U100/Z      P7              observe_only  BC_4
DW_BC_4     -         -          -
3         *         U200/E'     -               control       BC_2
DW_BC_2     -         -          -
2         out0      U200/A      P8              output3       BC_1
DW_BC_1     3         1          Z
1         *         U300/E'     -               control       BC_2
DW_BC_2     -         -          -
0         out1      U300/A      P9              output3       BC_1
DW_BC_1     1         1          Z
1
```

Generating the Boundary-Scan Design

As discussed in the previous sections, these are the steps you take before generating a boundary-scan design:

1. Read the technology library.
2. Read the RTL or gate-level netlist.

3. Define the IEEE Std 1149.1 test access ports.
4. Identify linkage ports.
5. Specify the compliance-enable pattern.
6. Identify the clock signals.
7. Configure the device identification register.
8. Select the boundary-scan configuration for IEEE Std 1149.1 and IEEE Std 1149.6.
9. Select the TAP controller reset configuration.
10. Declare test data registers.
11. Specify instructions.
12. (Optional) Insert scan on the core logic.
13. Preview the boundary-scan design.

When you finish setting all your boundary-scan specifications, you can generate the boundary-scan design. You do this by using the `insert_dft` command.

When you issue this command, BSD Compiler generates the boundary-scan design, synthesizes it, and outputs a status message as shown in Example 2-18. When `insert_dft` is completed successfully, it returns a 1.

Example 2-18 insert_dft Command Output

```
Information: Changed wire load model for 'pad_out_lvttl' from '(none)' to '5K'.
                                                                      (OPT-170)
Information: Changed wire load model for 'pad_in' from '1000K' to '5K'.
                                                                      (OPT-170)
Information: Changed wire load model for 'pad_in' from '1000K' to '5K'.
                                                                      (OPT-170)
Information: Changed wire load model for 'pad_in' from '1000K' to '5K'.
                                                                      (OPT-170)
Information: Changed wire load model for 'pad_in' from '1000K' to '5K'.
                                                                      (OPT-170)
Information: Changed wire load model for 'pad_ref' from '(none)' to '5K'.
                                                                      (OPT-170)
Information: Changed wire load model for 'pad_io_sstl' from '(none)' to '5K'.
                                                                      (OPT-170)
Information: Changed wire load model for 'pad_io_sstl' from '(none)' to '5K'.
                                                                      (OPT-170)
Information: Changed wire load model for 'pad_io_sstl' from '(none)' to '5K'.
                                                                      (OPT-170)
Information: Changed wire load model for 'pad_io_sstl' from '(none)' to '5K'.
                                                                      (OPT-170)
Information: Changed wire load model for 'pad_io_sstl' from '(none)' to '5K'.
                                                                      (OPT-170)
Information: Changed wire load model for 'pad_io_sstl' from '(none)' to '5K'.
                                                                      (OPT-170)
Information: Changed wire load model for 'pad_io_sstl' from '(none)' to '5K'.
```

```
                                                                   (OPT-170)
Information: Changed wire load model for 'pad_io_sstl' from '(none)' to '5K'.
                                                                   (OPT-170)
Information: Changed wire load model for 'bsd_test' from '1000K' to '5K'.
                                                                   (OPT-170)
Validating data propagation functionality for all instances of pad design
    pad_io_sstl...
Validating data propagation functionality for all instances of pad design pad_in...
    pad_in...
Validating data propagation functionality for all instances of pad design
    pad_out_lvttl...
Validating tristate functionality for all instances of pad design
    pad_io_sstl...
Validating tristate functionality for all instances of pad design
    pad_out_lvttl...
Validating input side data propagation functionality for all instances of
    pad design pad_io_sst1...
  Generating the TAP
Setting local link library 'dw01.sldb dw04.sldb' on design
    'DW_tap_uc_width4_id1_idcode_opcode2_version14_part155_man_num292_sync_model'
Loading db file '/remote/srm317/clientstore/A2007.12_rel_sp/snps/
    synopsys-d/libraries/syn/dw01.sldb'
Loading db file '/remote/srm317/clientstore/A2007.12_rel_sp/snps/synopsys-d/libraries/
    syn/dw04.sldb'
  Allocating blocks in
    'DW_tap_uc_width4_id1_idcode_opcode2_version14_part155_man_num292_sync_mode1'
  Building model 'DW01_MUX'
  Building model 'DW01_mmux_width2'
  Building model 'DW01_mmux_width3'
  Building model 'DW01_mmux_width4'
  Building model 'DW01_mmux_width5'

Statistics for case statements in always block at line 152 in file
    './DW_TAPFSM_str.vhd.e'
===============================================
|          Line           |  full/ parallel  |
===============================================
|          155            |    auto/auto     |
===============================================
Setting local link library 'dw01.sldb' on design 'DW_TAPFSM_sync_mode1'
  Allocating blocks in 'DW_TAPFSM_sync_mode1'
  Building model 'DW01_NAND2'
  Building model 'DW01_NOT'
  Building model 'DW_TAPFSM_sync_mode1'
Information: full structuring bails at 25009 on 16 x 32 : 256. (OPT-555)
Setting local link library 'dw04.sldb' on design 'DW_INSTRREG_width4'
  Allocating blocks in 'DW_INSTRREG_width4'
  Building model 'DW_CAPTURE'
  Building model 'DW_INSTRREG_width4'
  Building model 'DW_BYPASS'
Setting local link library 'dw04.sldb' on design
    'DW_IDREGUC_version14_part155_man_num292'
  Allocating blocks in 'DW_IDREGUC_version14_part155_man_num292'
  Building model 'DW_IDREGUC_version14_part155_man_num292'
Created db design
    'bsd_test_DW_tap_uc_width4_id1_idcode_opcode2_version14_part155_man_num29
2_sync_mode1' from module 'DW_tap_uc'(impl: 'str', lib: 'DW04') for lib_cell
    'bsd_test_DW_tap_inst' of library 'DW04'
```

```
Structuring
    'bsd_test_DW_tap_uc_width4_id1_idcode_opcode2_version14_part155_man_num29
    2_sync_mode1'
Information: full structuring bails at 25008 on 19 x 17 : 245. (OPT-555)
  Mapping
    'bsd_test_DW_tap_uc_width4_id1_idcode_opcode2_version14_part155_man_num29
    2_sync_mode1'

    ELAPSED           WORST NEG TOTAL NEG  DESIGN
    TIME      AREA      SLACK     SLACK   RULE COST          ENDPOINT
  --------- --------- --------- --------- ---------
-------------------------
    0:00:27  1246.5      0.00       0.0       1.9
  Generating the BSR cells
Setting local link library 'dw04.sldb' on design 'DW_bc_2'
  Allocating blocks in 'DW_bc_2'
Setting local link library 'dw04.sldb' on design 'DW_CAPUP'
  Allocating blocks in 'DW_CAPUP'
  Building model 'DW_CAPUP'
Created db design 'bsd_test_DW_bc_2' from module 'DW_bc_2'(impl: 'str',
    lib: 'DW04') for
lib_cell 'bsd_test_m_dq0[0]_bsr1' of library 'DW04'
  Structuring 'bsd_test_DW_bc_2'
  Mapping 'bsd_test_DW_bc_2'
Setting local link library 'dw04.sldb' on design 'DW_bc_7'
  Allocating blocks in 'DW_bc_7'
Created db design 'bsd_test_DW_bc_7' from module 'DW_bc_7'(impl: 'str', lib: 'DW04')
     for lib_cell 'bsd_test_m_dq0[0]_bsr2' of library 'DW04'
  Structuring 'bsd_test_DW_bc_7'
  Mapping 'bsd_test_DW_bc_7'
    0:00:30      45.1      0.00       0.0       5.0
  Mapping 10 unused opcode(s) to the BYPASS instruction
  Generating the control logic
  Writing implemented instructions information to the design
  Creating Hierarchy for Boundary Scan Logic ...
Information: Changed wire load model for 'bsd_test_Decoder_inst_design'
    from '(none)' to
'1000K'. (OPT-170)
Information: Changed wire load model for 'bsd_test_BSR_mode_inst_design'
    from '(none)' to '1000K'.                                  (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_7_test_8' from
    '(none)' to '5K'.                                          (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_2_test_8' from
    '(none)' to '5K'.                                          (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_7_test_7' from
    '(none)' to '5K'.                                          (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_2_test_7' from
    '(none)' to '5K'.                                          (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_7_test_6' from
    '(none)' to '5K'.                                          (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_2_test_6' from
    '(none)' to '5K'.                                          (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_7_test_5' from
     '(none)' to '5K'.                                         (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_2_test_5' from
     '(none)' to '5K'.                                         (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_7_test_4' from
    '(none)' to '5K'.                                          (OPT-170)
```

```
Information: Changed wire load model for 'bsd_test_DW_bc_2_test_4' from
    '(none)' to '5K'.                                         (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_7_test_3' from
    '(none)' to '5K'.                                         (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_2_test_3' from
    '(none)' to '5K'.                                         (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_7_test_2' from
    '(none)' to '5K'.                                         (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_2_test_2' from
    '(none)' to '5K'.                                         (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_7_test_1' from
     '(none)' to '5K'.                                        (OPT-170)
Information: Changed wire load model for 'bsd_test_DW_bc_2_test_1' from
    '(none)' to '5K'.                                         (OPT-170)
Information: Changed wire load model for 'bsd_test_BSR_top_inst_design'
    from '(none)' to '1000K'.                                 (OPT-170)
Information: Changed wire load model for
    'bsd_test_DW_tap_uc_width4_id1_idcode_opcode2_version14_part155_man_num29
    2_sync_mode1' from '(none)' to '5K'.                      (OPT-170)
  Structuring 'bsd_test_Decoder_inst_design'
  Mapping 'bsd_test_Decoder_inst_design'
Information: Changed wire load model for 'bsd_test_Decoder_inst_design'
    from '1000K' to '5K'.                                     (OPT-170)
    0:00:31    1922.3      0.00       0.0      33.8
  Structuring 'bsd_test_BSR_mode_inst_design'
  Mapping 'bsd_test_BSR_mode_inst_design'
Information: Changed wire load model for 'bsd_test_BSR_mode_inst_design' from
    '1000K' to '5K'.                                                (OPT-170)
Information: Changed wire load model for 'bsd_test_BSR_top_inst_design' from
    '1000K' to '5K'. (OPT-170)
Information: Changed wire load model for 'bsd_test' from '1000K' to '5K'. (OPT-170)
1
```

The `insert_dft` command inserts IEEE Std 1149.1 and 1149.6 compliant boundary-scan circuitry by using DesignWare components. During synthesis, it maps all boundary-scan logic inserted by the command.

For information on Inserting Boundary Scan components for IEEE Std 1149.6-2003, see Chapter 3, "Inserting Boundary-Scan Components for IEEE Std 1149.6-2003."

As discussed in previous sections, the following design requirements must be fulfilled:

- All pads on design ports must be present and functional, unless the ports are specified as linkage bits.
- All mandatory TAP ports must be specified.

If pads or TAP ports are missing, or if TAP functionality is not present, errors are generated and the command terminates without completion. If pads are missing, they must be specified as linkage bits.

Writing a Final Gate-Level Netlist

If you want to perform any of the following three:

- Place and route of your design
- Automatic test pattern simulation (ATPG)
- BSD Pattern Simulation

you must first generate a gate-level netlist of your design. However, you do not need to generate the netlist to continue with verification of the boundary-scan design, which is covered in Chapter 4, "Verifying the Boundary-Scan Design."

You can write a gate-level netlist in VHDL, DDC, or Verilog formats. The following example writes out a Verilog netlist with the IEEE Std 1149.1 logic.

```
/* SYNOPSYS required */
change_names -rules [ verilog | vhdl ] -hierarchy

write -hierarchy -format verilog -output bsd_design.v
```

When using `create_bsd_patterns` and `write_test` and `write_bsdl -output` to write to a file, you need to make sure the file is write permissible, if it exists in the target directory.

Reading the RTL or Gate-Level Netlist

You need a properly formatted RTL or gate-level netlist before you can begin boundary-scan insertion.

This netlist must meet certain design requirements, which are detailed in the following sections.

Design Requirements

Your RTL design can be in either of the following Design Compiler approved formats:

- Verilog
- VHDL

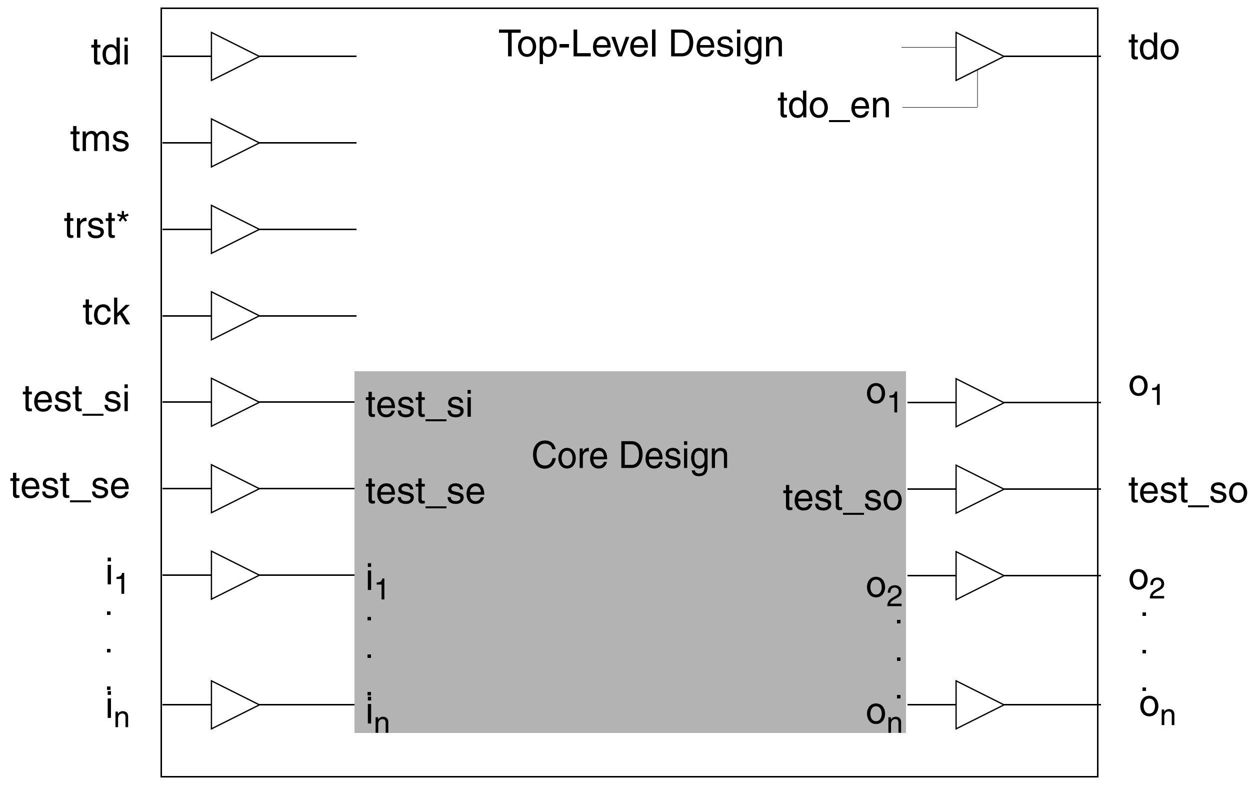
You can use a Verilog or VHDL gate-level netlist, but it should be in Synopsys database format (.db or .ddc) to preserve the attributes used during BSD Compiler synthesis. If you do use a gate-level netlist, you need not synthesize the core.

Your top-level design must have the following three characteristics:

1. The interface between the core and the boundary-scan logic must be defined. Every data pin in the core design interface must have a corresponding pad cell (see Figure 2-18). The core can contain logic, or it can be empty.

   If you use a design with an empty core for the purpose of increasing runtime performance, you must be aware of any feed-through logic in the core (for example, wires, clock signals) and any constraints that are provided to the I/O pads by the core (for example, enable signals, test modes) that you removed. The core logic is not necessary for compliance checking because no simulation is done on it, but any constraints provided to the I/O by the core must be satisfied to pass compliance.

Figure 2-18 Data Pins in Core Design Interface Have Corresponding Pad Cells



2. The top-level design must have I/O pad cells for all functional ports. The pad cells must be linked to the core design pins. There must be a one-to-one correspondence to top-level ports and core pins.

Note:
The pads for test access port (TAP) signals should be connected only on the port side of the design, not on the core side of the design. If connected to the core side, BSD Compiler breaks these connections as the pads must be dedicated pads as specified by IEEE Std 1149.1.

3. The design might or might not have scan chains inserted, but all scan ports must be defined. Each scan port at the core level must have a corresponding pad cell and port on the top level, regardless of whether scan has been inserted.

Reading the HDL Source Files

Input the HDL source files using the `read` command or the `analyze` and `elaborate` commands. For example, to read a Verilog design file, enter the following command.

Example 2-19 Reading a Verilog File

```
dc_shell> read_file -format verilog my_design.v
```

To read a VHDL design file, enter the following command.

Example 2-20 Reading a VHDL File

```
dc_shell> analyze -format vhdl my_design.v
dc_shell> elaborate my_design
```

To save execution time in subsequent `dc_shell` sessions, you can save the design in .ddc format using the following command:

```
dc_shell> write -format ddc -hierarchy -output my_design.ddc
```

Read the .ddc file using the following command.

```
dc_shell> read_file -format ddc my_design.ddc
```

For more information about these commands, see the Design Compiler documentation.

Reading HDL Source Files With Library Pad Cells

BSD Compiler supports the use of library pad cells in the netlist. To use library pad cells in your design, you must set the pad_cell and is_pad attributes.

- `pad_cell : true ;`

  This attribute is set within the library cell, and its value should be true.

- `is_pad : true ;`

  This attribute is set within the library cell pin that is connected to the port, and its value should be true.

An example of a portion of the library cell, in which these attributes are defined, is shown in Example 2-21.

Example 2-21 Library Pad Cell

```
cell(IBUF1) {
  area : 2;
  pad_cell : true;

  pin(A) {
    direction : input;
    capacitance : 1;
    is_pad : true;
    hysteresis : true;
    input_voltage : CMOS_SCHMITT;
  ...
  }
```

You should make sure that each pad cell of your design has these two attributes set appropriately. If these attributes are missing on the pad cell, you can set them explicitly by using the `set_attribute` command as shown in Example 2-22.

Example 2-22 Setting Pad Cell Attributes

```
dc_shell> set_attribute [find lib_cell lsi_10k/IBUF1] \
               pad_cell true -type boolean
dc_shell> set_attribute [find lib_pin lsi_10k/IBUF1/A] \
               is_pad true -type boolean
```

Example 2-23 shows how to set pad cell attributes for internal pull-up.

Example 2-23 Setting Pad Cell Attributes for Internal Pull-up

```
set_attribute IOLIB_65_FT_M6_LL_65A/BT4TARP_FT_65/Z driver_type 0 \
              -type short
```

Reading HDL Source Files With Differential I/O Pad Cells

BSD Compiler supports the use of differential I/O pad cells in the netlist. To use the I/O pad cell, you must set the `complementary_pin` attribute in the library cell. Setting this attribute identifies the differential input inverting pin with which the noninverting pin is associated and from which it inherits timing information and associated attributes.

To set the `complementary_pin` attribute, use the syntax

```
complementary_pin "string" ;
```

Specify the complementary_pin attribute to get the correct `check_bsd` and BSDL generation. In the absence of correct attribute settings, `check_bsd` incorrectly infers the differential ports. If you use the pad design specification then `check_bsd` infers the differential port correctly.

Example 2-24 shows a differential I/O pad with the `complementary_pin` attribute set to pin PADN for pin PAD.

Example 2-24 complementary_pin Attribute

```
cell(diff_in) {
 pad_cell : true;
 pad_drivers : 1;
 cell_footprint : pc3d_2096;
 area : 2095.85299744898;
 scaling_factors : diff_in_factors ;
 cell_leakage_power : 19525.492 ;
 pin(PAD) {
  direction : input;
  is_pad : true;
  complementary_pin : ("PADN");
 }
...
```

If the `complementary_pin` attribute is missing on the differential pad cell, you can set it explicitly by using the `set_attribute` command as shown in Example 2-25.

Example 2-25 Setting the complementary_pin Attribute

```
dc_shell> set_attribute \
      [find lib_cell lib_name / cell_name] \
      differential_cell true -type boolean

dc_shell> set_attribute [find lib_pin lib_name/ \
      cell_name / pos_pin_name] \
      complementary_pin neg_pin_name -type string

dc_shell> set_attribute [find lib_pin lib_name/ \
      cell_name/neg_pin_name] \
      master_complementary_pin pos_pin_name \
      -type string
```

See the Library Compiler documentation for more information on defining differential I/O pad cells. For more information on understanding differential I/O pad cells, see the *BSD Compiler Reference Manual*.

Enabling BSD Compiler

Before you execute `preview_dft` and `insert_dft` commands you need to enable BSD Compiler as follows:

```
## enable bsd and disable scan ##
set_dft_configuration -bsd enable -scan disable
```

Specifying Complex Pad Designs

BSD Compiler allows you to specify complex pads in Liberty models and in Verilog structural models. The Verilog structural pad models are known as soft macro pads. For more information on the soft macro pads, see the *BSD Compiler Reference Manual*.

You provide the Verilog structural models for the pads. All instances must be supported by the technology library (no black boxes). The pad design model or the library cell must be loaded in the Design Compiler database.

To specify a complex pad, first identify the soft macro pad or the liberty pad models to be used. You can then associate the pad cell signals with the pad design pins using the `define_dft_design` command, as follows:

```
define_dft_design -design_name design_name -type PAD
     [-interface access_list] [-params param_list]
```

Table 2-11 describes the `define_dft_design`, also known as Pad Design, command options.

Table 2-11 The define_dft_design Command Options

Option	Description	Choices
`-design` *`design_name`*	Identifies the design in memory that is instantiated during boundary-scan insertion.	
`-type`	Specifies the PAD cell design	PAD - Specifies a PAD cell design TAP - Specifies a custom TAP design equivalent to DW_tap interface TAP_UC - Specifies a custom TAP design equivalent to DW_tap_uc interface
`-interface` *`access_list`*	Specifies the list of signal mapping triplets relating a signal type to a pin of the specified design. Valid signal mapping: signal_type pin_name pin_polarity	data_in – Input of OUTPUT2/ OUTPUT3/INOUT tristate buffer where the BSR can be inserted or for the tool to recognize this pin, if used for TDO

Table 2-11 The define_dft_design Command Options (Continued)

Option	Description	Choices
		data_out – Output of the INPUT/ INOUT buffer where the BSR can be inserted or for the tool to recognize this pin, if used for the TAP ports defined in set_dft_signal for TCK, TRST and others enable – Enable of the OUTPUT3/ INOUT tristate buffer where the control BSR can be inserted or for the tool to recognize, if used for TDO enable port - Input of INOUT pad, Output of OUTPUT2/OUTPUT3/INOUT pads low – When multiple enable pins are present, use low to control the enable signal by TAP to a LOW state
		high – When multiple enable pins are present, use high to control the enable signal by TAP to a HIGH state pin_polarity: h - active high polarity l - active low polarity receiver_p - specifies the core side pins of the positive Test Receivers of the pad receiver_n - specifies the core side pins of the negative test receivers of the pad ac_init_clk - specifies the clock pins of hysteretic memories of positive and negative test receivers of the pad ac_init_data_p - specifies the initialization pins of hysteretic memories of positive test receivers of the pad

Table 2-11 The define_dft_design Command Options (Continued)

Option	Description	Choices
		ac_init_data_n - specifies the initialization pins of hysteretic memories of negative test receivers of the pad ac_mode - specify whether the comparator is sensitive to input levels (DC mode) or sensitive to input transitions (AC mode). 1 indicates AC Mode and 0 indicates DC Mode Note: For test receivers of non-differential input/bidi ports, receiver_p, ac_init_data_p signal types should be used. A BC_1, BC_2, or BC_4 (default) BSR cell is inserted for receiver_p, receiver_n pins of a PAD cell.
`-params` *`param`*	Specifies the parameter for the pad cell design, as follows: pad_type string - type of pad cell	input – If the input option is defined, the signal_type defined by the -access option is expected to be data_out. output – If the output option is defined, the signal_type defined by the -access option is expected to be data_in. tristate_output – If the tristate_output option is defined, the signal_type defined by the -access option is expected to be data_in or enable. bidirectional – If the bidirectional option is defined, the signal_type defined by the -access option is expected to be data_in, data_out, or enable. open_drain_output – If the open_drain_output option is defined, the signal_type defined by the -access option is expected to be enable.

Table 2-11 The define_dft_design Command Options (Continued)

Option	Description	Choices
		open_source_output – If the open_source_output option is defined, the signal_type defined by the -access option is expected to be enable.
		open_drain_bidirectional – Expects enable and data_out.
		open_source_bidirectional – Expects enable and data_out.
	lp_time string - specifies the low pass time constant for the pad's test receiver	time_string should be a string of positive, nonzero real numbers, in units of nanoseconds
	hp_time string - specifies the high pass time constant for the pad's test receiver	time_string should be a string of positive, nonzero real numbers, in units of nanoseconds
	on_chip boolean - specifies that the pad's test receiver has on chip AC coupling	true | false
	The info specified for lp_time, hp_time, on_chip is used only in BSDL generation.	
	$differential$ string - Specifies whether the pad cell is a differential pad cell. For differential pad designs, both ports (high and low) must be specified. For other (non-differential) pad designs, the port should be specified if the pad design must be validated during preview_dft or insert_dft.	true | false
	lib_cell string - Specifies whether or not the pad design is a library cell.	true | false
	$disable_res$ string - Specifies the disable result for a tristate output pad if the write_bsdl command is used.	WEAK0 – external pulldown WEAK1 – external pullup PULL0 – internal pulldown PULL1 -internal pullup Z – high-impedance state

Note:
When setting the `$differential$` parameter for differential pad designs, you must specify both ports with pin polarity high and low. For all other nondifferential pad designs, you need to specify port only.

When setting pad design specifications to characterize the functionality of a differential pad using the pad design command `define_dft_design -interface port (h|l)`, you must specify (l) for inverted port and (h) for non-inverted port of the differential pad connected to the design ports, as show in Example 2-26.

Example 2-26 Setting Pad Design Specifications to Characterize Differential Pad Functionality

```
dc_shell> define_dft_design -design_name BUF_BIDI \
               -type PAD \
               -interface {data_out out h data_in in h \
                  enable en 1 port out h port out_n 1} \
              -params {$pad_type$ string bidirectional $differential$ \
                  string true}
```

Assume a design called my_output_buffer_inverted is used as a pad cell. This design has an input pin A, an output pin Z, and other input and output pins. Example 2-27 shows the command options to use to characterize the pad cell.

Example 2-27 Characterizing a Pad Cell

```
dc_shell> define_dft_design\
             -design_name my_output_buffer_inverted -type PAD \
             -interface {data_in A low port IO high} \
             -params {$pad_type$ string output}
```

Always specify the access on the opposite side of the pad. For example, data_out access belongs on input pads, and data_in access belongs on output, tristate_output, or bidirectional pads.

The polarity of the enable port on bidirectional and tristate output pads should be set to transparent mode or high impedance mode, the value of which depends on whether your pad is active-high or active-low.

When setting pad design specifications to characterize the functionality of a complex pad design using `define_dft_design -interface`, you must always specify the output port of the complex pad connected to the design port. Example 2-28 shows the command options to use to specify the port.

Example 2-28 Specifying the Output Port of a Complex Pad

```
dc_shell> define_dft_design -design_name BUF_TRI -type PAD \
             -interface {data_out ZI h data_in A h \
             enable EN  h port IO h} \
             -params {$pad_type$ string bidirectional}
```

The `define_dft_design` command supports open_source_output and open_source_bidirectional pads, as well as open_drain_output and open_drain_bidirectional. The open source pad goes to high impedance Z when its function is evaluated to logic 0 (0/Z). This pad is designed without the pull-down transistor. The open drain pad goes to high impedance Z when its function is evaluated to 1 (1/Z). This pad is designed without the pull-up transistor.

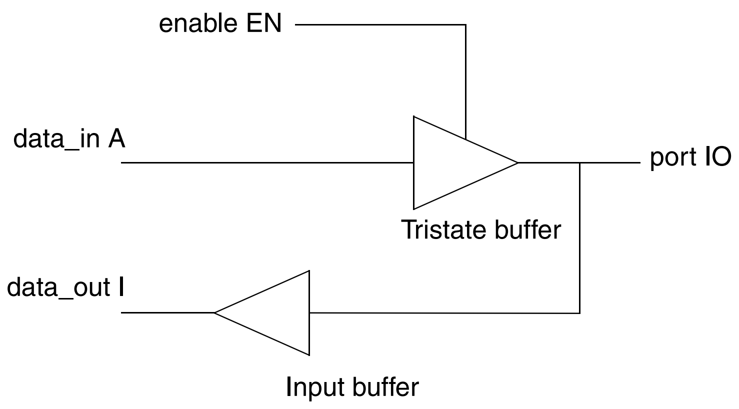
Note:
If you are validating soft macro pads or library pad cells with a pad design, you must constrain compliance-enable ports using `set_bsd_compliance` or pad designs command options high and low for the TAP to these compliance-enable ports. You can alternately provide the functional constraints of the pads in the design before running the `preview_dft` command, as shown in Example 2-29.

Example 2-29 Defining the Functional Constraints for a Pad

```
dc_shell> define dft_design -design_name BIDI_PAD_CELL \
               -type PAD \
               -params {$pad_type$ string bidirectional\
               $differential$  string false \
               $lib_cell$ string true } \
               -interface {data_in A h data_out I h \
               enable EN h port IO h}
```

See Figure 2-19 for the figurative representation of this example.

Figure 2-19 Figure for Example 2-29



Use the `set_bsd_configuration -check_pad_designs` command to control the validation of the soft macro pad designs.

Note:
When using the pad design command the pin polarity is high (h) for active high pin polarity, and low (l) for active low polarity; the pad design command supports both access types—high and low for the TAP FSM state to control this signal during compliance checking.

Using the Test Receivers in a Different Hierarchy

The following procedure describes another UI model that you can use for test receivers (TX).

- Ensure that the port is connected to the pad and the receiver cell.
- If the pad is differential, the differential attribute should be set on the pad and the differential ports should be connected to corresponding receiver cells.
- Both the pad and the receiver cells should be described with the `define_dft_design` command as pad designs.
- For the receiver cells, the interface should describe the receiver pin that is connected to the port as a port signal type.
- All other receiver pins are described the way that a pad design is described.
- Only the positive receiver cells should have the timing attributes.
- Ensure that the pad design name and the pad pin name pairs are not overloaded or have different semantics.

Use a unique design for the receiver cells of the port, so the tool knows which cell is used as the receiver for the positive and the negative.

Example 2-30 shows the UI model in which IEEE_1149_6_1V8 is used as a receiver cell for both positive and negative legs of the port. IEEE_1149_6_1V8_TR_N is a wrapper around IEEE_1149_6_1V8 that differentiates between the RX positive and negative models.

Example 2-30 UI Model for Using the Test Receivers

```
## RX for the positive leg
define_dft_design -design_name IEEE_1149_6_1V8 \
   -type PAD \
   -interface { port D h receiver_p TR h ac_init_clk IC h ac_init_data_p
ID h ac_mode AC h } \
   -params {$pad_type$ string input $lp_time$ float 5.0 $hp_time$ float
15.0}
## RX for the negative leg
define_dft_design -design_name IEEE_1149_6_1V8_TR_N \
   -type PAD \
   -interface { port D h receiver_n TR h ac_init_clk IC h ac_init_data_n
ID h ac_mode AC h } \
   -params {$pad_type$ string input}
## differential driver
define_dft_design -design_name LVDSREC1G25_1V8_STAG -type PAD \
      -params {$pad_type$ string input $differential$ string true
$lib_cell$ string true } \
     -interface {data_out DS h port VIA h port VIB l low EN h }
```

If the test receiver (RX) and the differential driver are in the same model, then only one pad design command is required for the pad and the two test receiver (RX) components. Example 2-31 is a script where the test receiver (RX) and the differential driver are in the same model.

Example 2-31 INPUT DIFF_RX_HYST Pad with Test Receiver and Hysteresis Model

```
## INPUT DIFF_RX_HYST pad with Test Receiver and Hysteresis model
define_dft_design -design_name DIFF_RX_HYST \
     -type PAD \
     -interface { port            t4_diffrx_in_pos h \
                  port            t4_diffrx_in_neg l \
                  receiver_p      t4_diffrx_out_pos h \
                  receiver_n      t4_diffrx_out_neg h \
                  data_out        t4_diffrx_out h \
                  ac_init_clk     t4_init_clk h \
                  ac_init_data_p  t4_init_data_p h \
                  ac_init_data_n  t4_init_data_n h \
                  ac_mode         t4_AC_Mode h } \
     -params {$pad_type$ string input $hp_time$ float 15.0 \
              $differential$ string true}
```

Specifying Complex Soft Macro Pads

When a Liberty pad cell has complex pin functions, the pad pin functions can be modeled as a soft macro pad cell using structural Verilog and standard library cells. These models can be instantiated as part of the design pad reference and specified in BSD Compiler using the pad design command. These include all combinations of single-ended and double-ended transmitter/receivers for pad design validation, synthesis, and compliance checking.

When specifying complex soft macro pads, the `define_dft_design` access signal types low, high, and observe are synthesized as follows:

- low and high

  If the pad access pins specified for `[high | low]` are floating (unconnected), they are tied to 1'b1 and 1'b0. If the pad pins are functionally connected to 1'b1 and 1'b0 in the netlist, no changes are made to them. Otherwise, they are connected to (func_drvr, test_logic_reset_state_inv), or (func_drvr, test_logic_reset_state).

  This assumes that the TAP controller is set to test logic reset during the mission mode

- observe

  BSD Compiler inserts OBSERVE Boundary-scan cells (BC_4 type) to data pins specified in the pad design command interface; the pin specified in the pad interface for observe cell insertion is not the data pin associated to the pad port. The OBSERVE BSR cells of output pads become internal cells in the BSDL file after compliance.

Use the `set_boundary_cell -function` command to define pin functions. You can specify the following types:

- input
- input_inverted
- output
- output_inverted
- bidir
- bidir_inverted
- control
- observe
- receiver_p
- receiver_n
- ac_select
- none

BSD Compiler inserts BSR cells at pin sites with the access signal type `observe`. The `-function` option of the `set_boundary_cell` command defines the BSR cell implementation. If the BSR cell identifier represents a BSR cell of type not equal to BC4, then a warning occurs and the default BSR cell type BC4 is used.

The `-function` option does not support specifying different BC4 implementations for different observe pins at a port pad. However, use the pad design command interface `observe` to specify different observe pins at a port pad.

The syntax for specifying an observe BSR cell for complex soft macro pads with the `set_boundary_cell` command is:

```
set_boundary_cell
         -type BC4
         -function observe
         -ports port_name
         -class bsd
```

The following example applied to the pad design in Figure 2-20 on page 2-85 results in the boundary scan design shown in Figure 2-21 on page 2-86.

Example 2-32 USB Differential Bidirectional Pad Cell

```
define_dft_design -design_name USB_DIFF_PAD -type PAD \
     -interface {data_in DPO h data_in DNO 1 enable EN_3 1 \
        port DPos h port DNeg 1 data_out DF_I h observer DPI h \
        observe DNI h low EN_2 h}
     -params {$pad_type string \
        bidirectional $differential$ string true}

set_boundary_cell -class bsd -type BC2 -function input \
     -ports BIDI
set_boundary_cell -class bsd -design BC2_FAST -function output_inverted \
     -ports BIDI
set_boundary_cell -class bsd -design BC4_SLOW -function observe \
     -ports BIDI
set_boundary_cell -class bsd -type BC2_FAST -function control \
     -ports BIDI -name CTL1
```

Figure 2-20 USB Differential Bidirectional Pad

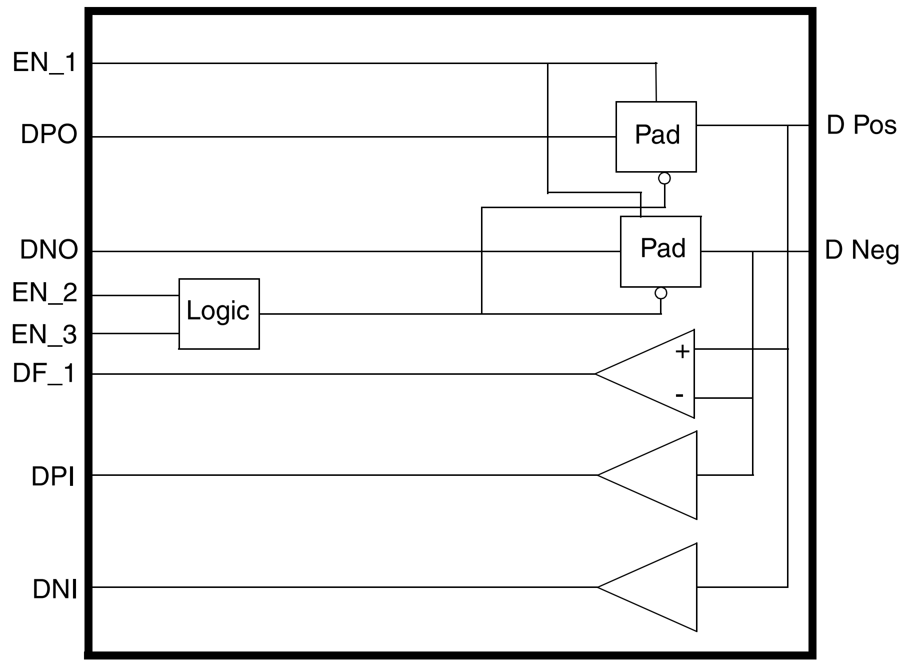
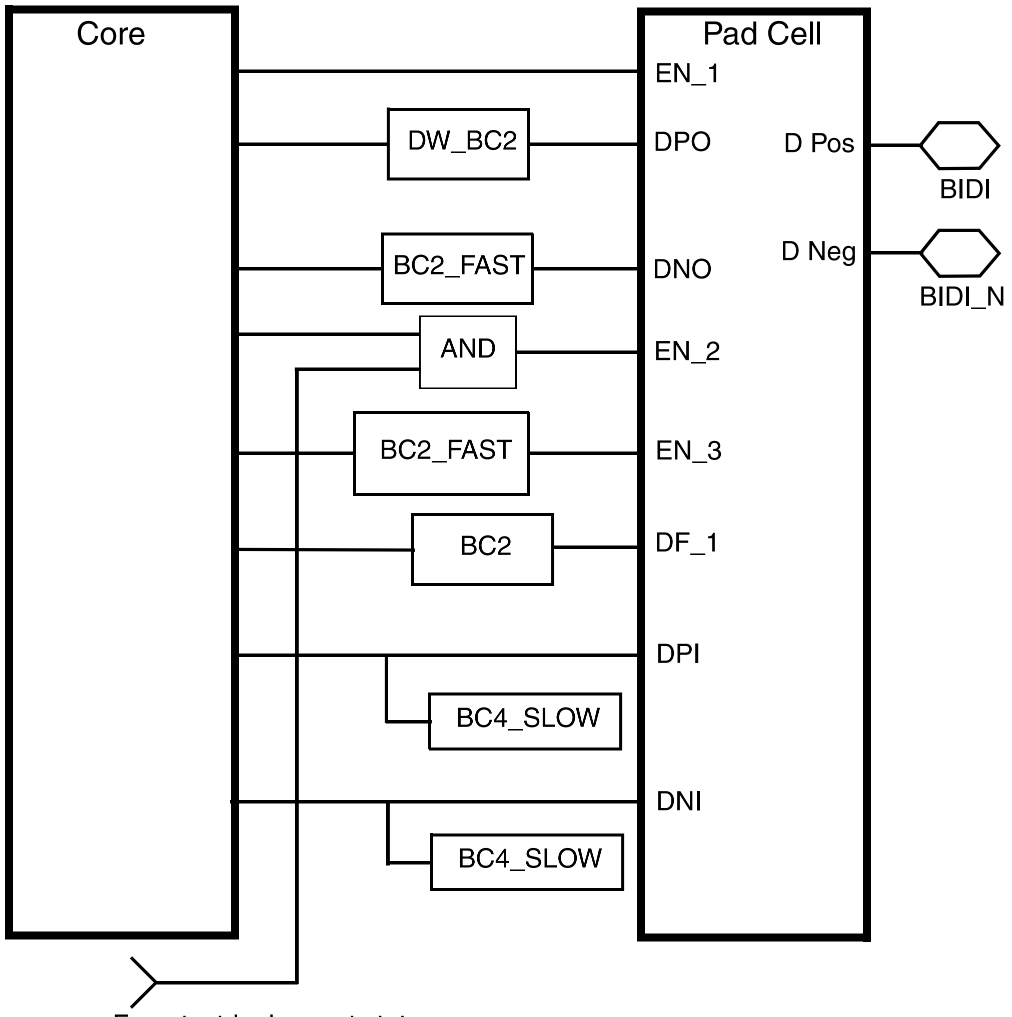


Figure 2-21 BSD for USB Differential Bidirectional Pad



From test-logic-reset state

Preview BSD supports independent differential legs for pad design validation. During synthesis, BSR cells can be inserted on both independent differential legs.

For differential output pads one data_in pin is required with the associated positive port of the differential pad. Compliance checking treats the differential legs independently on the output side.

The following example applied to the pad design in Figure 2-22 on page 2-87 results in the boundary scan design shown in Figure 2-23 on page 2-88.

Example 2-33 Pad Cell With Multiple Enable Pins

```
define_dft_design -design_name MY_IN_PAD -type PAD \
   -interface {data_in DO h enable EN_1 h data_out DI h low PI h high \
      EN_3 h high EN_4 h observe EN_2 h port PAD h} \
   -params {$pad_type$ string bidirectional}
```

Figure 2-22 Pad With Multiple Enable Pins

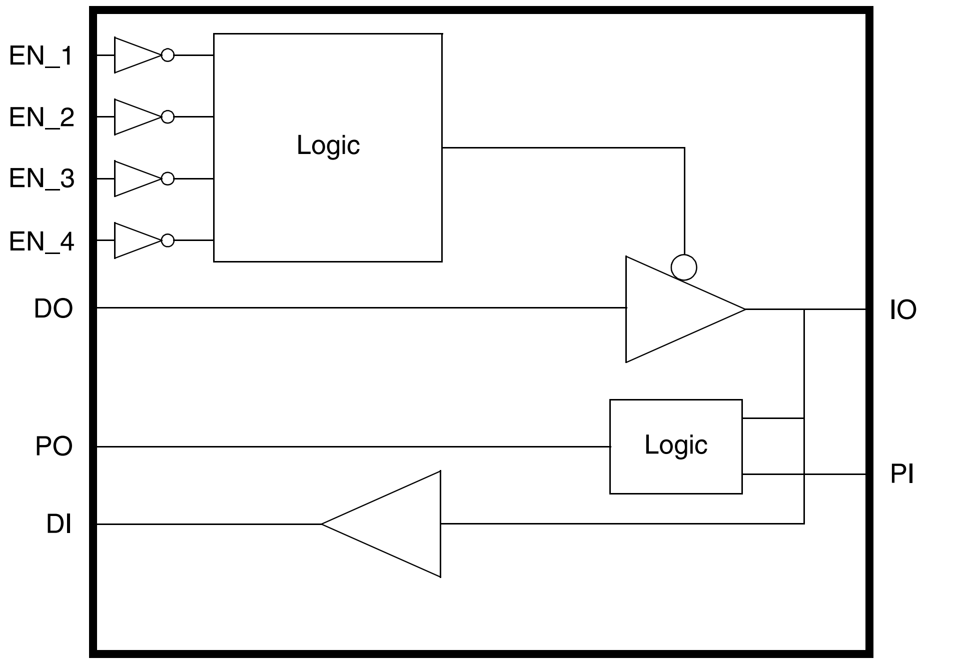
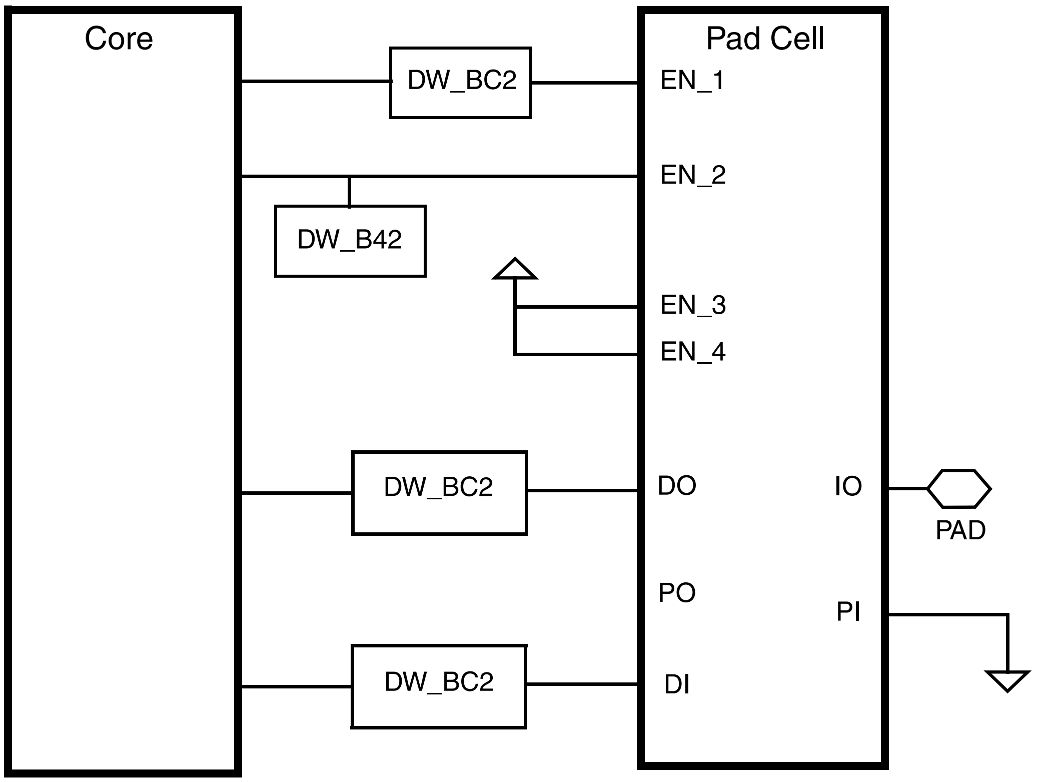


Figure 2-23 BSD for Pad Cell With Multiple Enable Pins



You can set the enable, data_out, and data_in pins with the `define_dft_design` command. In addition, you can insert observe-only BSR cells on any unused enable pins with the observe access signal type.

You can also drive some or all of the enable pins with logic 0/1 values.

RTL Generation Flow

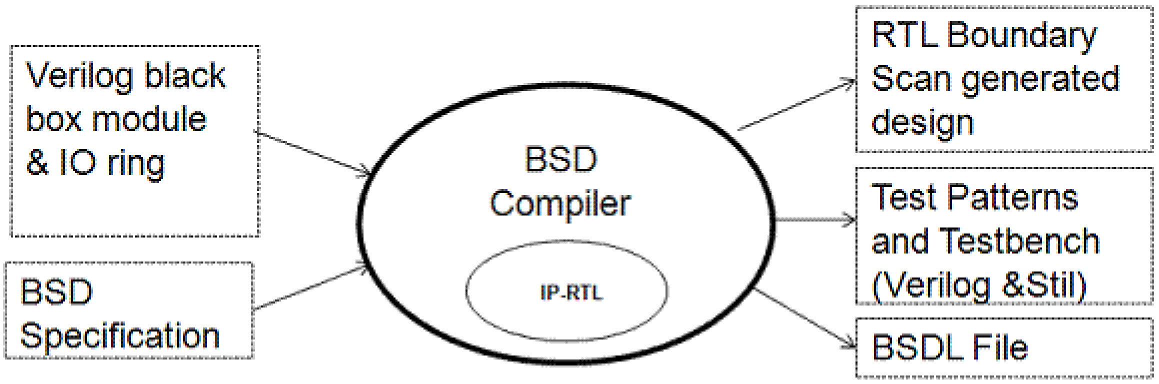
The RTL generation of the BSD components is based on the user's RTL specification and is intended to support designs for the IEEE Std 1149.1 and 1149.6 applications. The tool generates a complete RTL file for the TAP component, the boundary-scan chain, and the instruction register decoded mode block, and it connects the generated RTL logic.

The RTL generation flow requires a top-level Verilog black box module, including the I/O ring, with the port definitions and their functions and the I/O functional pads in Liberty format or as a soft macro model. No other RTL code is allowed.

As shown in Figure 2-24, the tool generates the following:

- The RTL components, connecting them to the pads
- The Verilog testbench
- The Boundary Scan Description Language (BSDL) file, requiring the pin map file
- The test patterns in STIL format

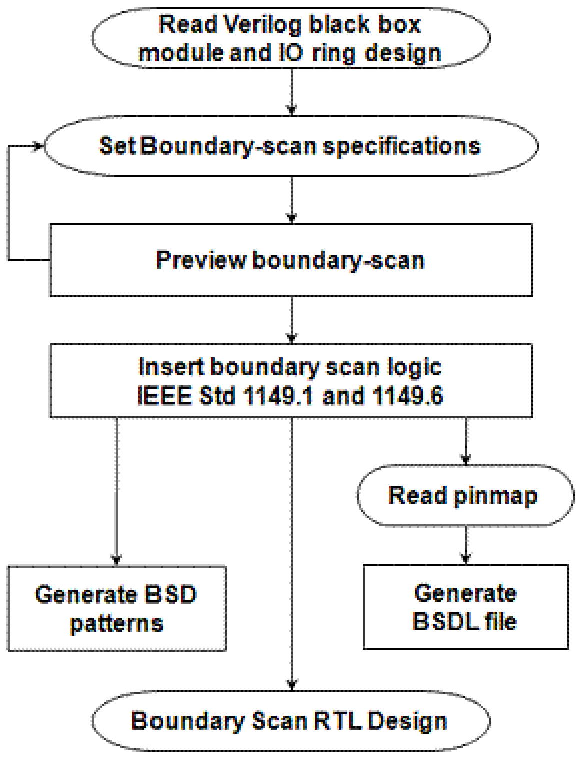
Figure 2-24 BSD Compiler RTL Generation I/O Diagram


Verilog black box module & IO ring
BSD Specification
BSD Compiler
IP-RTL
RTL Boundary Scan generated design
Test Patterns and Testbench (Verilog &Stil)
BSDL File

BSD Compiler generates boundary-scan RTL logic as a Verilog file. The IP-RTL components are generated from within BSD Compiler. The Verilog and STIL test patterns, along with testbenches, can be generated by the tool. The BSDL (Boundary Scan Description Language) file describes the Boundary Scan architecture of the generated BSD logic, which can also be generated by the BSD Compiler. The pin map information is required for BSDL file generation.

Figure 2-25 on page 2-90 shows the RTL generation flow for BSD Compiler, including the input requirements and the BSD specifications to generate the BSD RTL logic, the BSD patterns, and the BSDL file, which supports the IEEE Std 1149.1 and 1149.6.

Figure 2-25　BSD Compiler RTL Flow


Read Verilog black box module and IO ring design
Set Boundary-scan specifications
Preview boundary-scan
Insert boundary scan logic IEEE Std 1149.1 and 1149.6
Read pinmap
Generate BSD patterns
Generate BSDL file
Boundary Scan RTL Design

The BSD Compiler RTL design flow consists of the following steps:

1. Read the Verilog black box module and the I/O ring design.
2. Set the following boundary-scan design specifications:
   - Enable BSD Compiler for RTL generation.
   - Select the required IEEE Std 1149.1 or 1149.6.
   - Identify all boundary-scan test access ports and assign their attributes.
   - Identify leakage ports.
   - Specify the compliance-enable pattern.
   - Define clock ports.
   - Define the boundary-scan register configuration, and assign all boundary-scan register instructions.
   - Configure the device identification register and its capture value.

- Select boundary-scan cells from the default cell types.
- Read the pin map specification for BSDL file generation. (Optional)

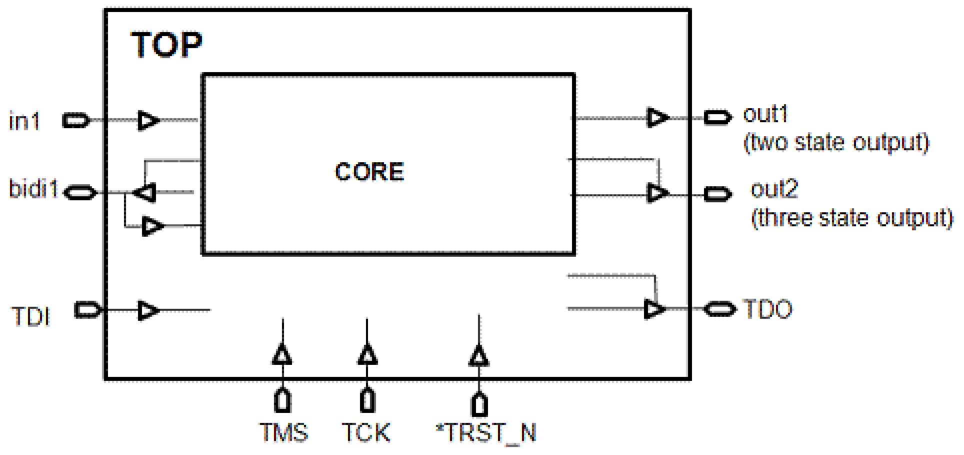
3. Preview the boundary-scan logic.
4. Run the `insert_dft` command to generate the boundary-scan RTL design.
5. Generate the boundary-scan design RTL file.
6. Generate the BSDL file and the functional, leakage, and DC parametric test vectors.

Input RTL Black Box Module and I/O Ring

You use a black box model for core logic. The I/O ring can be a functional model or a soft macro model. Advanced I/O pads can be block box models, but note that all I/Os are required for compliance simulation. The port types are identified for RTL BSR cell generation. For example, for input ports, the tool generates BC_2 RTL cells; for inout ports, the tool generates BC_7/BC_2 RTL cells; for output ports the tool generates BC_1 RTL cells; for output tristates, the tool generates BC_1/BC_2 RTL cells. The TAP ports do not have BC cells and must be specified in the black box design, as shown in Figure 2-26.

Figure 2-26 shows the Verilog black box module top-level design and the I/O ring with an empty core. The arrow directions show the type of port. In this case, in1 is the input port, bidi1 is the inout port, and out1 and out2 are output ports. Ports TDI, TDO, TMS, TCK, and TRST_N are TAP ports.

Figure 2-26 Input RTL Black Box Design



Example 2-34 shows a Verilog black box module `(input_file.v)` with its I/O ring netlist.

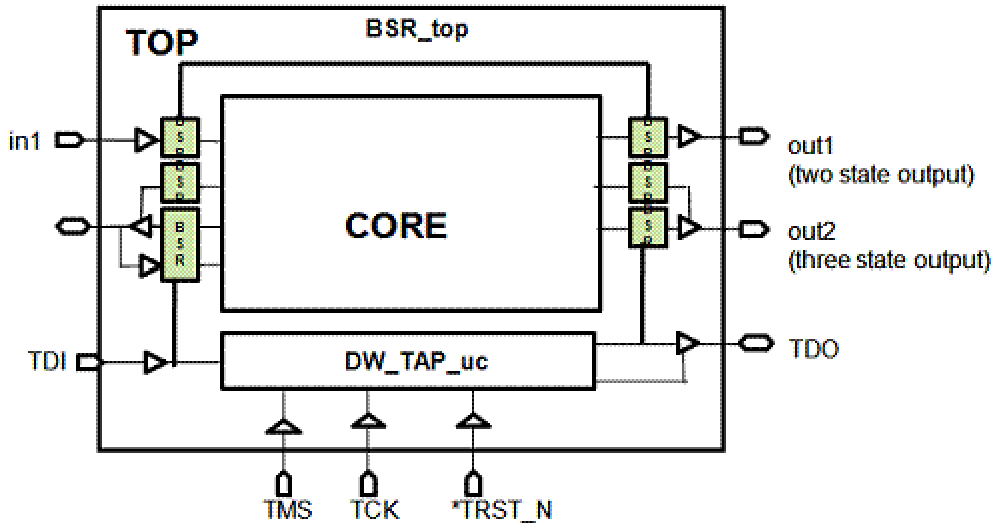
Example 2-34 Verilog Black Box With I/O Ring Netlist

```
module CORE (i_clk, i_d, i_in1, o_en1, o_en2, o_out1, o_out2, o_bidi1);
     input  i_clk,i_d;
     input  i_in1;
     output o_en1, o_en2;
     output o_out1,o_out2, o_bidi1;

endmodule

module TOP(clk,d,tck,tms,tdi,trst_n,tdo, in1,out1, out2, bidi1);
     input  d,clk;
     input  tck,tms,tdi,trst_n;
     output tdo;
     input  in1;
     output out1,out2, bidi1;

     wire i_clk, i_d, i_in1, o_en1, o_en2, o_out1, o_out2, o_bidi1;

     IBUF2 U2   (.A(trst_n),.Z());
     IBUF2 U3   (.A(tdi), .Z());
     IBUF2 U4   (.A(tms), .Z());
     IBUF2 U5   (.A(tck), .Z());
     BIDI  U6   (.A(),.E(),.Z(tdo));

     IBUF2 U7   (.A(in1), .Z(i_in1));
     IBUF2 U8   (.A(clk), .Z(i_clk));
     IBUF2 U9   (.A(d), .Z(i_d));

     BIDI  U10  (.A(o_bidi1), .E(o_en1), .Z(bidi1));
     OBUF2 U11  (.A(o_out1), .Z(out1));
     BIDI  U12  (.A(o_out2), .E(o_en2),.Z(out2));

     CORE core_inst (i_clk, i_d, i_in1, o_en1, o_en2, o_out1,
                     o_out2,o_bidi1);

endmodule
```

Output RTL Boundary Scan Generated Design

The generated output file is a Verilog format, behavioral RTL file. It includes the RTL modules for the boundary scan components and their connections to the pads, as well as the DW_TAP_uc module. Figure 2-27 shows a diagram of the RTL design after the `insert_dft` command is run.

Figure 2-27 Output Boundary Scan RTL Design

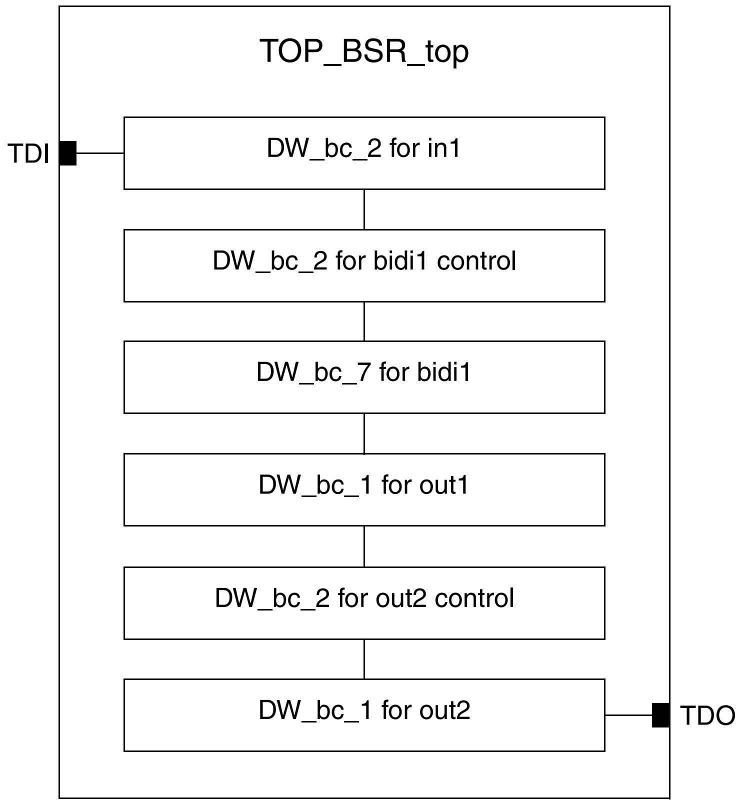


As shown in Figure 2-27, the RTL design is made up of the TAP module component and the BSR TOP (Boundary Scan Register) module component.

- The DW_TAP_uc module is the RTL component for the TAP controller. This component includes the following test data registers:
  - TAP FSM state machine register
  - Instruction Register (IR)
  - BYPASS Register
  - DEVICE ID Register (DIR)
- The BSR_top module describes the Boundary Scan Register (BSR) chain, which contains all the boundary-scan cells (BC) for all ports in the design. The boundary-scan cells are RTL components of the BSR chain.

The Boundary Scan Register chain is created by connecting the boundary-scan cells to form a chain from TDI to TDO. In this case, input port in1 is attached to BC_2 cell, inout port bidi1 is attached to BC_7 cell, and output ports out1 and out2 are attached to BC_1 cells. The BSR scan chain is created as shown in Figure 2-28.

Figure 2-28 Output TOP BSR Module Showing Scan Chain



RTL Generation User Interface Model

The user interface commands associated with RTL generation include

- The `-rtl` option for the `set_bsd_configuration` Command
- The `write_bsd_rtl` command
- Other commands used in the RTL Generation Flow

The -rtl Option for the set_bsd_configuration Command

The `set_bsd_configuration` command supports the `-rtl <enable|disable>` option. You use this option to enable the RTL generation flow.

The write_bsd_rtl Command

The `write_bsd_rtl` command is used to write the boundary-scan inserted RTL Verilog file. The command has the following syntax:

```
write_bsd_rtl -format verilog [-all|-tap|-bsr] -output rtl_filename.v
```

`-format`

Specifies Verilog format for the RTL output file. This is a default option.

`-all`

Specifies that all boundary scan logic is to be written into the designated output file. This is a default option.

`-tap`

Specifies that the TAP and its submodules are to be written into a separate file. For the TAP modules, the file prefix tap is pre-appended to the output file name, for example, tap_rtl_file.v. Note: other logic is written into the file specified with the `-output` option.

`-bsr`

Specifies that the BSR top module and its submodules are to be written into separate files. The prefix bsr is pre-appended to the output file name, for example. bsr_rtl_file.v. Note: other logic is written into the file specified with the -`output` option.

`-output`

Specifies the name of the output RTL file. This switch is mandatory.

Other Commands Used in the RTL Generation Flow

The following commands, used in the RTL generation flow, have no user interface changes and are functionally equivalent to the existing BSD Compiler user interface:

- `read_verilog` *`input_filename.v`*
- `set_bsd_configuration -rtl [enable|disable]`
- `set_dft_signal`
- `set_bsd_instruction`
- `set_bsd_linkage_port`
- `preview_dft`
- `insert_dft`
- `write_bsd_rtl -format verilog -all -output` *`rtl_filename.v`*
- `read_pin_map top.pinmap`

Example RTL Generation Run Script

Example 2-35 shows the order of events, that is, the order of commands, in the run script.

Example 2-35 RTL Run Script

```
#Read input black box design
read_verilog rtl_input.v

current_design TOP
set_dft_configuration -scan disable -bsd enable
set_bsd_configuration -std {ieee1149.1_2001}

#Enable RTL generation
set_bsd_configuration –rtl enable

#Specify TAP ports
set_dft_signal -type tdi -port TDI
set_dft_signal -type tdo -port TDO
set_dft_signal -type tck -port TCK
set_dft_signal -type tms -port TMS
set_dft_signal -type trst -port TRST_N

#Specify linkage ports
set_bsd_linkage_port -port_list {port_lkg}

#Preview Boundary Scan
preview_dft -bsd all

#Insert RTL Boundary Scan logic
insert_dft

# Write Boundary scan RTL Generation file rtl_out_file.v
write_bsd_rtl –format verilog –all –output rtl_out_file.v

#Read pin map info
read_pin_map top.pinmap

#Write BSDL file
write_bsdl -output  TOP_bsd.bsdl

create_bsd_patterns -type all

#Write STIL patterns
write_test -format stil_testbench -output TOP_debug_pat_tb

#Write Verilog patterns
write_test -format verilog -output TOP_debug_vlog_tb
```

Properties and Limitations of the RTL Generation Flow

Note the following properties and limitations of the RTL generation flow:

- User RTL logic, apart from the Verilog black box module at the top level and the I/O ring design of the input file, is missing in the generated output RTL file.
- Compliance checking is confirmed for the RTL netlist and the RTL BSD patterns by passing VCS simulation; the `check_bsd` command does not use RTL.
- The output RTL design can be mapped to logic by using Design Compiler commands and optionally by using the `check_bsd` command for compliance check.
- The RTL flow supports Verilog RTL format output.
- Custom TAP, custom BSRs, and user-defined test data registers are supported in the RTL flow if they are defined as black box references, that is, as empty modules.
- User-RTL in the input design is not supported.

3

Inserting Boundary-Scan Components for IEEE Std 1149.6-2003

This chapter describes how to use BSD Compiler to synthesize IEEE Std 1149.6 logic and generate an IEEE Std 1149.6 BSDL file and BSD patterns. The IEEE Std 1149.6 defines extensions to IEEE Std 1149.1 that standardize boundary-scan testing of advanced digital networks, especially for networks that are AC coupled, differential with test receivers, or both.

Note:
The BSD Compliance Check Specifications requires a gate level netlist to validate BSD logic. The checker validates the IEEE Std 1149.6 logic in accordance with the IEEE Std 1149.1, and it does not carry out any checks to validate the IEEE Std 1149.6 logic in accordance with IEEE Std 1149.6.

This chapter includes the following sections:

- BSD Compiler Commands
- IEEE Std 1149.6 Architecture
- IEEE Std 1149.6 Preview Specifications
- IEEE Std 1149.6 Synthesis Specifications
- BSDL Generation Specifications
- BSD Pattern Generation Specifications for IEEE Std 1149.6

- Limitations
- Example Scripts for an IEEE Std 1149.6 Design

BSD Compiler Commands

The commands in BSD Compiler that support the synthesis of IEEE Std 1149.6 are as follows:

- set_bsd_configuration
- set_attribute
- set_bsd_ac_port
- define_dft_design
- set_bsd_instruction
- set_boundary_cell

DW Foundation Library to Support IEEE Std 1149.6

`dft_jtag.sldb` is the DesignWare foundation library to support IEEE Std 1149.6 logic. This is in addition to the DesignWare foundation library, `dw_foundation.sldb` which supports IEEE Std 1149.1.

set_bsd_configuration

The `-std` option of the command `set_bsd_configuration` supports the IEEE Std 1149.6 logic in BSD synthesis.

The syntax is

```
set_bsd_configuration -std { ieee1149.6_2003 }
```

Support for `ieee1149.1_1993` is enabled by the following command:

```
set_bsd_configuration -std { ieee1149.1_1993 }
```

where the default is `ieee1149.1_2001`.

set_attribute

Usage of the set_attribute command for IEEE Std 1149.6 is similar to the usage of the command for the IEEE Std 1149.1. See “Reading HDL Source Files With Differential I/O Pad Cells” on page 2-73.

set_bsd_ac_port

The `-port` option of the `set_bsd_ac_port` command captures the AC port specification. The syntax is as follows:

```
set_bsd_ac_port -port_list list_of_ac_ports
```

where `list_of_ac_ports` specifies the ports for which IEEE Std 1149.6 BSR cells need to be synthesized, as shown in Example 3-1.

Note:
This command has to be specified before the `set_boundary_cell` command for AC port.

For differential ports, you must specify only positive ports. An error is issued if the specified ports include input ports. If this command is not specified, IEEE Std 1149.6 BSR cells are not added to the design.

Example 3-1 Setting the AC Port Specification

```
## identify dot6 ports
set_bsd_ac_port -port_list { out0 out1 diff_out_pos}
```

define_dft_design

The `lp_time` and `hp_time` options of the `define_dft_design` command enable you to specify low pass and high pass time constants. The `receiver_p, receiver_n, ac_init_data_p,` and `ac_init_data_n` options of this command enable you to specify the signal types.

The syntax of this command for IEEE Std 1149.6 is as follows:

```
define_dft_design
  -type PAD
  -interface {..[receiver_p pin_name h]
                [receiver_n pin_name h]
                [ac_init_data_p pin_name h]
                [ac_init_data_n pin_name h]
                [ac_init_clk pin_name h]
                [ac_mode 0 | 1]}
  -params { ... [$lp_time$ float time_float]
                [$hp_time$ float time_float
                [$on_chip$ string true | false}
```

The following table shows the `define_dft_design` command options:

Option	Description
`lp_time`	Specifies the low pass time constant for the pad test receiver
`hp_time`	Specifies the high pass time constant for the pad test receiver
`on_chip`	Specifies that the pad test receiver had an on chip AC coupling
`receiver_p` `receiver_n`	Specify the core side pins of the positive and negative test receivers of the pad
`ac_init_data_p` `ac_init_data_n`	Specify the initialization pins of the hysteretic memories of positive and negative test receivers of the pad
`ac_init_clk`	Specifies the clock pins of hysteretic memories of positive and negative test receivers of the pad
`ac_mode`	Specifies whether the comparator is sensitive to input levels (DC mode) or sensitive to input transitions (AC mode). 1 indicates AC Mode and 0 indicates DC Mode

The values specified by `lp_time`, `hp_time`, `on_chip` are used only for BSDL generation. The value of the `time_float` option should be a positive, nonzero real number in units of nanoseconds.

For test receivers of non-differential input and bidirectional ports, the `receiver_p` and `ac_init_data_p` signal types should be used. A BC BSR cell is inserted for the `receiver_p` and the `receiver_n` pins of a pad cell.

Note:
Set the `ac_init_data_n` and `receiver_n` pins to h for active high sense pins.

The `define_dft_design` command supports user-specified BSR cell designs of the following IEEE Std 1149.6 BSR (AC) types.

- AC_1
- AC_2
- AC_7
- AC_SELX
- AC_SELU

Note:
For differential pad design, you need to always specify both the positive and negative ports for accuracy in the `preview_dft` and `insert_dft` commands and in the BSD file.

Example 3-2 shows the `define_dft_design` command with test receiver and hysteresis.

Example 3-2 Specifying a Test Receiver and Hysteresis Model

```
## INPUT DIFF_RX_HYST pad with Test Receiver and Hysteresis model
define_dft_design -design_name DIFF_RX_HYST \
-type PAD \
-interface { port t4_diffrx_in_pos h \
port t4_diffrx_in_neg l \
receiver_p t4_diffrx_out_pos h \
receiver_n t4_diffrx_out_neg h \
data_out t4_diffrx_out h \
ac_init_clk t4_init_clk h \
ac_init_data_p t4_init_data_p h \
ac_init_data_n t4_init_data_n h \
ac_mode t4_AC_Mode h }
-params {$pad_type$ string input $differential$ string true
}
```

set_bsd_instruction

The `set_bsd_instruction` command supports IEEE Std 1149.6 instructions EXTEST_PULSE and EXTEST_TRAIN with the `-time` option.

The syntax is as follows:

```
set_bsd_instruction [EXTEST_PULSE | EXTEST_TRAIN]
     -time time_float
     -clock_cycles clock_cycles
```

where `time_float` is a positive, nonzero real number in units of nanoseconds.

This option is allowed only for EXTEST_PULSE, EXTEST_TRAIN, INTEST or RUNBIST instructions. If the option is used for other instructions, an error is issued.

For the EXTEST_PULSE instruction, `-time` specifies the minimum time to remain in the run-test and idle states (real number in ns units), and the `-clock_cycles` option specifies the minimum wait time in terms of full TCK cycles. Note that you can specify either the `-time` option or the `-clock_cycles` option but not both.

For the EXTEST_TRAIN instruction, `-time` specifies the maximum time to remain before exiting the run-test and idle states (real number in ns units), and the `-clock_cycles` option specifies the minimum wait time before exiting the run-test and idle states in terms of full TCK cycles.

The `set_bsd_instruction` command considers the EXTEST_PULSE, EXTEST_TRAIN instructions as standard instructions with BOUNDARY as the TDR. Any other TDR specification for these instructions results in error. Example 3-3 shows a specification of these instructions.

Example 3-3 Specifying EXTEST_PULSE and EXTEXT_TRAIN Instructions

```
# Specify EXTEST_PULSE minimum wait duration in TCK cycles
set_bsd_instruction [list EXTEST_PULSE] -code 1001 \
         -register BOUNDARY -time 6.2


# Specify EXTEST_TRAIN minimum duration in TCK cycles, maximum time
set_bsd_instruction [list EXTEST_TRAIN] -code 1000 \
        -register BOUNDARY -clock_cycles {tck 10} -time 5.0
```

set_boundary_cell

The `set_boundary_cell` command supports the alternating current (AC) or the direct current (DC) cell specification and the BSR cell specification for test receiver pins.

The syntax is as follows:

```
set_boundary_cell
   [-class bsd]
   [-type cell_type]
   [-function ac_select ]
   [-port list_of_ac_ports]
   [-name name]
   [-share true | false]
   [-function receiver_p]
   [-function receiver_n ]
   [-port list_of_input_or_bidi_ports]
   [-design_name design_name]
```

The following table shows some of the `set_boundary_cell` command options.

Option	**Description**
`-function ac_select`	The function type ac_select specifies that the BSR cell of type cell_type is used as AC/DC selector cell for the specified AC ports. Valid BSR cell types for this function type include: AC_SELU AC_SELX NONE

Option	Description

`-function receiver_p`
`-function receiver_n`
	These function types specify that the BSR cell of type cell_type is added for test receiver pins of the specified ports.
	Valid BSR cell types for these function types include:
	BC_1
	BC_2
	BC_4
	None
	If cell_type is none, then the BSR cell is not added for the port function.

`-share`	This option allows or disallows the sharing of AC/DC selector cells. The default value of this option is `true`.
`-design_name`	Specifies the name of the design to be used for boundary-scan insertion in DFT insertion.

Note:
You always need to specify the `-function` option for the `set_boundary_cell` command.

Example 3-4 shows you how to specify the AC function for the boundary cell pins.

Example 3-4 Specifying the Test Receiver Pin AC Function for Boundary Cells

```
##bsdl all ports in ac section
set_boundary_cell -class bsd \
-type AC_SELU \
-function ac_select \
-port { out0 out1 diff_out_pos} \
-name SELU_1 \
-share true
```

IEEE Std 1149.6 Architecture

Figure 3-1 shows a typical design with various types of pads and differential pads with test receivers and drivers before IEEE Std 1149.6 logic insertion, and Figure 3-2 depicts the same design after the insertion of AC (IEEE Std 1149.6) and DC (IEEE Std 1149.1) boundary-scan cells.

Figure 3-1 Design Before Boundary Scan

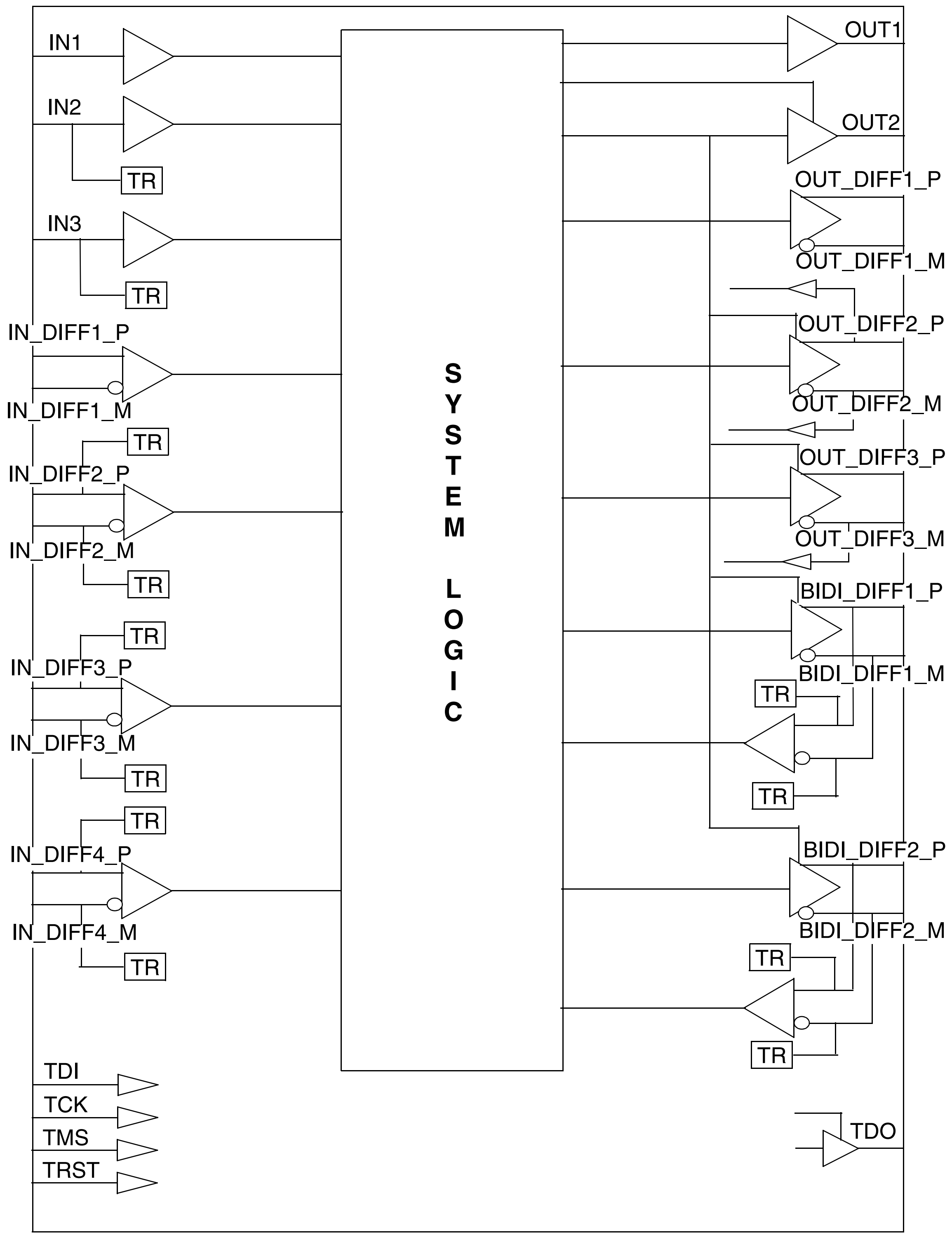


Figure 3-2 Design After Boundary Scan

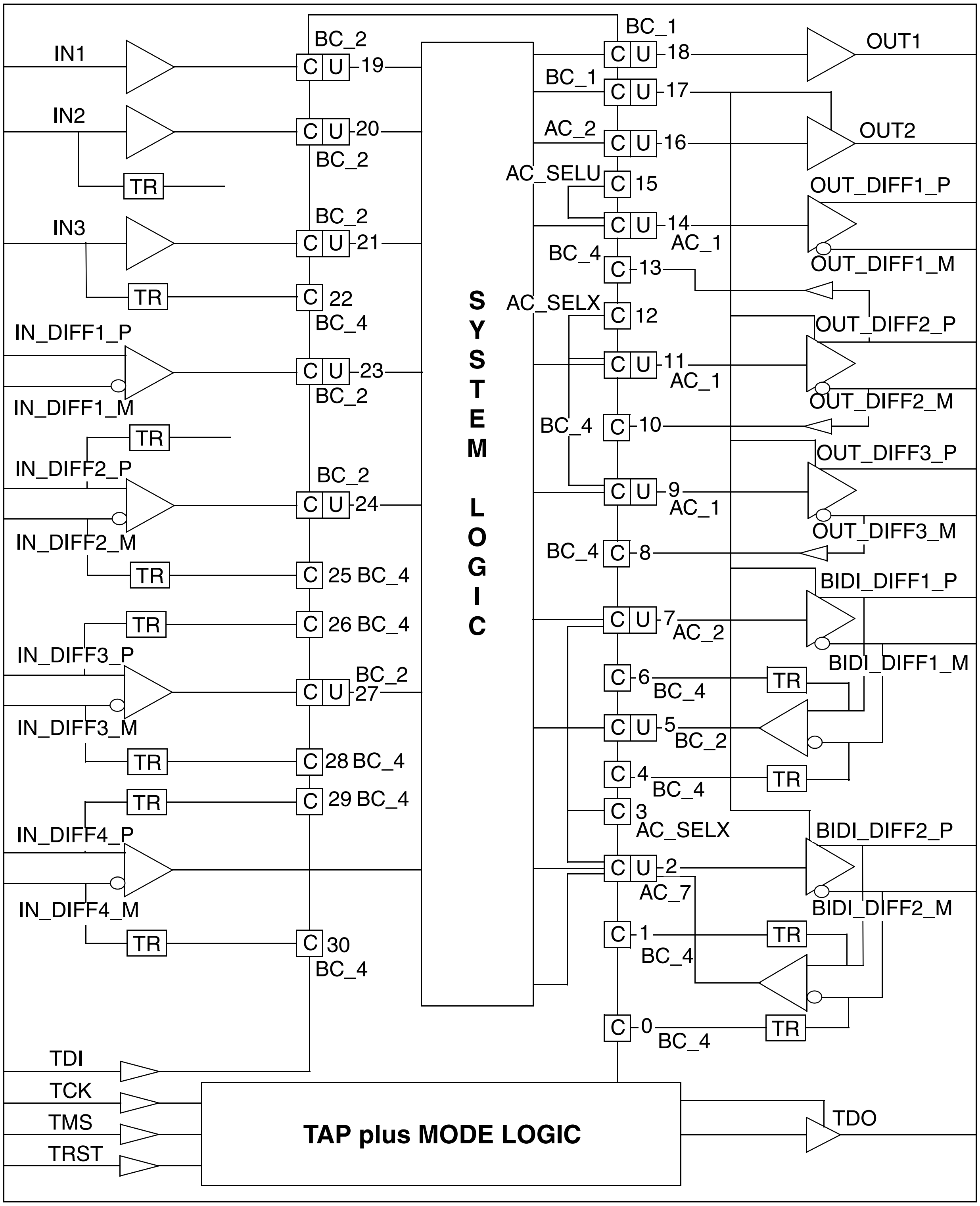
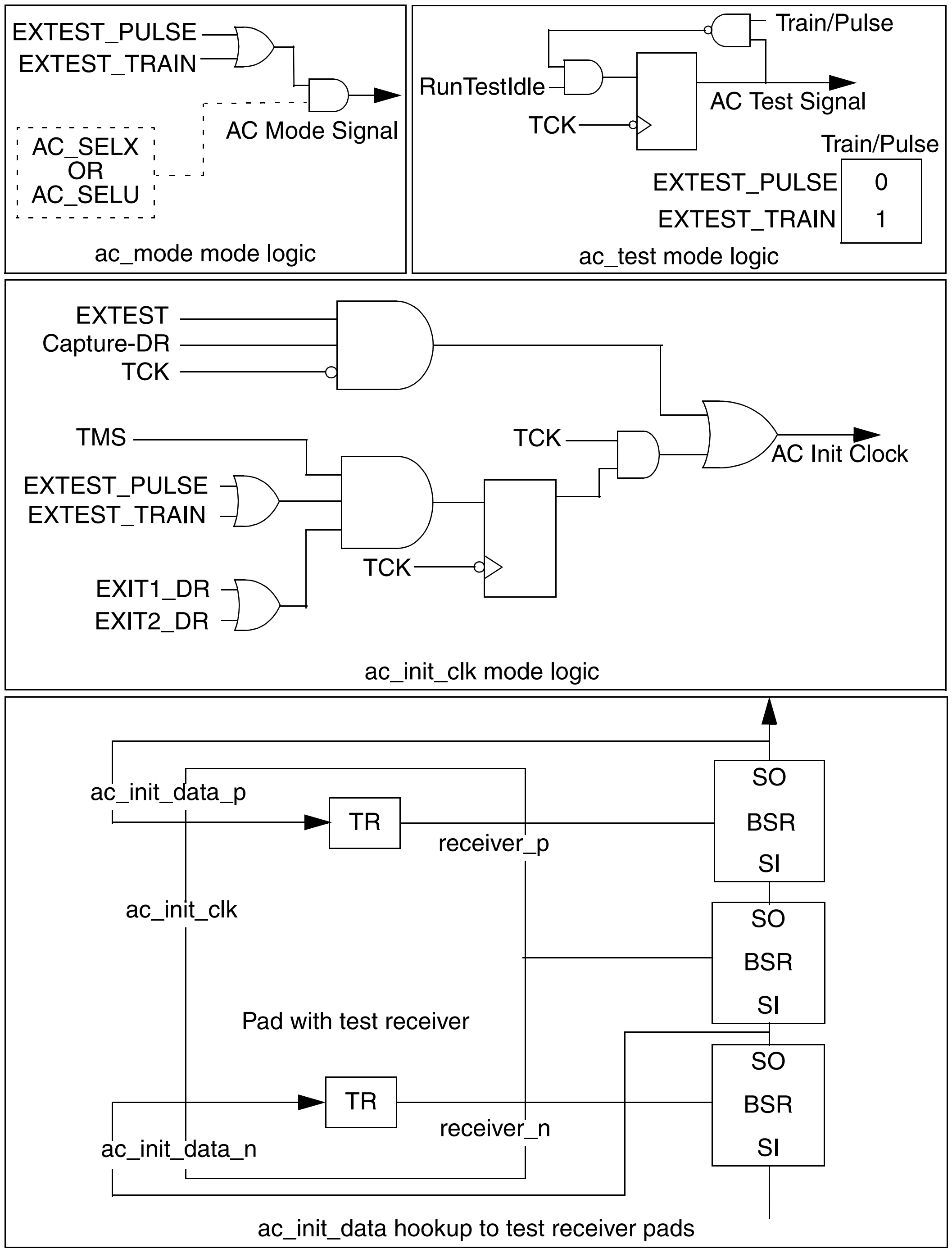


Figure 3-3 exhibits the control logic added in the mode block for the IEEE Std 1149.6 implementation.

Figure 3-3 Control Logic Added for IEEE 1149.6 Implementation



IEEE Std 1149.6 Preview Specifications

The preview specifications are listed here:

- In the absence of the `set_boundary_cell` command specification for AC ports (specified by the `set_bsd_ac_port` command), BSD Compiler adds AC_1 BSR cells for all AC ports if IEEE Std 1149.6 support is enabled. AC or DC selection cells are not added to AC_1 cells by default.
- BSD Compiler adds user-specified AC or DC BSR cells instead of DesignWare BSR cells for AC ports if they are specified.
- BSD Compiler adds user-specified AC or DC selection cells for AC ports if they are specified and shares the selection cells as specified by the user.
- BSD Compiler preview architects EXTEST_TRAIN and EXTEST_PULSE instructions if IEEE Std 1149.6 support is enabled. These instructions select BOUNDARY register as TDR.
- BSD Compiler preview or insertion adds an observe-only BSR cell for pad pins of signal type `receiver_p`, `receiver_n` in the absence of user cell specification for these pins.
- Preview report shows AC BSR cells along with DC BSR cells if they are inserted for the design. The position of AC selector (AC_SELU, AC_SELX) cells are also shown in preview report.
- Preview report shows EXTEST_PULSE, EXTEST_TRAIN instructions if they are architected for the design.

IEEE Std 1149.6 Synthesis Specifications

The specifications for IEEE Std 1149.6 synthesis are outlined below:

- BSD Compiler synthesizes the following types of BSR cells, if they are architected.
  - AC_1
  - AC_2
  - AC_7
  - AC_SELU
  - AC_SELX

- BSD Compiler synthesizes EXTEST_PULSE, EXTEST_TRAIN instructions if they are architected. These instructions use BOUNDARY register as TDR. These instructions use BSR output conditioning.
- BSD Compiler synthesizes ac_mode logic as shown in Figure 3-3 to connect ac_mode pins of AC BSR cells and test receiver pads if IEEE Std 1149.6 support is enabled.
- BSD Compiler synthesizes ac_test logic as shown in Figure 3-3 to connect ac_test pins of AC BSR cells if IEEE Std 1149.6 support is enabled.
- BSD Compiler synthesizes ac_init_clk logic as shown in Figure 3-3 to connect ac_init_clk pins of all test receiver pads if IEEE Std 1149.6 support is enabled.
- BSD Compiler hooks up ac_init_data_p, ac_init_data_n pins of a pad as shown in Figure 3-3 on page 3-11.

BSDL Generation Specifications

The section lists the specifications that are applicable to generate the IEEE 1149.6 BSDL file after BSD insertion.

- BSDL generated for designs with IEEE Std 1149.6 logic shows AC BSR cells and EXTEST_PULSE or EXTEST_TRAIN instructions if they are implemented.
- BSDL generated for designs with IEEE Std 1149.6 logic would have the following use statement.

  ```
  use STD_1149_6_2003.all
  ```

- BSDL generation adds the following attributes to BSDL for designs with IEEE Std 1149.6 logic.

  ```
  attribute AIO_COMPONENT_CONFORMANCE of <entity>: entity
  is "STD_1149_6_2003
  ```

- The following BSDL attributes are optionally generated for IEEE Std 1149.6 design AC ports. The AC or DC cell numbers are automatically computed, and `hp_time` and `lp_time` are captured from user specification. They are as follows:
- The following attribute string is added to the BSDL file for all output or bidirectional AC ports.

  ```
  attribute AIO_Pin_Behavior of <entity> : entity is
  "<ac_ports> [: AC_Select=<cell_num>];"
  ```

- The following attribute string is added to the BSDL file for input and bidirectional AC ports with test receivers that do not have low pass filter and require external AC coupling.

  ```
  attribute AIO_Pin_Behavior of <entity> : entity is
  "<ac_pins> : hp_time=<time>;"
  ```

- The following attribute string is added to the BSDL file for input and bidirectional AC ports with test receivers that have low pass filter and that are also on the chip driver.

  ```
  attribute AIO_Pin_Behavior of <entity> : entity is
  "<ac_pins> : lp_time=<time1> hp_time=<time2> On_Chip;
  ```

- The following BSDL attributes are optionally generated for EXTEST_PULSE and EXTEST_TRAIN instructions. The information shown in these attribute strings is captured from user specifications.
- The following attribute string is added to the BSDL file for EXTEST_PULSE instruction to show minimum wait time in run-test and the idle states in units of full TCK cycles of TCK:

  ```
  attribute AIO_EXTEST_Pulse_Execution of <entity> : entity
  is "Wait_Duration TCK <min_num_TCK_cycles>";
  ```

- The following attribute string is added to the BSDL file for EXTEST_PULSE instruction to show the minimum wait time in run-test and idle states in units of real time.

  ```
  attribute AIO_EXTEST_Pulse_Execution of <entity> : entity
  is "Wait_Duration <min_time>";
  ```

- The following attribute string is added to the BSDL file for EXTEST_TRAIN instruction to show minimum wait time in run-test and idle states in units of full cycles of TCK.

  ```
  attribute AIO_EXTEST_Train_Execution of <entity> : entity
  is "train <min_num_TCK_cycles>";
  ```

- The following attribute string is added to the BSDL file for EXTEST_PULSE instruction to show maximum wait time in run-test and idle states in units of real time.

  ```
  attribute AIO_EXTEST_Train_Execution of <entity> : entity
  is "train <min_num_TCK_cycles>, maximum_time <time>";
  ```

BSD Pattern Generation Specifications for IEEE Std 1149.6

BSD Compiler supports the generation of BSD Patterns or IEEE Std 1149.6. After BSD insertion, the formats supported are STIL, Verilog DPV, or WGL_Serial and Verilog Testbench. See Chapter 5, “Generating BSDL and BSD Patterns.”

Limitations

The limitations associated with IEEE Std 1149.6 are as follows:

- There is no compliance checker for validating the IEEE Std 1149.6 logic.
- There is no support for AC_8, AC_9, and AC_10 BSR cell support.

Example Scripts for an IEEE Std 1149.6 Design

Note:
Add the advanced DFT library components `dft_jtag.slb` in the dc_setup file, as follows:

```
set synthetic_library [list dw_foundation.sldb dft_jtag.sldb]
set link_library [concat $link_library $synthetic_library "*"]
```

Example 3-5 Script to Generate the IEEE Std 1149.6 BSD Logic.

```
# Read and link  the design
read_verilog bss_ut15.v current_design M1 link

# Read pin map
read_pin_map M1_15.pinmap

# Specify BSD Configuration
set_bsd_configuration -default_package M1_15 \
-std {ieee1149.6_2003 }

# Specify TAP Ports
set_dft_signal -view spec -type TCK -port TCK set_dft_signal\
-view spec -type TDI -port TDI set_dft_signal -view spec \
-type TMS -port TMS set_dft_signal -view spec -type TDO \
-port TDO set_dft_signal -view spec -type TRST -port TRST

# Specify input pad designs, used for IN2
define_dft_design -design_name in_pad1 -type PAD \
-params { $pad_type$ string input $hp_time$ float 15.0 } \
-interface { port DI h data_out DO h ac_init_clk \
ICLK h ac_mode ACM h }

# Used for IN3
define_dft_design -design_name in_pad2 -type PAD \
-params { $pad_type$ string input $hp_time$ float 15.0 \
$on_chip$ string true } -interface { port DI h data_out DO\
h ac_init_clk ICLK h receiver_p TR h ac_mode ACM h \
ac_init_data_p INIT h }
```

```
# Used for IN_DIFF1
define_dft_design -design_name diff_in_pad1 -type PAD \
-params { $pad_type$ string input $differential$ \
string true} -interface { port DP h port DM l data_out DO h }

# Used for IN_DIFF2
define_dft_design -design_name diff_in_pad2 -type PAD \
-params { $pad_type$ string input $differential$ \
string true $lp_time$ float 5.0 $hp_time$ float 15.0 } \
-interface { port DP h port DM l data_out DO h \
ac_init_clk ICLK h receiver_n TRM h ac_mode ACM h \
ac_init_data_n INITN h }

# Used for IN_DIFF3, IN_DIFF4
define_dft_design -design_name diff_in_pad3 -type PAD \
-params { $pad_type$ string input $differential$ string true\
$lp_time$ float 5.0 $hp_time$ float 15.0 $on_chip$ \
boolean true } -interface { port DP h port DM l data_out \
DO h ac_init_clk ICLK h receiver_p TRP h receiver_n TRM h \
ac_mode ACM h ac_init_data_p INIT h ac_init_data_n INITN h }

# Specify Output pad designs # Used for OUT_DIFF1
define_dft_design -design_name diff_out_pad1 -type PAD \
-params { $pad_type$ string output $differential$ \
string true } -interface { port DP h port DM l data_in DI \
h ac_mode ACM h ac_test ACT h }

# Used for OUT_DIFF2
define_dft_design -design_name diff_out_pad2 -type PAD \
-params { $pad_type string tristate_output $differential$\
string true } -interface { port DP h port DM l data_in DI\
h ac_mode ACM h ac_test ACT h observe DPI h observe DMI l }

# Used for OUT_DIFF3
define_dft_design -design_name diff_out_pad3 -type PAD \
-params { $pad_type string tristate_output $differential$ \
string true } \ -interface { port DP h port DM l \
 data_in DI h ac_mode ACM h ac_test ACT h observe DMI l }

# Specify Bidirectional pad designs # Used for BIDI_DIFF1,
BIDI_DIFF2
define_dft_design -design_name diff_out_pad3 -type PAD \
-params { $pad_type$ string bidirectional $differential$ \
string true $lp_time$ float 5.0 \
$hp_time$ float 15.0 } \ -interface { port DP h port DM l \
data_in DI h data_out DO h ac_mode ACM h \ ac_test ACT h \
receiver_p DPI h receiver_n DMI h ac_init_data_p INIT h \
ac_init_data_n INITN h }
```

```
# Specify AC ports
set_bsd_ac_port -port_list { OUT2 OUT_DIFF1 OUT_DIFF2 OUT_DIFF3 \
BIDI_DIFF1 BIDI_DIFF2 }

# Specify EXTEST_PULSE minimum wait duration in TCK cycles
set_bsd_instruction EXTEST_PULSE -time 6.2

# Specify EXTEST_TRAIN minimum duration in TCK cycles,
maximum time
set_bsd_instruction EXTEST_TRAIN -clock_cycles{ TCK 30 }\
-time 5.0

# Specify BC/AC BSR cells to be used
set_boundary_cell -type none -ports IN_DIFF4 -function input
set_boundary_cell -type AC_2 -ports OUT2 -function output
set_boundary_cell -type AC_SELU -function ac_select \
-ports OUT_DIFF1 -name SEL1
set_boundary_cell -type AC_SELX -function ac_select \
-ports { OUT_DIFF2 OUT_DIFF3 }-name SEL2
set_boundary_cell -type AC_SELX -function ac_select \
-ports { BIDI_DIFF1 BIDI_DIFF2 } \
-name SEL3
set_boundary_cell -type BC_2 -function input \
-ports BIDI_DIFF1
set_boundary_cell -type AC_2 -function output \
-ports BIDI_DIFF1
set_boundary_cell -name CNTRL1 -type BC_1 \
-ports { OUT2 OUT_DIFF2 OUT_DIFF3 BIDI_DIFF1 BIDI_DIFF2 }

preview_dft -bsd all

insert_dft

# Generate BSD netlist
change_names -hierarchy -rules verilog
write -f verilog -hier -output M1_bsd.v

# Generate BSDL
write_bsdl -output M1_bsdl

# Generate BSD Patterns
create_bsd_patterns -type all
write_test -format stil_testbench -output M1_bsd_pat_tb
write_test -format wgl_serial -output bsd_wgl_serial
write_test -format verilog -output bsd_verilog_tb
```

Example 3-6 Preview BSD Report

```
****************************************
Preview bsd report
Design : M1
****************************************

Number of TAP ports : 5

port type port name pad pin(s) package pin

TCK TCK T1/Z TDI TDI T2/Z TDO TDO
T3/A,T3/E TMS TMS T4/Z TRST TRST T5/Z

Test Logic Reset Method: Synchronous and Asynchronous(TRST)
Number of test data registers: 2

Mandatory:
Register Length


BYPASS 1 BOUNDARY 31


Instruction Register Length : 3

Instruction Encoding : default

Number of instructions : 6

Instructions that select the register 'BYPASS': BYPASS
111 Instructions that select the register 'BOUNDARY':
EXTEST 000 SAMPLE 001 PRELOAD 001
EXTEST_PULSE 010 EXTEST_TRAIN 011

Number of unused opcode(s) mapped to the BYPASS instruction:
3

Number of ports reduced or disabled: 0

Boundary Scan Register length: 31

index port pin(s) package pin function type impl ccell disval
rslt
30 IN_DIFF4_M ID4/TRM P22 observe_only BC_4 DW_BC_4 --29
IN_DIFF4_P ID4/TRP P21 observe_only BC_4 DW_BC_4 --28
IN_DIFF3_M ID3/TRM P20 observe_only BC_4 DW_BC_4 --27
IN_DIFF3_P ID3/DO P19 input BC_2 DW_BC_2 --26 IN_DIFF3_P
ID3/TRP P19 observe_only BC_4 DW_BC_4 --25 IN_DIFF2_M ID2/
TRM P18 observe_only BC_4 DW_BC_4 --24 IN_DIFF2_P ID2/DO P17
input BC_2 DW_BC_2 --23 IN_DIFF1_P ID1/DO P15 input BC_2
```

```
DW_BC_2 --22 IN3 I3/TR P14 observe_only BC_4 DW_BC_4 --21
IN3 I3/DO P14 input BC_2 DW_BC_2 --20 IN2 I2/DO P13 input
BC_2 DW_BC_2 --19 IN1 I1/DO P12 input BC_2 DW_BC_2 --18 OUT1
O1/DI P11 output2 BC_1 DW_BC_1 --17 * O2/EN -control BC_1
DW_BC_1 --16 OUT2 O2/DI P10 output3 AC_2 DFT_AC_2 17 0 Z 15
* --internal AC_SELU DFT_AC_SELU --14 OUT_DIFF1_P OD1/DI P8
output2 AC_1 DFT_AC_1 15 -13 OUT_DIFF2_P OD2/DPI P7
observe_only BC_4 DW_BC_4 --12 * --internal AC_SELX
DFT_AC_SELX --11 OUT_DIFF2_P OD2/DI P7 output3 AC_1 DFT_AC_1
17 0 Z 10 OUT_DIFF2_M OD2/DMI P6 observe_only BC_4 DW_BC_4
--9 OUT_DIFF3_P OD3/DI P5 output3 AC_1 DFT_AC_1 17 0 Z 8
OUT_DIFF3_M OD3/DMI P4 observe_only BC_4 DW_BC_4 --7
BIDI_DIFF1_P B1/DI P3 output3 AC_2 DFT_AC_2 17 0 Z 6
BIDI_DIFF1_P B1/DPI P3 observe_only BC_4 DW_BC_4 --5
BIDI_DIFF1_P B1/DO P3 input BC_2 DW_BC_2 --4 BIDI_DIFF1_M
B1/DMI P2 input BC_2 DW_BC_4 --3 * --internal AC_SELX
DFT_AC_SELX --2 BIDI_DIFF2_P B2/Z P1 bidir AC_7 DFT_AC_7 17
0 Z 1 BIDI_DIFF2_P B2/DPI P1 observe_only BC_4 DW_BC_4 --0
BIDI_DIFF2_M B2/DMI P0 observe_only BC_4 DW_BC_4 --Completed
Boundary Scan Preview
```

Example 3-7 BSDL Report Generated by Example 3-1, Setting the AC Port Specification

```
*********************************************************************
BSDL file for design M1

*********************************************************************

entity M1 is

--This section identifies the default device package selected

generic (PHYSICAL_PIN_MAP: string:= "M1_15");

This section declares all the ports in the design.

port (
TCK  : in bit;
TDI  : in bit;
TMS  : in bit;
TRST : in bit;
IN1  : in bit;
IN2  : in bit;
IN3  : in bit;
IN_DIFF1_P : in bit; IN_DIFF1_M : in bit; IN_DIFF2_P : in
bit; IN_DIFF2_M : in bit; IN_DIFF3_P : in bit; IN_DIFF3_M :
in bit; IN_DIFF4_P : in bit; IN_DIFF4_M : in bit; TDO : out
bit; OUT1 : out bit; OUT2 : out bit; OUT_DIFF1_P: out bit;
OUT_DIFF1_M: out bit; OUT_DIFF2_P: out bit;
OUT_DIFF2_P: out bit;
OUT_DIFF3_P: out bit;
OUT_DIFF3_M: out bit;
BIDI_DIFF1_P: inout bit;
BIDI_DIFF1_M: inout bit
BIDI_DIFF2_P: inout bit;
BIDI_DIFF2_M: inout bit );

use STD_1149_1_1994.all; use STD_1149_6_2003.all;

attribute COMPONENT_CONFORMANCE of M1: entity is
"STD_1149_1_1993";

attribute PIN_MAP of M1: entity is PHYSICAL_PIN_MAP;

#This section specifies the pin map for each port. This
information is --extracted from the port-to-pin map file
that was read in using the --"read_pin_map" command

constant M1_15: PIN_MAP_STRING :=

"IN1  : P12," &
"IN2  : P13," &
"IN3  : P14," &

"IN_DIFF1_P : P15," & "IN_DIFF1_N : P16," & "IN_DIFF2_P :
P17," & "IN_DIFF2_N : P18," & "IN_DIFF3_P : P19," &
"IN_DIFF3_N : P20," & "IN_DIFF4_P : P21," & "IN_DIFF4_N :
P22," & "OUT1 : P11," & "OUT2 : P10," & "OUT_DIFF1_P : P9,"
& "OUT_DIFF1_N : P8," & "OUT_DIFF2_P : P7," & "OUT_DIFF2_N
: P6," & "OUT_DIFF3_P : P5," & "OUT_DIFF3_N : P4," &
"BIDI_DIFF1_P : P3," & "BIDI_DIFF1_N : P2," & "BIDI_DIFF2_P
: P1," & "BIDI_DIFF2_N : P0," & "TCK : P25," &
"TDI : P26," &
"TMS : P27," &
"TRST : P28," &
"TDO : P29";

#This section specifies the differential IO port groupings
attribute PORT_GROUPING of M1: entity is
"Differential_Voltage ( " &
(IN_DIFF1_P, IN_DIFF1_M)," &
(IN_DIFF2_P, IN_DIFF2_M)," &
(IN_DIFF3_P, IN_DIFF3_M)," &
(IN_DIFF4_P, IN_DIFF4_M)," &
(OUT_DIFF1_P, OUT_DIFF1_M)," &
(OUT_DIFF2_P, OUT_DIFF2_M)," &
(OUT_DIFF3_P, OUT_DIFF3_M)," &
(BIDI_DIFF1_P, BIDI_DIFF1_M)," &
(BIDI_DIFF2_P, BIDI_DIFF2_M))";
```

```
#This section specifies the TAP ports. For the TAP TCK port,
the parameters in the brackets are:
First Field : Maximum TCK frequency.
Second Field: Allowable states TCK may be stopped in

attribute TAP_SCAN_CLOCK of TCK : signal is (10.0e6, BOTH);
attribute TAP_SCAN_IN of TDI : signal is true;
attribute TAP_SCAN_MODE of TMS : signal is true;
attribute TAP_SCAN_OUT of TDO : signal is true;
attribute TAP_SCAN_RESET of TRST: signal is true;

#Specifies the number of bits in the instruction register.
attribute INSTRUCTION_LENGTH of M1: entity is 2;

Specifies the boundary-scan instructions implemented in the
design and their --opcodes.

attribute INSTRUCTION_OPCODE of M1: entity is
"BYPASS (111)," &
"EXTEST (000)," &
"SAMPLE (001)," &
"PRELOAD (001)," &
"EXTEST_PULSE (010)," &
"EXTEST_TRAIN (011)";

#Specifies the bit pattern that is loaded into the
instruction register when --the TAP controller passes through
the Capture-IR state. The standard mandates --that the two
LSBs must be "01". The remaining bits are design specific

attribute INSTRUCTION_CAPTURE of M1: entity is "001";

#This section specifies the test data register placed between
TDI and TDO for --each implemented instruction.

attribute REGISTER_ACCESS of M1: entity is "BYPASS (BYPASS),"
& "BOUNDARY (EXTEST, SAMPLE, PRELOAD, EXTEST_PULSE,
EXTEST_TRAIN)";

#Specifies the length of the boundary scan register.

attribute BOUNDARY_LENGTH of M1: entity is 31;
```

```
#The following list specifies the characteristics of each
cell in the boundary --scan register from TDI to TDO. The
following is a description of the label --fields: num : Is
the cell number. cell : Is the cell type as defined by the
standard. port : Is the design port name. Control cells do
not have a port name. function: Is the function of the cell
as defined by the standard. Is one of input, output2, output3,
bidir, control or controlr. safe : Specifies the value that
the BSR cell should be loaded with for safe operation when
the software might otherwise choose a random value. ccell :
The control cell number. Specifies the control cell that
drives the output enable for this port. disval : Specifies
the value that is loaded into the control cell to disable
the output enable for the corresponding port. rslt :
Resulting state. Shows the state of the driver when it is
disabled.

attribute BOUNDARY_REGISTER of M1: entity is

num  cell  port  function safe [ccell disval rslt]
"30 "
29 "
28 "(BC_4, IN_DIFF4_M, OBSERVE_ONLY, X), " & (BC_4,
IN_DIFF4_P,   OBSERVE_ONLY,
27 "X), " & (BC_4, IN_DIFF3_M, OBSERVE_ONLY, X), " & (BC_2,
IN_DIFF3_P, INPUT, X), "
26" & (BC_4, IN_DIFF3_P, OBSERVE_ONLY, X), " &
"25   (BC_4, IN_DIFF2_M, OBSERVE_ONLY, X), " &
"24   (BC_2, IN_DIFF2_P, INPUT, X), " &
"23   (BC_2, IN_DIFF1_P, INPUT, X), " &
"22   (BC_4, IN3, OBSERVE_ONLY, X), " &
"21   (BC_2, IN3, INPUT, X), " &
"20   (BC_2, IN2, INPUT, X), " &
"19   (BC_2, IN1, INPUT, X), " &
"18   (BC_1, OUT1, OUTPUT2, X), " &
"17   (BC_1, *, CONTROL, 0), " &
"16   (AC_2, OUT2, OUTPUT3, X, 17, 0, Z), " &
"15   (AC_SELU, *, INTERNAL, 0), " &
"14   (AC_1, OUT_DIFF1_P, OUTPUT2, X), " &
"13   (BC_4, OUT_DIFF2_P, OBSERVE_ONLY, X), " &
"12   (AC_SELX, *, INTERNAL, 0), " &
"11   (AC_1, OUT_DIFF2_P, OUTPUT3, X, 17, 0, Z), " &
"10   (BC_4, OUT_DIFF2_M, OBSERVE_ONLY, X), " &
"9    (AC_1, OUT_DIFF3_P, OUTPUT3, X, 17, 0, Z), " &
"8    (BC_4, OUT_DIFF3_M, OBSERVE_ONLY, X), " &
"7    (AC_2, BIDI_DIFF1_P, OUTPUT3, X, 17, 0, Z), " &
"6    (BC_4, BIDI_DIFF1_P, OBSERVE_ONLY, X), " &
"5    (BC_2, BIDI_DIFF1_P , INPUT, X), " &
```

```
"4   (BC_4, BIDI_DIFF1_M, OBSERVE_ONLY, X), " &
"3   (AC_SELX, *, INTERNAL, 0), " &
"2   (AC_7, BIDI_DIFF2_P, BIDIR, X, 17, 0, Z), " &
"1   (BC_4, BIDI_DIFF2_P, OBSERVE_ONLY, X), " &
"0   (BC_4, BIDI_DIFF2_M, OBSERVE_OBLY, X)";

#Advanced I/O Description
attribute COMPONENT_CONFORMANCE of M1: entity is
"STD_1149_6_2003";
attribute AIO_EXTEST_Pulse_Execution of M1: entity is
"Wait_Duration 6.2e-9";

attribute AIO_EXTEST_Train_Execution of M1: entity is "train
30, maximum_time 5.0e-9";
attribute AIO_Pin_Behavior of M1 : entity is
"IN2 : HP_time=15.0e-9 ; " &
"IN3 : HP_time=15.0e-9 On_Chip ; " &
"IN_DIFF2 : LP_time=5.0e-9 HP_time=15.0e-9 ; " &
"IN_DIFF3, IN_DIFF4 : LP_time=5.0e-9 HP_time=15.0e-9 On_Chip
; " &
"OUT2 ;"&
"OUT_DIFF1 : AC_Select=15 ; " &
"OUT_DIFF2, OUT_DIFF3 : AC_Select=12 ; " &
"BIDI_DIFF1, BIDI_DIFF2 : AC_Select=7 LP_time=5.0e-9
HP_time=15.0e-9 " ;
end M1;
```

Example 3-8 Script for Design With Test Receiver and Hysteresis Model

```
set search_path [list ./ ./libs ./rtl $search_path] set
target_library [list class.db lsi_10k.db diff_io.db] set
synthetic_library [list dw_foundation.sldb dft_jtag.sldb]
set link_library [concat "*" $target_library
$synthetic_library]

#Note the new DesignWare library for AC components
dft_jtag.sldb

set verilogout_no_tri "true" read_file \
-format verilog diff_rx_bsr.v read_file \
-format verilog diff_rx_hyst.v read_file \
-format verilog tr.v read_file -f verilog tr_hyst.v read_file \
-format verilog top_hyst.v link

current_design M1 set_dont_touch U* set_dont_touch
IBUF2
```

```
#check_design
set_bsd_configuration -style synchronous \
-instruction_encoding one_hot -ir_width 4 \
-asynchronous_reset true -control_cell_max_fanout 3 \
-std { ieee1149.6_2003 } -check_pad_designs all

# Boundary-scan option enable
set_dft_configuration -bsd enable -scan disable

# Specfy TAP ports
set_dft_signal -view spec -type TDI -port tdi
set_dft_signal -view spec -type TDO -port tdo
set_dft_signal -view spec -type TCK -port tck
set_dft_signal -view spec -type TMS -port tms
set_dft_signal -view spec -type TRST -port trst_n

# Case TCK 20MHz
set test_default_period 50
set test_default_strobe 44
set test_default_bidir_delay 5
set test_default_delay 5

create_clock clk -period 100 -waveform [list 20 30]
create_clock clk1 -period 100 -waveform [list 20 30]

#differential as a clock but remember that a BC4 is added
to the output of this diff
create_clock diff_in1_pos -period 100 -waveform [list 20 30]

read_pin_map pin_map.txt
set_bsd_configuration -default_package my_package

#INPUT DIFF_RX_BSR pad with 3-BC4 embedded cells
define_dft_design -design_name DIFF_RX_BSR -type PAD \
-interface { ac_mode t_AC_Mode h \
ac_init_clk t_ac_init_clk h \
port t_diffrx_in_pos h port t_diffrx_in_neg l \
data_out t_diffrx_out h shift_dr t_shift_dr h \
capture_en t_capture_en h capture_clk t_capture_clk h \
si t_sih so t_soh} \
-params { $pad_type$ string input $differential$ \
string true $bsr_segment$ list string \
"0 BC4 t_diffrx_in_pos -X -" "1 BC4 t_diffrx_in_pos -X -" \
"2 BC4 t_diffrx_in_neg -X -" $end_list$}
```

```
#INPUT DIFF_RX_HYST
define_dft_design -design_name DIFF_RX_HYST -type PAD \
-params {$pad_type$ string input $differential$ \
string true } -interface { port t4_diffrx_in_pos h \
port t4_diffrx_in_neg l receiver_p t4_diffrx_out_pos h \
receiver_n t4_diffrx_out_neg h data_out t4_diffrx_out h \
ac_init_clk t4_init_clk h \
ac_init_data_p t4_init_data_p h \
ac_init_data_n t4_init_data_n h ac_mode t4_AC_Mode h }
#note: ac_init_data_n & receiver_n are h for active high sense

# optional instructions
set_bsd_instruction [list IDCODE] -code [list 1110] \
-capture_value {32'b00000000000010011011001001001001} \
-register DEVICE_ID

set_bsd_instruction [list USERCODE] -code [list 0001] \
-capture_value {32'b00011100111110000000010101010011} \
-register DEVICE_ID

set_bsd_instruction [list EXTEST] -code [list 0010] \
-register BOUNDARY
set_bsd_instruction [list SAMPLE] -code [list 0011] \
-register BOUNDARY
set_bsd_instruction [list PRELOAD] -code [list 0011] \
-register BOUNDARY
set_bsd_instruction [list HIGHZ] -code [list 0101] \
-register BYPASS
set_bsd_instruction [list CLAMP] -code [list 0110] \
-register BYPASS
set_bsd_instruction [list BYPASS] -code [list 1111] \
-register BYPASS
set_bsd_instruction INTEST -code [list 0111] -register BOUNDARY \
-input_clock_condition TCK -output_condition HIGHZ

set_bsd_instruction [list EXTEST_TRAIN] -code [list 1000]\
-register BOUNDARY
set_bsd_instruction [list EXTEST_PULSE] -code [list 1001]\
-register BOUNDARY
set_bsd_instruction [list INITIALIZE] -code [list 1010]\
-register BOUNDARY

#Must be spec before set_boundary_cell
set_bsd_ac_port -port_list { out0 out1 diff_out_pos}

#Ports in ac sel
set_boundary_cell -class bsd -type AC_SELU \
-function ac_select -port { out0 out1 diff_out_pos} \
-name SELU_1 -share true
```

```
preview_dft -bsd all
insert_dft

change_names -rules Verilog -hierarchy
write -format verilog -output f_bsd.v -hierarchy
write -format ddc -output f_bsd.ddc -hierarchy
write_bsdl -naming_check BSDL -output f1_bsd_d6.bsdl
```

Example 3-9 Script With set_scan_path

```
set search_path [list ./ ./libs $search_path]
set search_path [list ./ ./libs $search_path]
set synthetic_library [list dw_foundation.sldb
dft_jtag.sldb]
set link_library [concat "*" $target_library
$synthetic_library]

set verilogout_no_tri "true"
read_file -format verilog top.v
link
current_design M1
set_dont_touch U*

#check_design

# Boundary-scan option enable
set_dft_configuration -bsd enable -scan disable

# specify TAP ports
set_dft_signal -view spec -type TDI -port tdi
set_dft_signal -view spec -type TDO -port tdo
set_dft_signal -view spec -type TCK -port tck
set_dft_signal -view spec -type TMS -port tms
set_dft_signal -view spec -type TRST -port trst_n

#Case TCK
set test_default_period 50
set test_default_strobe 45
set test_default_bidir_delay 5
set test_default_delay 5

create_clock clk -period 100 -waveform [list 20 30]
create_clock clk1 -period 100 -waveform [list 20 30]

#pin map read here
read_pin_map pin_map.txt
set_bsd_configuration -default_package my_package
set_bsd_configuration -style synchronous \
-instruction_encoding one_hot -ir_width 4 \
-asynchronous_reset true -control_cell_max_fanout 5 \
-std { ieee1149.6_2003 ieee1149.1_1993} \
-check_pad_designs all

# instructions
set_bsd_instruction [list IDCODE] -code [list 0000] \
-capture_value {32'b00000000000010011011001001001001} \
-register DEVICE_ID

set_bsd_instruction [list USERCODE] -code [list 0001] \
-capture_value {32'b00011100111110000000010101010010} \
-register DEVICE_ID
set_bsd_instruction [list EXTEST] -code [list 0010] \
-register BOUNDARY
set_bsd_instruction [list HIGHZ] -code [list 0101]\
-register BYPASS set_bsd_instruction [list CLAMP] \
-code [list 0110] -register BYPASS
set_bsd_instruction [list BYPASS] -code [list 1111] \
-register BYPASS

set_bsd_instruction [list INITIALIZE] -code [list 1010] \
-register BOUNDARY ## specify user instructions
set_bsd_instruction [list UDI_1] -code [list 1100] \
-register BYPASS set_bsd_instruction [list UDI_2] \
-code [list 0111] -register BOUNDARY
#Specify private instruction
set_bsd_instruction -private [list PDI_1] -code [list 1011]\
-register BYPASS set_bsd_instruction -private [list PDI_2] \
-code [list 1101] -register BOUNDARY

set_bsd_ac_port -port_list { diff_out_pos diff_out2_pos }

set_boundary_cell -class bsd -type AC_2 \
-port { diff_out_pos } -function output

set_boundary_cell -class bsd -type BC_1 \
-port { diff_in_pos } -function input set_boundary_cell \
-class bsd -type BC_4 -port { diff_in_pos } \
-function input

set_boundary_cell -class bsd -type BC_1 -function control \
-port diff_out_pos -name CTL_1 -share false

set_boundary_cell -class bsd -type AC_SELU \
-function ac_select -port {diff_out_pos} -name SELU_1 \
-share false
```

```
set_boundary_cell -class bsd -type AC_SELX \
-function ac_select -port {diff_out2_pos} -name SELX_1

#Order bsr's
set_scan_path boundary -class bsd \
-ordered_elements [list clk clk1 diff_in_pos \
diff_out2_pos SELX_1 diff_out_pos SELU_1 CTL_1]

preview_dft -bsd all insert_dft

change_names -hier -rules verilog
write -hier -format ddc -output f_bsd.ddc
write -hier -format verilog -output f_bsd.v

set_dft_signal -view exist -type TCK -port tck \
-timing { 28 35 }
create_bsd_patterns write_test -format stil_testbench \
-output f_tb write_test -format wgl_serial -output f_wgl
write_bsdl -naming_check BSDL -output f_bsd.bsdl
```

Example 3-10 Bidirectional Ports Not Using BC_7 and BC_2 BSR Cells

```
##The set_boundary_cell -type none option is supported, but #use linkage
to
omit BSR insertion for the entire #bidirectional port.
set_boundary_cell -class bsd -type AC_1 -port { bidi } \
-function output
set_boundary_cell -class bsd -type BC_4 -port { bidi }\
-function input
set_boundary_cell -class bsd -type BC_1 \
-function control -port bidi -name CTL_1 -share false
```

4

Verifying the Boundary-Scan Design

This chapter describes the design flow for verifying a boundary-scan design after boundary-scan insertion, which includes preparing for and performing compliance checking against IEEE Std 1149.1 and then resolving compliance violations.

If you run a compliance check on a design with IEEE Std 1149.1 and IEEE Std 1149.6, the tool only checks compliance to the IEEE Std 1149.1.

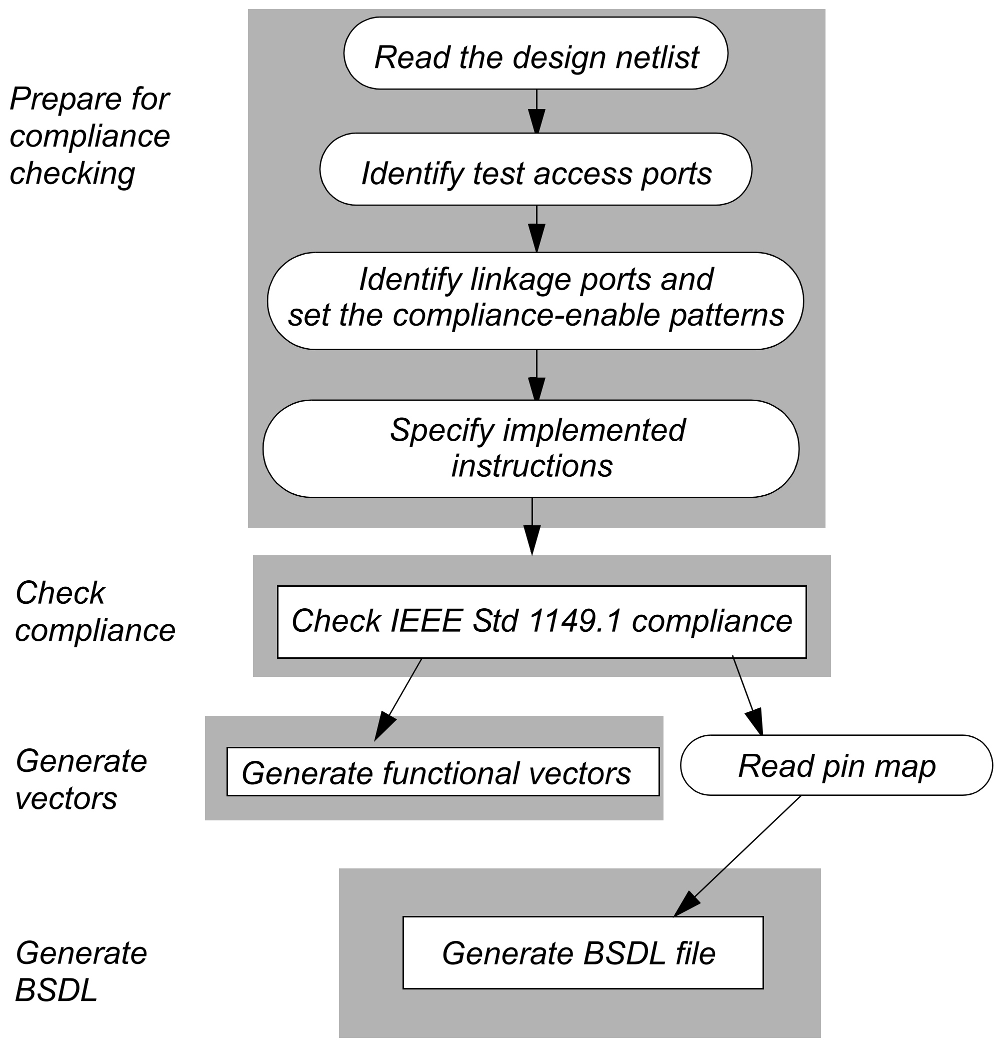
The following sections are included in this chapter:

Design Flow for Verifying a Boundary-Scan Design

Whether you insert the boundary-scan logic by using the flow described in the previous chapter or you insert it by using a third-party tool, you can verify its compliance with the rules declared in the IEEE Std 1149.1.

The flow for verifying boundary-scan logic and extracting information from it is shown in Figure 4-1.

Figure 4-1 Boundary-Scan Verification Flow


Prepare for compliance checking
Read the design netlist
Identify test access ports
Identify linkage ports and set the compliance-enable patterns
Specify implemented instructions
Check compliance
Check IEEE Std 1149.1 compliance
Generate vectors
Generate functional vectors
Read pin map
Generate BSDL
Generate BSDL file

Verify your boundary-scan design by using the following flow:

1. Read the boundary-scan design's gate-level netlist and synthesis technology library. See "Reading the Netlist for Your Design With Boundary Scan" on page 4-4.
2. Enable BSD Compiler. See "Specifying Complex Pad Designs" on page 2-75.
3. Identify your Test Access Ports. See "Verifying IEEE Std 1149.1 Test Access Ports" on page 4-4.
4. Identify linkage ports. See "Identify Linkage Ports" on page 4-6.
5. Set the compliance enable patterns. See "Compliance-Enable Patterns" on page 4-6.
6. Specify implemented instructions.
7. Run the IEEE Std 1149.1 compliance checker. See "Checking IEEE Std 1149.1 Compliance" on page 4-8.
8. Generate BSD functional and DC parametric test vectors. See "Creating Test Patterns" on page 5-2.
9. Read the port-to-pin mapping file. See "Setting Boundary-Scan Specifications" on page 2-23.
10. Generate your BSDL file. See "Generating Your BSDL File" on page 5-14.

Preparing for Compliance Checking

To prepare for compliance checking, do the following:

- Read the design netlist and technology libraries.
- Resolve all references in the design.
  - Run `link` to identify unresolved references and fix.
  - Run `check_design` to check for other multiple driven nets.
  - Run `report_cell` to identify black boxes.
  - Run `report_hierarchy` to show the hierarchy references.
- Enable BSD Compiler.
- Verify the existence of test signal ports.
- Add clock signals with `set_dft_signal`.

- Specify the linkage bits.
- Define the compliance-enable patterns, if necessary.

Reading the Netlist for Your Design With Boundary Scan

You can read your design netlist in one of the formats supported by Synopsys. For example, to read your design netlist in Verilog format, use the command:

```
read_file -format verilog top_with_bsd_scan.v
```

For more information about the `read` command, see the Design Compiler documentation.

Enabling BSD Compiler

Before you execute `preview_dft` and `check_bsd` commands, enable BSD Compiler:

```
## enable bsd and disable scan ##
set_dft_configuration -bsd enable -scan disable
```

Verifying IEEE Std 1149.1 Test Access Ports

When you read in a design, test signal port attributes might already exist. You can check for test access ports by using the command

```
report_dft_signal
```

If ports are identified by this report, you do not need to take further action to identify test access ports.

Identifying Test Access Ports

If no ports are identified by `report_dft_signal`, you must identify them by using the `set_dft_signal` command.

The syntax of the command is

```
set_dft_signal -view existing_dft -type signal_type
               -port port_list [-timing list rise fall]
```

If you have a timing requirement for the test clock (TCK), specify the `-timing` option.

The command has the options shown in Table 4-1.

Table 4-1 set_dft_signal Command Options

Option	Description
`-type` *`signal_type`*	Specifies the type of IEEE Std 1149.1 test signal you are defining. Table 4-2 shows the valid signal type of keywords.
`-port` *`port_list`*	Name of the design port with the IEEE Std 1149.1 signal type.
`-timing` *`rise fall`*	Specifies the rise and fall times for the test clock.
`-view spec`	The design must change as a result of the specification.
`-view existing_dft`	The design is not changed by the specification.

Port signals defined with the `set_dft_signal` command include those listed in Table 4-2.

Table 4-2 Port Signals Defined With the set_dft_signal Command

Signal	Description
`tck`	test clock
`tdi`	test data in
`tdo`	test data out
`tms`	test mode select
`trst`	asynchronous test reset

The `set_dft_signal` command identifies IEEE Std 1149.1 test signals by placing an attribute on the specified port. The attribute value is the same as the signal type keyword. The following example shows the `set_dft_signal` command with the `-view existing_dft` option.

```
set_dft_signal -view existing_dft -type tck -port TAP_TCK \
               -timing [list 45 55]
```

Removing Test Access Ports

Use the `remove_dft_signal` command to remove IEEE Std 1149.1 test signal attributes. The `remove_dft_signal` command has the following syntax:

```
remove_dft_signal [-view name] [-port list]
```

`port`

Name of the IEEE Std 1149.1 test signal port.

Identify Linkage Ports

Use the `set_bsd_linkage_port` command to identify the ports of the current design to be considered as linkage ports. Such ports might be analog, power or ground, or any driven by a black box cell that you do not want to consider for boundary-scan insertion and compliance checking. The linkage ports are listed as linkage bits in the BSDL. The syntax of the command is

```
set_bsd_linkage_port -port_list list_of_ports
```

For more information on setting linkage ports, see “Identifying Linkage Ports” on page 2-28.

Compliance-Enable Patterns

The `set_bsd_compliance` command is used to set logical conditions and a `compliance_enable` attribute at input ports that enable the functionality of an IEEE Std 1149.1 boundary-scan design. The logical conditions are used by `check_bsd` (the command that checks the IEEE 1149.1 rules compliance), `create_bsd_patterns`, and `write_bsdl` commands.

If your design contains one or more compliance-enable ports, you must define the compliance-enable pattern using the `set_bsd_compliance` command. The patterns generated from successive runs of `set_bsd_compliance` are appended to the previous set of patterns generated using this command.

The syntax of the command is

```
set_bsd_compliance -name pattern_name
                   -pattern [port1 bit_value1 port2 bit_value2...]
```

or

```
set_bsd_compliance -name pattern_name1
                   -pattern [port1 bit_value1]
```

```
set_bsd_compliance -name pattern_name2
                   -pattern [port2 bit_value2]
```

Note:
BSD Compiler does not support multiple compliance patterns on a pin where the same pin is changing states for an initialization sequence.

The command has the options shown in Table 4-3.

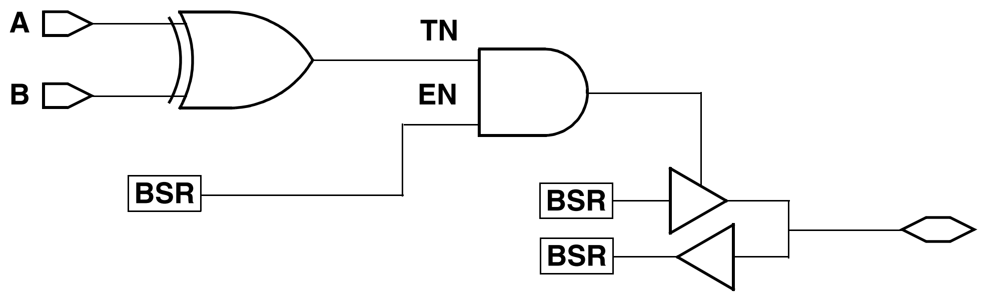
Table 4-3 set_bsd_compliance Command Options

Option	Description
`-name` *pattern_name*	Name of the pattern (not part of the bsdl file).
`-pattern` *port_bit_pairs*	A list of the signal port-binary value pairs specifying the compliance-enable pattern. The bit value states of the specified compliance-enable port that enables boundary-scan functionality. You can specify either 0 or 1 for the bit_value argument; however, BSD Compiler does not support specifications of don't care (X).

Compliance-Enable Port Example

Figure 4-2 shows an example of compliance-enable ports.

Figure 4-2 Defining Compliance-Enable Patterns Example



Ports A and B drive TN. TN must be driven to logic 1 if the logic from the boundary-scan register is to propagate through the logic.

Consequently, you must define ports A and B as compliance-enable ports, and you must drive values on each of the ports so that TN achieves the appropriate value.

You can use either of the following commands to define the compliance-enable pattern that defines port A and port B and drives TN to a logic 1.

```
set_bsd_compliance -name P1 -pattern {A 1 B 0}
```

or

```
set_bsd_compliance -name P1 -pattern {A 0 B 1}
```

or

```
set_bsd_compliance -name P1 -pattern {A 1}
set_bsd_compliance -name P2 -pattern {B 0}
```

Note:

Do not place a boundary-scan register on a compliance-enable port; doing so violates IEEE Std 1149.1. By default, the tool to all ports is defined as compliance-enable ports.

When defining the compliance-enable values in your design, do not use `set_dft_signal -port -active_state` in place of `set_bsd_compliance` to define the compliance-enable pattern.

The compliance enable signal must be a dedicated signal for JTAG compliance (`check_bsd`). It is OK to use a bidirectional pad, but for the design port definition, the compliance enable signals must be defined as input to pass `check_bsd`. This ensures that the compliance enable signal is not shared with system signals resulting in an IEEE Std 1149.1 violation, and the place and route tools can optimize this port with this definition.

Removing Definitions of Compliance-Enable Ports

Use the `reset_bsd_compliance` command to remove the compliance-enable pattern definitions. This command removes all specifications made with the `set_bsd_compliance` command.

Checking IEEE Std 1149.1 Compliance

You can use the command `check_bsd` to verify the compliance of your logic with IEEE Std 1149.1. This command analyzes your boundary-scan logic, verifying its behavior and assuring that it complies with IEEE Std 1149.1.

As `check_bsd` goes through the design, it reports on its progress through messages printed to the screen. These messages tell you what is checked and inform you of any violations that occur.

The `check_bsd` command verifies IEEE Std 1149.1 compliance of

- The TAP controller
- The instruction register

- The bypass register
- The boundary-scan register
- Other test data registers, if they exist
- Input and output ports, to see if they are controllable or observable
- Inferred instructions, to see if they select the correct register and trigger the correct behavior

At the end of the report is a summary of the kinds of violations you have, how many elements are concerned, and reference to the IEEE Std 1149.1 rules being violated.

If you use the `-verbose` option, `check_bsd` provides you with the complete list of

- TAP controller states
- Instructions
- Instruction opcodes
- Test data registers (A flush test is performed during extraction of the test data registers by shifting 0011 through the test data register.)
- BSR cells (Their structure is extracted from your design by the compliance checker.)

Note:
If you want BSD Compiler to automatically extract your design instruction's opcode, use `check_bsd -infer_instructions true`. If you want to specify the opcode manually, use `check_bsd -infer_instructions false` (the default).

Using the check_bsd Command

Check the compliance of your design against IEEE Std 1149.1 by using the `check_bsd` command. This command has the following syntax:

```
check_bsd [-verbose] [-effort low | medium | high]
          [-infer_instructions true | false]
```

Table 4-4 shows the command options.

Table 4-4 The check_bsd Command Options

Option	Description
`-verbose`	Provides more detailed message reporting for violations.
`-effort`	Specifies the amount of effort BSD Compiler exerts when extracting the boundary-scan instructions for a design that has a large instruction register.
`-infer_instructions`	Default is false, which specifies that the `check_bsd` command uses implemented instructions and their opcodes values from the boundary-scan-inserted design.

Table 4-5 Choices for the -effort Option

Option	Description
`low`	Use low effort for designs with the one-hot encoding scheme. If your design does not have one-hot encoding, use medium or high effort. BSD Compiler uses heuristics to extract boundary-scan instructions.
`medium`	This is the default effort value. BSD Compiler uses heuristics and limited sequential search to extract boundary-scan instructions.
`high`	BSD Compiler uses heuristics and full sequential search to extract boundary-scan instructions.

Run the `check_bsd -infer_instructions false` (the default) command to specify the BSD instructions manually. When you set this option to `true`, BSD Compiler extracts the instructions automatically.

Note:
To eliminate TEST-819 violations, indicate in your pad model that the input has a pull-up resistor. In the library description of your pad cell, add the following line in the declaration for the pin connected to the input port:

```
driver_type: pull_up;
```

For more information about the `driver_type` attribute see the Library Compiler user guides.

Resolving Compliance Violations

After you check your design for IEEE Std 1149.1 compliance and the results show compliance violations, you might need to troubleshoot and debug your design. To successfully troubleshoot and debug your design in BSD Compiler, you should have experience with Design Compiler and knowledge of logic design. See also the *BSD Compiler Reference Manual* for further information on these violations.

The process for troubleshooting and debugging your design is described in the following sections:

- Downgrading Errors to Warning Status
- Preparing to Debug Your Design
- Troubleshooting Problems in Your Design
- Debugging Your Source File

Downgrading Errors To Warning Status

Certain errors found during check_bsd prevent BSDL and pattern generation. For debugging purposes, you can change the severity status of these errors so that BSD Compiler completes the verification flow.

To downgrade an error message, use the following command:

```
set_message_severity -names {message_id_list}
                            (e)rror | (w)arning
```

The following example downgrades the severity of TEST-893 and TEST-813 from error to warning:

```
set_message_severity -names {TEST-893 TEST-813} w
```

To upgrade back to error status, specify:

```
set_message_severity -names {TEST-893 TEST-813} e
```

In certain situations, the `check_bsd` command generates TEST-813 or TEST-816 error messages because the tool is unable to infer the correct set of TAP state elements. Use the command `define_dft_design -type tap` to specify the set of TAP state elements to be used by `check_bsd`.

Use the `set_boundary_cell_io` command as follows to guide BSD Compiler when the TEST-893, TEST-894, or TEST-891 error occurs:

```
set_boundary_cell_io -type boundary
                     -access [in_pi | in_po | out_pi | out_po]
                     -cell bsr_cell_shift_flop
```

The following table shows the `set_boundary_cell_io` command options.

Option	Description
`-type boundary`	boundary-scan register chain
`in_pi`	Specify the port associated with the input PI BSR cell
`in_po`	Specify the PO path of the BSR cell
`out_pi`	Specify the PI path of the BSR cell for OUTPUT or CONTROL BSR cell
`out_po`	Specify the port associated with the BSR cell PO for OUTPUT or CONTROL BSR cell

For example:

```
set_boundary_cell_io -type boundary \
     -access {out_pi U11/U1/Z out_po q2scan/inst1/inst6/U1/Z} \
     -cell q2bscan/inst1/inst4/q_reg
```

You can change the severity status only for the messages listed in Table 4-6. Do not attempt to downgrade or upgrade any other messages; the following message appears if you do:

```
message_name cannot be down/up gradable
```

Use the `set_tap_elements` command as follows to when the TEST-816 and TEST-813 error occurs:

```
set_tap_elements -state_cells {list_of_tap_state_elements 1 ...2 ...3
                               ...4 n_tap_state_element}
```

Example 4-1 shows you how a TAP FSM designed with four registers is specified.

Example 4-1 A TAP Finite-State-Machine (FSM) Designed With Four Registers

```
set_tap_elements -state_cells { tap/simple/tap_state_reg[0],
tap/simple/tap_state_reg[1],
tap/simple/tap_state_reg[2],
tap/simple/tap_state_reg[3]}
```

Table 4-6 Violations That Support Severity Changes

Message	Type	Implication
TEST-813 - TAP controller state flops have been found, which is an insufficient number of state flops.	E	Specify the TAP states with set_tap_elements.
TEST-816 - TRST does not reset TAP controller asynchronously. TRST specification ignored.	E	Specify the TAP states with set_tap_elements. Use synchronous reset for all purposes.
TEST-828 - The capture value of the least significant bits in the instruction register is %s%s. It must be the fixed pattern "01".	E	None.
TEST-835 - The capture value of the least significant bit in the Device Identification Register is logic zero. It must be logic 1.	E	None.
TEST-843 - Logic cannot exist between boundary scan cell %s and design port %s.	E	The controlling/disabling value of cells for which TEST-843 occurred might be incorrect in the BSDL file. An annotation appears in the BSDL file when TEST-843 downgraded.
TEST-859 - Unable to find the Boundary Scan Register Update flops on falling edge of TCK.	E	None.
TEST-877 - The output pin of the boundary scan register cell %s is not being driven by the input pin during the instruction %s with opcode %s.	W	Only applicable to BSR cells with PI and PO (BC1 through BC7).
TEST-891 - Not able to locate the parallel output for the boundary scan register cell %s.	E	Returns the BSR cell type (output or control). This error only occurs with input and control BSR cells.
TEST-893 and TEST-894 - Not able to locate the parallel input for the boundary scan register cell %s.	W	This message occurs while checking PIs for output and controlling BSR cells. BSD Compiler returns TEST-894 for only output BSR cells and TEST-893 for only controlling BSR cells.

Preparing to Debug Your Design

When you are debugging your design, you should constrain the runtime to no longer than 5 minutes. The test case you use to debug should include at a minimum:

- Netlist
- I/O pad cells for input, output, and bidirectional ports
- Technology libraries
- Run script file run.scr

If your test case is large (requiring more than 5 minutes to run), use Design Compiler or Design Vision to reduce it. To reduce your design, you can do the following:

1. Identify the core logic not used by BSD Compiler.
2. Group all the core logic.
   ```
   group {list_of_cells} -design_name CORE -cell
   ```
3. Set the current design to CORE.
   ```
   current_design CORE
   ```
4. Remove the CORE design logic using
   ```
   remove_cell -all
   remove_net -all
   ```
5. Set the current design to TOP.
   ```
   current_design TOP
   ```
6. Save your design and run `run.scr`.

When using the BSD Compiler design flow, the easiest way to reduce your test case is to select a one-port-per-pad reference in your design for BSR insertion and assign a linkage bit to the other ports. If the core logic is too big, then remove them as shown in the previous steps.

Troubleshooting Problems in Your Design

You can gather information about problems in your design by generating reports using the `check_bsd` and `check_design` commands before setting boundary-scan design specifications. The reports generated using `preview_dft, insert_dft`, and `check_bsd -verbose` provide information critical to locating compliance violations in your design.

Write your netlist in Verilog format to assist in tracing problems, or use Design Vision. To write out your netlist in Verilog format, use the following command:

```
change_names -rules verilog -hierarchy
write -hier -format verilog -output design.v
```

Troubleshooting Using preview_dft

You can use `preview_dft` to gather information about the violations in your design and issues with pad validation. Error messages are issued that describe boundary-scan violations. A warning message generated using `preview_dft` is shown in Example 4-2.

Example 4-2 Warning Message From preview_dft

```
Warning: Inferring default device package ‘my_pack’. (TEST-601)

index  port   pin(s)  package  function   type   impl     ccell
disval rslt
                       pin
-----  ----   ------  -------  --------   ----    ----    -----   ----  ----
6      clk    U10/Z   -        clock       BC4    DW_BC4 -        -     -
5      en     U12/Z   -        input       BC2    DW_BC2 -        -     -
4      in0    U100/Z  -      observe_only BC4    DW_BC4 -        -     -
3      *      U200/E’ -        control     BC2    DW_BC2 -        -     -
2      out0   U200/A  -        output3     BC1    DW_BC1 3        1     Z
1      *      U300/E’ -        control     BC2    DW_BC2 -        -     -
0      out1   U300/A  -        output3     BC1    DW_BC1 1        1     d
```

Troubleshooting Using insert_dft

All pad validation issues must be well understood and resolved before moving on to the `insert_dft` command.

Use the following guidelines for issues with pads failing `preview_dft` validation:

- Review your pad reference data sheet.
- Review the liberty pad cell or soft macro pad cell for syntax issues.
- Review the constraints applied in the netlist to the instance of the pad reference.
- Review the way the pad functionality is reduced from bidirectional to INPUT or OUTPUT.
- Review the way extra enables and data pins are constraints blocking the pad validation.
- Review the pad design command specifications for correct BSR assignments.
- Review the active state for pad design specifications and any instruction enables.

A warning message generated from `insert_dft` is shown in Example 4-3.

Example 4-3 Warning Message From insert_dft

```
Warning: Enable pin(s) attached to the tristate port my_port
drive the port to high impedance value always. The port is ignored
for boundary scan insertion. (TEST-432)
```

Troubleshooting Using check_bsd

You can use `check_bsd` to gather information about problems in your design. BSD Compiler generates messages as it runs `check_bsd`, displaying information on problems in your design. BSD Compiler is robust enough to accept user guidance in the event of `check_bsd` violations; here are some of the commands for user guidance:

```
set_boundary_cell_io

set_tap_elements

set_message_severity
```

A warning message generated from `check_bsd` is shown in Example 4-4.

Example 4-4 Warning Message From check_bsd

```
Warning: A boundary scan register cell is missing on design
OUTPUT
port my_port. (TEST-838a)

Information: This problem occurred because pin IO of the cell
io_pads/my_port_buf is driven by a weak 'z' signal. (TEST-
905)
```

If you insert BSD manually, the `check_bsd` command requires an EXTEST binary code. If you don't specify an EXTEST code, the `check_bsd` command issues the following error message:

```
Error: EXTEST opcode not specified...
```

EXTEST opcode is not mandatory if the `set_bsd_configuration` option `-std` is set to IEEE Std 1149.1_1993

Debugging Your Source File

BSD errors in your design can include the following:

- The TAP state machine is not extracted.
- The reset is missing or incorrect.
- The bypass register is not found.

- There is a missing pullup.
- There is a missing BSR on a port.
- Pad validation fails.
- The pad is a non-functional black box.

After you know what the problems are in your design, find the location of the problem in your source file and fix it. First locate the functional pad and pin attributes.

Here are some suggestions on how to debug pad issues:

- For .lib format without pad attributes, see "Reading the HDL Source Files" on page 2-72.
- The .lib model might be incorrect or incomplete. See the Library Compiler Documentation.
- Design Compiler optimized the pads. Use `set_dont_touch` for all .lib pad cells before the `insert_dft` command.
- The pad pin functions are not compiled by Design Compiler/Library Compiler and the pad becomes a black box.
- The pad might have hierarchy that is not mapped to gates.
- If the .lib cell does not exist in the read libraries, use the following commands in dc_shell:

  ```
  report_cell <instance>
  report_attribute find (lib_pin, '<library>/<reference>/*')
  ```
- If it is a black box, then turn off the pad validation for insertion. (But remember that it must be functional for verification.)
- IDDQ structures and PU/PD are not constrained for JTAG compliance: the test MUX enables are not in the correct mode, and the IDDQ structures and PU/PD are not controlled by the TAP.
- The TAP port pads are linkage bit.
- If you are using soft macro cells with the pad design command, make sure they are JTAG compliant as well.
- TAP ports pads must be dedicated ports and not shared.

For the verification flow, most of the these guidelines apply, but all pads must be functional; black boxes and unmapped logic are not allowed.

There are no special checks for pads in IEEE Std1149.1 and IEEE Std 1149.6 applications.

If the design has IEEE Std 1149.6 cells, the pad are validated during an IEEE Std 1149.1 compliance check. However, the IEEE Std 1149.6 cells are checked only for IEEE Std 1149.1 compliance.

Note:
bsr-segments that are embedded BSR in pads are validated only during an IEEE Std 1149.1 compliance check.

Then find any problematic pins or buffer I/O pads in the Verilog netlist.

After you locate and debug the problems found in your design, save the source file.

Using the set_test_assume for preview_dft, insert_dft, and check_bsd Commands

You can use the `set_test_assume` command to assign a known logic state for pad validation during `preview_dft`, `insert_dft`, and `check_bsd`. Also, you can use the command to add values to the user-defined-test-data-register (UTDR) shift enables or MUX enables for UTDR validation during `check_bsd` internal simulation. The command accepts both driver pins and load pins.

The command syntax is `set_test_assume value pin_list`.

The value argument specifies the assumed 1 or 0 value on the specified pins, and the pin_list argument specifies the pin names of the pad enables, net enables, and MUX enables, including the UTDR registers.

The hierarchical path to the pin should be specified for pins in subblocks of the current design.

The `check_bsd` command takes into account the conditions you define with the `set_test_assume` command.

The `set_test_assume` command has no impact on BSD patterns or BSDL files. The command is ignored during BSD pattern generation or BSDL file generation.

Using the `set_test_assume` command allows you to apply alternative values to a design net, which can aid in passing a compliance check or pad validation. However, this might then require postprocessing of the netlist, using the correct values in the `set_test_assume` command for the final netlist checks with `check_bsd` simulation and pad validation.

Using the dont_touch Attribute With the insert_dft Command

As noted previously, you can use the `set_dont_touch` command to prevent Design Compiler from optimizing the pads when you run the `insert_dft` command. However, during JTAG insertion, the tool changes the design by adding the JTAG logic and modifying the existing functional connections as needed.

If you have set a `dont_touch` attribute on the hierarchy or a net that requires modification during JTAG insertion, the tool carries out the modification of the `dont_touch` logic and issues a warning (TEST-1840). For example, if a new port must be added, the tool does so and issues the warning. Similarly, a warning is issued for a modified `dont_touch` hierarchy netlist or net.

5

Generating BSDL and BSD Patterns

This chapter describes how to create test patterns in various formats for use in simulation. It also provides information on BSDL file generation and using BSD patterns with TetraMAX and VCS.

Generating BSD patterns and BSDL files for IEEE Std 1149.1 and the IEEE Std 1149.6 uses the same user interface (UI) or commands as described in this chapter.

The BSD patterns can be written after `insert_dft` for both IEEE Std 1149.1 and IEEE Std 1149.6 and after `check_bsd` for IEEE Std 1149.1.

This chapter includes the following sections.

Creating Test Patterns

You can use BSD Compiler to generate patterns to test your boundary-scan logic. By simulating these vectors through your boundary-scan logic, you can achieve significant fault coverage of your boundary-scan design. The vectors generated are typically functional vectors and are based on the structure of your boundary-scan logic. The vectors also allow leakage and Design Compiler parametric testing, enabling you to characterize your circuit's ports.

Generating Test Vectors

Create functional test patterns for your boundary-scan design by using the `create_bsd_patterns` and then the `write_test` commands. The tool supports STIL, Verilog-DPV testbench, PLI, Verilog testbench, and wgl_serial. The default is STIL.

First, run the `create_bsd_patterns` command, which generates the following:

- Functional test vectors to test the synchronous and asynchronous TAP reset
- Functional test vectors to test the TAP controller finite state machine
- Functional test vectors to test the implemented boundary-scan instructions and their associated test data registers
- Functional test vectors to test the boundary-scan register
- Functional test vectors to test for I/O dc-parametric and leakage

The syntax for `create_bsd_patterns` is

```
create_bsd_patterns
         [-output test_program_name]
         [-effort low | medium | high]
         [-type all | functional | dc_parametric | tap_controller
            | reset | tdr | bsr | leakage | ac_input_pulse
            | ac_input_train | ac_output_pulse | ac_output_train]
```

The command has the options shown in Table 5-1 on page 5-3.

Table 5-1 The create_bsd_patterns Command Options

Option	Description
`-output` *`test_program_name`*	Specifies the name of the output test program, `create_bsd_patterns` writes a temporary file to memory for further processing the pattern types.
`-effort`	Controls the effort used to search for implemented instructions. The time taken increases exponentially for a sequential search on instruction registers (IRs) of length 8 or more.
`-type`	Generates vectors to test the various aspects of the design's boundary scan. This option allows you to generate vectors to test bsr, TAP, tdr, and reset mechanisms of the boundary scan separately. This option also allows generation of leakage vectors and all functional vectors (bsr, TAP, tdr, and reset vectors). This option allows the generation of vectors to test AC receivers and transmitters.

Table 5-2 Choices for the -effort Option

Option	Description
`low`	BSD Compiler uses heuristic methods to extract boundary-scan instructions.
`medium`	This is the default effort value. BSD Compiler uses heuristics then random opcode generation to extract boundary-scan instructions.
`high`	BSD Compiler uses heuristic methods and full sequential search to extract boundary-scan instructions.

Table 5-3 Choices for the -type Option

Option	Description
`all`	Generates all the vectors.
`functional`	Generates the following vectors: tap_controller, reset, bsr, tdr, ac_input_pulse, ac_input_train, ac_output_pulse, ac_output_train.
`dc_parametric`	Generates vectors to test I/O voltage and current levels; BSR and leakage patterns are merged for dc_parametric patterns.
`leakage`	Generates vectors to test I/O leakage.

Table 5-3 Choices for the -type Option (Continued)

Option	Description
`tap_controller`	Generates vectors to test the TAP controller.
`tdr`	Generates vectors to test the test data registers, if they exist.
`bsr`	Generates vectors to test the boundary-scan registers.
`reset`	Generates vectors to test the reset behavior in the TAP controller.
`ac_input_pulse`	Generates vectors for AC input tests with EXTEST_PULSE instruction. These vectors test the preset/transition/single ended/DC behaviors of AC receivers and the DC behavior of the AC instruction.
`ac_input_train`	Generates vectors for AC input tests with EXTEST_TRAIN instruction. These vectors test the preset/transition/single ended/DC behaviors of AC receivers and the DC behavior of the AC instruction.
`ac_output_pulse`	Generates vectors for AC output tests with EXTEST_PULSE instruction. These vectors test the transition behavior of AC drivers with the AC instruction. The vectors also test AC/DC selection cells.
`ac_output_train`	Generates vectors for AC output tests with EXTEST_TRAIN instruction. These vectors test the transition behavior of AC drivers with the AC instruction. The vectors also test AC/DC selection cells.

The combination of both BSR and leakage patterns allows you to check for voltage levels and current using SAMPLE, PRELOAD, and EXTEST instructions. Vectors of the type `bsr` allow the checking of DC design characteristics such as input threshold voltage Vil/Vih or current Iil/Iih and the output threshold voltage Vol/Voh or current Iol/Ioh. Use the `bsd_max_in_switching_limit` and `bsd_max_out_switching_limit` variables to control how many inputs and outputs can be allowed to switch simultaneously. See the BSD Compiler Reference Manual for additional information on DC parametrics.

To generate vectors that test for both voltage and leakage, use the following command.

```
create_bsd_patterns -type dc_parametric
```

For more information on setting input/output switching limits, see the BSD Compiler Reference Manual.

The `create_bsd_patterns` command performs the following tasks:

- Invokes the compliance checker if necessary
- Generates functional test patterns to test the synchronous and asynchronous TAP reset

- Generates functional test patterns to test the TAP controller finite state machine
- Generates functional test patterns to test the implemented boundary-scan instructions and their associated test data registers
- Generates functional test patterns to test the boundary-scan register

These patterns are written into memory in the Synopsys pattern database (.ctldb) format.

Note:
If a boundary-scan design is non-compliant, the `create_bsd_patterns` command does not generate any functional vectors. You can downgrade some errors to warnings and then write the patterns or use the user guidance. See section "Downgrading Errors To Warning Status" on page 4-11 for more information.

Example 5-1 shows typical output from the `create_bsd_patterns` command.

Example 5-1 create_bsd_patterns Command Output

```
.....Generating vectors to test the asynchronous test logic reset
.....Generating vectors to test the synchronizing sequence of 5 1's on
tms
.....Generating vectors to test the TAP FSM
.....Generating vectors to test boundary scan instructions
.......Generating vectors to test the 'BYPASS' instruction.
.......Generating vectors to test the 'EXTEST' instruction.
.......Generating vectors to test the 'SAMPLE' instruction.
.......Generating vectors to test the 'PRELOAD' instruction.
.......Generating vectors to test the 'EXTEST_PULSE' instruction.
.......Generating vectors to test the 'EXTEST_TRAIN' instruction.
.....Generating vectors to test the boundary scan register
.....Generating vectors to perform leakage test.
.....Generating vectors for ac output tests with EXTEST_PULSE
instruction.
  .....Generating vectors to test the transition behavior of the
instruction.
.....Generating vectors for ac output tests with EXTEST_TRAIN
instruction.
   .....Generating vectors to test the transition behavior of the
instruction.
   .....Generating vectors to test the behavior of ac/dc selections
cells.
.....Generating vectors for ac input tests with EXTEST_PULSE instruction.
   .....Generating vectors to test the preset behavior of ac receivers.
   .....Generating vectors to test the edge sensitive behavior of ac
receivers.
   .....Generating vectors to test the single ended behavior of ac
receivers.
   .....Generating vectors to test dc behavior of the instruction.
.....Generating vectors for ac input tests with EXTEST_TRAIN instruction.
   .....Generating vectors to test the preset behavior of ac receivers.
   .....Generating vectors to test the edge sensitive behavior of ac
receivers.
   .....Generating vectors to test the single ended behavior of ac
receivers.
   .....Generating vectors to test the level sensitive behavior of ac
receivers.
   .....Generating vectors to test dc behavior of the instruction.
```

If you want to specify the file name for your BSD pattern, use the following command.

```
create_bsd_patterns -output my_patterns -type all
```

This creates a file called my_patterns.stil.

Writing STIL and Other Patterns

Use the `write_test` command to convert the test vectors generated in memory by `create_bsd_patterns` to the pattern format you want.

The syntax for `write_test` is

```
write_test [-output output_vector_file_name]
           [-format stil | stil_testbench | wgl_serial | verilog]
```

Table 5-4 provides a description of the available command options.

Table 5-4 The write_test Command Options

Option	Description
`-output` *`output_vector_file_name`*	Specifies as output the name of the path and base file name for the test program files. By default, test program files are written in the current directory, using the current design name as the base file name. •
`-format stil | stil_testbench | wgl_serial | verilog`	Selects the format for the output test program. Valid values for test_program_format are: • Specifying stil generates STIL patterns. • Specifying stil_testbench generates STIL patterns and a Verilog DPV testbench that uses the generated STIL patterns. • Specifying wgl_serial generates wgl_serial patterns. • Specifying verilog generates verilog patterns.

Example 5-2 shows typical output from the `write_test` command.

Example 5-2 write_test Command Output

```
write_test -format stil -output top_stil_tb
...Getting test program from design.
...Writing test patterns to file top_stil_tb.stil.
1
write_test -format stil_testbench -output top_stiltb_tb
...Getting test program from design.
...Writing test patterns to file top_stiltb_tb.stil.
Generating Test Bench ...
1
write_test  -f verilog -output f_verilog_tb
...Getting test program from design.
...Writing native Verilog test bench to file f_verilog_tb.v.
1
write_test -format wgl_serial -output top_wgl_tb
...Getting test program from design.
...Writing test patterns to file top_wgl_tb.wgl.

/remote/release/synthesis/A-2007.12/sparcOS5/syn/ltran/stil2wgl:
processing the unnamed
PatternExec block
//     STIL2WGL Version  A-2007.12-LS
//     Copyright (c) 2002-2006 by Synopsys, Inc.
//              ALL RIGHTS RESERVED
//
STIL-parse(top_wgl_tb.wgl.stil): ... STIL version 1.0 ( Design P2001.01)
...
STIL-parse(top_wgl_tb.wgl.stil): ... Building test model ...
STIL-parse(top_wgl_tb.wgl.stil): ... Signals ...
STIL-parse(top_wgl_tb.wgl.stil): ... SignalGroups  ...
STIL-parse(top_wgl_tb.wgl.stil): ... Timing  ...
STIL-parse(top_wgl_tb.wgl.stil): ... PatternBurst "_BSD_burst_" ...
STIL-parse(top_wgl_tb.wgl.stil): ... PatternExec  ...
STIL-parse(top_wgl_tb.wgl.stil): ... Pattern block "_BSD_block_" ...
stil2wgl: End of STIL data; WGL generation complete
1
```

STIL is the default format, but you can also write out patterns in the stil_testbench and wgl_serial formats by using the `-format` option to the `write_test` command. For example, to write out the wgl_serial file, top_wgl_tb, specify the following:

```
write_test -format wgl_serial -output top_wgl_tb
```

By default, this command also generates the STIL file, TOP.stil.

Example 5-3 shows a script for writing out your boundary-scan test vectors to STIL.

Example 5-3 Generating STIL Using BSD Compiler

```
# setting up
set search_path [concat ./ $search_path]
set target_library "asic_vendor.ddc"
set synthetic_library {dw_foundation.sldb}
set link_library [concat "*" $target_library $synthetic_library]

# read the boundary-scan design
read_file -format ddc TOP.ddc
link

# by default create test patterns generates STIL patterns
create_bsd_patterns

# write test vectors from ctldb
# write_test -format stil

# To write out Verilog DPV testbench from ctldb
write_test -format stil_testbench

quit
```

Writing the Verilog Testbench

You can specify the name of your Verilog testbench for pattern simulation by using the following command:

```
write_test -format verilog -output my_verilog_tb
```

This command creates the file, `my_verilog_tb.v`.

BSD Compiler can generate the Verilog testbench after the `insert_dft` command and after the `check_bsd` command using the same command:

```
insert_dft
create_bsd_patterns
write_test -format verilog -output top_verilog_tb
```

The Verilog Testbench can be split in vectors type already supported by the `create_bsd_patterns` command.

Here is an example on generating Verilog testbench for the BSR & TDR patterns; the default for the tool is `-type all`.

```
insert_dft
create_bsd_patterns -type bsr
write_test -format verilog -output top_verilog_tb_bsr
create_bsd_patterns -type tdr
write_test -format verilog -output top_verilog_tb_tdr
```

It can also be done after the `check_bsd` command using the same commands:

```
check_bsd -v
create_bsd_patterns -type bsr
write_test -format verilog -output top_verilog_tb_bsr
create_bsd_patterns -type tdr
write_test -format verilog -output top_verilog_tb_tdr
```

Preparing for BSDL Generation

Before you proceed to BSDL generation, you must include certain information in your boundary-scan design, which might already exist in your design database. Before generating your BSDL file, verify this information; if it does not exist, you must provide it.

You can generate the following reports to verify boundary-scan design information:

- BSD configuration
- BSD specification
- Pin mapping and package type

Reading the Port-to-Pin Mapping File

Your top-level design should include a mapping of logical ports to physical package pins.

To read in the pin mapping file, use the following command:

```
read_pin_map file_name
```

The command has the option shown in Table 5-5.

Table 5-5 read_pin_map Command Argument

Argument	Description
file_name	The name of the port-to-pin mapping file.

You might want to choose one package from the various packages for your design. You can optimize port ordering if you specify a pin map file before synthesis. The ordering can’t be changed after synthesis. For example, if you have a ceramic package, you might use the command

```
read_pin_map ceramic_port_to_pin_map
```

The following sections discuss the way BSD Compiler uses the port-to-pin mapping file and explain how to create one. For more information on linkage, bus, and non-bused ports, see the BSD Compiler Reference Manual.

You can use the `read_pin_map` command to read in multiple packages. See section "Generating BSDL and BSD Patterns for Multiple Packages" on page 5-17 for more information.

Purpose of the Port-to-Pin Mapping File

By default, BSD Compiler routes boundary-scan register cells based on the alphanumeric order of their associated ports. This is generally not the optimum ordering for wiring, timing, or placement.

The port-to-pin mapping file allows you to specify the correspondence between logical ports and physical package pins. In the pin mapping file, you specify the physical package you intend to use, and then map logical ports on the top level of the design to the physical pins on the package.

The port-to-pin mapping file serves two purposes:

1. Provides BSD Compiler with information regarding the relative order of the ports for boundary-scan synthesis

   Note:
   There is another way to order the boundary-scan register cells, which is described in "Custom Boundary-Scan Design" on page A-1"

2. Provides BSD Compiler with the naming relationship between ports and pins for BSDL file generation.

Defining Relative Port Positions

When deciding how ports are to be associated with pins, consider the relative position of the ports within the logic. The boundary-scan cell positions in the design's top-level logic should determine the port-to-pin mapping. For example, if `RESETN` is next to `EN` in the top-level logic, map the pins as shown here, to facilitate timing and optimize wiring:

```
PORT = EN, PIN = P5;
PORT = RESETN, PIN = P4;
```

The ordering of entries relative to one another in the port-to-pin mapping file determines the ordering of the logical ports to physical pins. However, the specific pin numbers denoted in the port-to-pin mapping file are placeholders only and do not map to actual package pin numbers.

Defining Port and Pin Naming Relationships

When you generate a BSDL file that corresponds to the boundary-scan design, you need to show the correspondence between logical ports and physical pins. The port-to-pin mapping file allows you to define this relationship.

Creating the Port-to-Pin Mapping File

The port-to-pin mapping file contains package type and port ordering information.

The package type information is listed in the following manner:

```
PACKAGE = insert_my_package;
```

Use only one package type for each pin mapping file.

List elements in the port-to-pin mapping file first by logical port name and then by physical pin number.

```
PORT = port_name, PIN = pin_number;
```

For example, if you want to connect the port RESETN to package pin P5, you would enter the following information into your pin mapping file:

```
PORT = RESETN, PIN = P5;
```

Example 5-4 shows a port-to-pin mapping file.

Example 5-4 Port-to-Pin Mapping File

```
        PACKAGE = insert_my_package;
        PORT = IN1, PIN = P1;
        PORT = IN2, PIN = P2;
        PORT = IN3, PIN = P3;
        PORT = EN, PIN = P4;
        PORT = RESETN, PIN = P5;
        PORT = IN4, PIN = P6;
        PORT = CLK, PIN = P7;
        PORT = CLOCKDR, PIN = P8;
        PORT = configENABLE, PIN = P9;
        PORT = config_in, PIN = P10;
        PORT = my_tdi, PIN = P11;
        PORT = my_tms, PIN = P12;
        PORT = my_tck, PIN = P13;
        PORT = my_trst, PIN = P14;
        PORT = configCAPTUREx, PIN = P15;
        PORT = configENABLEx, PIN = P16;
        PORT = OUT3, PIN = P17;
        PORT = OUT6, PIN = P18;
        PORT = OUT6enable, PIN = P19;
        PORT = configCAPTURE, PIN = P20;
        PORT = my_tdo, PIN = P21;
        PORT = OUT2, PIN = (P26, P27, P28, P29, P30, P31);
        PORT = CONFIGURATION(3:0), PIN = (P42, P43, P44, P45);
        PORT = VDD, PIN = (P46, P48, P50);
        PORT = GND, PIN = (P47, P49, P51);
```

Creating an Initial Pin Map File

The following example of a Tcl script creates a basic pin map file using the design port declaration where you can change the order of the ports in this file according to your specific package pin map:

```
proc CreatePinMap {filename package} {
set f [open $filename "w"]
puts $f "PACKAGE = $package;"
set n 0
foreach_in_collection p [get_port *] {
incr n
puts $f "PORT = [get_object_name $p], PIN = P$n;"
}
close $f
}
# The pin map is needed to write BDSL file
CreatePinMap "pin_map.txt" my_package
```

Verifying the BSD Configuration and Specification

You can verify the boundary-scan configuration and the current state of the BSD compliance-enable specification by specifying the `preview_dft -bsd all` command.

Removing BSD Specifications

You can use the `remove_boundary_cell_io` command to remove BSD specifications from your boundary-scan design that you inserted by using the BSD specification commands with the exception of `check_bsd`, `write_bsdl`, and `create_bsd_patterns`.

Generating Your BSDL File

You can generate a BSDL file for your boundary-scan design by using the `write_bsdl` command.

The `write_bsdl` command has the following syntax:

```
write_bsdl [-naming_check VHDL | BSDL | none]
           [-output file_name]
           [-effort low | medium | high]
```

Invoke this command immediately after running the `insert_dft` command or the `check_bsd` command (for 1149.1 only). Invoking other commands could invalidate the BSL design data from which the BSDL and test patterns are generated.

The command has the options shown in Table 5-6.

Table 5-6 write_bsdl Command Options

Option	Description
`-naming_check`	Contains VHDL and BSDL reserved words. Some BSDL readers accept these reserved words in the BSDL file; others do not. This option enables you to specify the reserved words BSD Compiler checks during design processing. BSD Compiler performs naming checks on port, bus, register, instruction, package pin, and identifier names.
`-output` *`file_name`*	Defines the file name for the output BSDL file. The file name can use either an absolute or relative path name. You must specify the full file name when using the `-output` option. BSD Compiler does not append a default suffix.
`-effort`	Controls the effort used to search for implemented instructions.

Table 5-7 Choices for the -effort option

Option	Description
`low`	BSD Compiler uses heuristic methods to extract boundary-scan instructions.
`medium`	This is the default effort value. BSD Compiler uses heuristics and then random opcode generation to extract boundary-scan instructions.
`high`	BSD Compiler uses heuristic methods and full sequential search to extract boundary-scan instructions.

Performing Naming Checks

The `-naming_check` option has the choices shown in Table 5-8.

Table 5-8 The naming_check Option

Option	Description
`-naming_check VHDL`	Specifies checks for both VHDL and BSDL reserved words. BSD Compiler generates warning messages for names that use either VHDL or BSDL reserved words. This is the default naming_check value.
`-naming_check BSDL`	Specifies checks for BSDL reserved words only. BSD Compiler generates warning messages for names that use the BSDL reserved words. BSD Compiler writes VHDL reserved words to the BSDL file without warnings.
`-naming_check none`	Disables all naming checks.

If you want to write a BSDL file and do a naming check on that file, use the following command:

```
write_bsdl -naming_check BSDL -output filename.bsdl
```

Note:
The `-naming_check` options are mutually exclusive. Running the command with a different `-naming_check` option overwrites the previous value.

Note:
If a boundary-scan design is noncompliant, the `write_bsdl` command does not generate any BSDL file.

See Chapter 4, “Verifying the Boundary-Scan Design,” for more information.

Defining the Bused Ports

If your design contains bused ports, the BSDL generator uses the `bus_naming_style` variable to determine how to output the bused port names.

Generating BSDL and BSD Patterns for Multiple Packages

Some ports in your design might not be wire-bound, that is, no-connect, to any pin in the current package pin map file, but they can have BSR cells associated with their ports. BSD Compiler describes these ports as linkage ports and the BSR cells associated with these ports as internal BSR cells. These ports are also commented as no-connect ports in the BSDL file.

In addition, for such packages, BSD Compiler generates BSD patterns that don't drive or sense the no-connect ports.

To generate a BSDL file in which the non-wire-bound ports are characterized as no-connect ports or linkage ports in the BSDL file, or to generate patterns for a package that has no-connect ports, do the following:

1. Use the `read_pin_map` command to read in the package pin map file that wire bonds all ports of the design.
2. Insert BSD logic into the design and map the BSD logic.

   Steps 1 and 2 are not required for the verification flow, in which you manually insert the BSD logic into the design.
3. Run the `check_bsd` command on the design.
4. Use the `read_pin_map` command to read in the package pin map files. Some of these pin map files might contain no-connect ports.
5. To write out BSDL or BSD patterns for a package, specify the package as the default package with the `set_bsd_configuration -default_package` command.

For example, suppose design M1 has two package configurations. Package M1_large_pkg bonds out all the ports in the design. Package M1_small_pkg contains no-connect ports in2, out2, and inout2. Both of these packages have unused pins in3, out3, and inout3. The following script fragment shows boundary scan insertion for package M1_large_pkg. The script then runs compliance checking on the BSD-inserted design and outputs a BSDL file and pattern file. Next, the script reads in package M1_small_pkg and generates its BSDL and pattern files.

Example 5-5 Multiple Packages

```
#Read the package that bonds out all the ports in the design
read_pin_map M1_large.pinmap
insert_dft
check_bsd -verbose
#Set the default package
set_bsd_configuration -default_package M1_large_pkg
#Generate BSDL file for package M1_large_pkg
write_bsdl -naming_check BSDL -output M1_large.bsdl
```

```
#Generate patterns for package M1_large_pkg
create_bsd_patterns
write_test -format stil_testbench -output M1_large_stil_tb
#Read the pin map for the reduced package
read_pin_map M1_small.pinmap
#Generate BSDL file for package M1_small_pkg
set_bsd_configuration -default_package M1_small_pkg
write_bsdl -naming_check BSDL -output M1_small.bsdl
#Generate patterns for package M1_small_pkg
create_bsd_patterns
write_test -format stil_testbench -output M1_small_stil_tb
```

The BSDL output annotates which ports are no-connect ports. For example, the following fragment for package M1_small_pkg declares all the ports in the design that are no-connect ports.

```
port (
     .
     .
     .
     trst_n   :   in        bit:
     inout1   :   inout     bit;
     tdo      :   out       bit;
     en       :   linkage   bit;   -- NC port
     in1      :   linkage   bit;   -- NC port
     in3      :   linkage   bit;
     inout2   :   linkage   bit;   -- NC port
     inout3   :   linkage   bit;
     out1     :   linkage   bit;   -- NC port
     out2     :   linkage   bit;   -- NC port
     out3     :   linkage   bit
);
```

BSR cells connected to no-connect ports are described as internal BSR cells in the BSDL file. Merged BSR cells have duplicate entries in the BSDL with the same cell number. If a merged BSR cell is associated with only one no-connect port, the corresponding cell entry does not appear in the BSDL file. If both ports associated with a merged BSR cell are no-connect ports, the cell is described as an internal BSR cell.

When a merged BSR cell also controls multiple ports and all its associated ports are no-connect ports, the BSR cell is described as an internal cell in the BSDL. When the input port associated with this BSR cell is a no-connect port, its cell entry is removed. If all the ports controlled by the BSR cells are no-connect ports, the cell entry for the control BSR cell is removed.

The following BSDL output fragment for package M1_small_pkg describes the BSR cells. In this example, BSR cells 7, 2, and 0 are exclusively connected to NC ports.

```
-- num   cell    port      function    safe   [ccell  disval reslt]
   7    (BC2,   *,       internal.     X),                        &
```

```
6     (BC1,    in2,      input.         X),                                 &
5     (BC2,    clk,      input,         X),                                 &
4     (BC1,    *,        control,       0),                                 &
3     (BC1,    en1,      input,         X),                                 &
2     (BC1,    *,        internal,      X),                                 &
1     (BC7,    inout1,   bidir,         X),       4,      0,     Z)         &
0     (BC7,    *,        internal,      X),                                 &
```

Detailed information on merged BSR cells can be found in section B.11.1.3, Merged Cells, pages 187-188, of the IEEE Std 1149.1-2001. Note in particular, BSR cell #9, which is a feed-through signal, and Figure B.9 on page 188, as well as the last paragraph describing the BSR cell #9 in this figure.

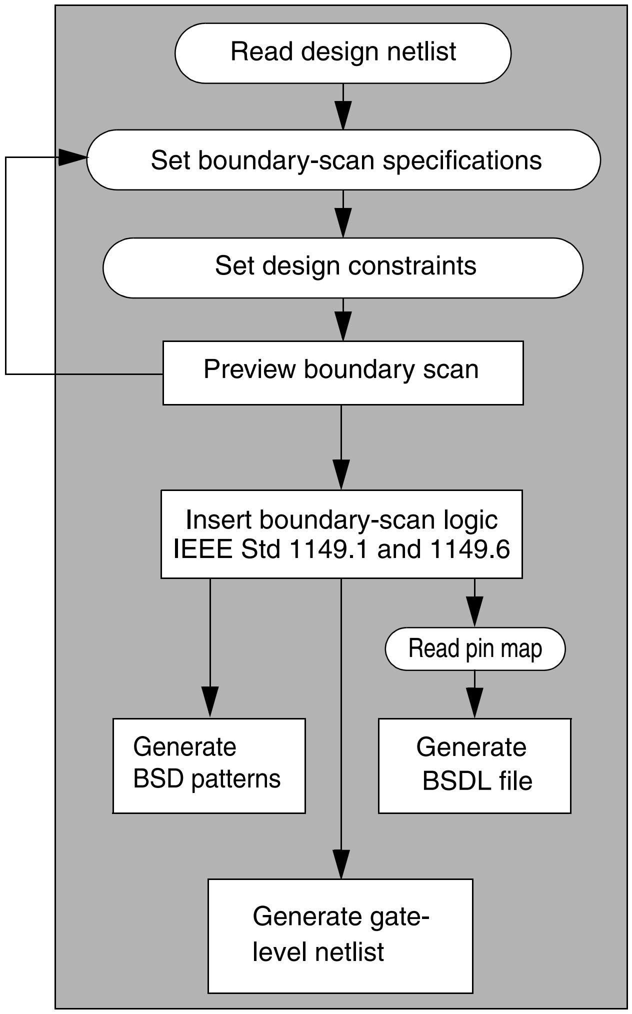
Generating BSDL and Test Patterns After BSD Insertion

You can optionally generate BSDL and test patterns after BSD synthesis and before compliance checking. This is particularly useful when you have inserted DesignWare components in your boundary-scan design.

The flow described in this section applies to the BSD insertion flow (when using `insert_dft`) but not the verification-only flow.

Figure 5-1 shows the flow for generating BSDL and test patterns after BSD insertion.

Figure 5-1 Generating BSDL and Test Pattern Flow



Example 5-6 illustrates a standard design flow using BSD Compiler.

Example 5-6 Generating BSDL and Test Pattern After BSD Insertion

```
read_file -format verilog des.v
current_design DESIGN
# Specify top port signals
set_dft_signal -view spec -type tdi -port tdi
set_dft_signal -view spec -type tdo -port tdo
set_dft_signal -view spec -type tck -port tck
set_dft_signal -view spec -type tms -port tms
set_dft_signal -view spec -type trst -port trst
# BSD Insertion
insert_dft
read_pin_map pin_map.txt
# BSDL generation
write_bsdl
create_bsd_patterns
# Verilog DPV testbench generation
write_test -format stil_testbench -output DESIGN
# Write the gate-level netlist
change_names -rules Verilog -hierarchy
write -f verilog -h -output des_gate.v
write -f ddc -h -output des_gate.ddc
quit
```

BSDL-Based Test Pattern Generation

The `read_bsdl` command in BSD Compiler reads the BSDL file to generate the BSD patterns in STIL, WGL, and Verilog format.

Use the `read_file` command to read the black box description of the netlist (I/O information only) into BSD Compiler; this step is optional.

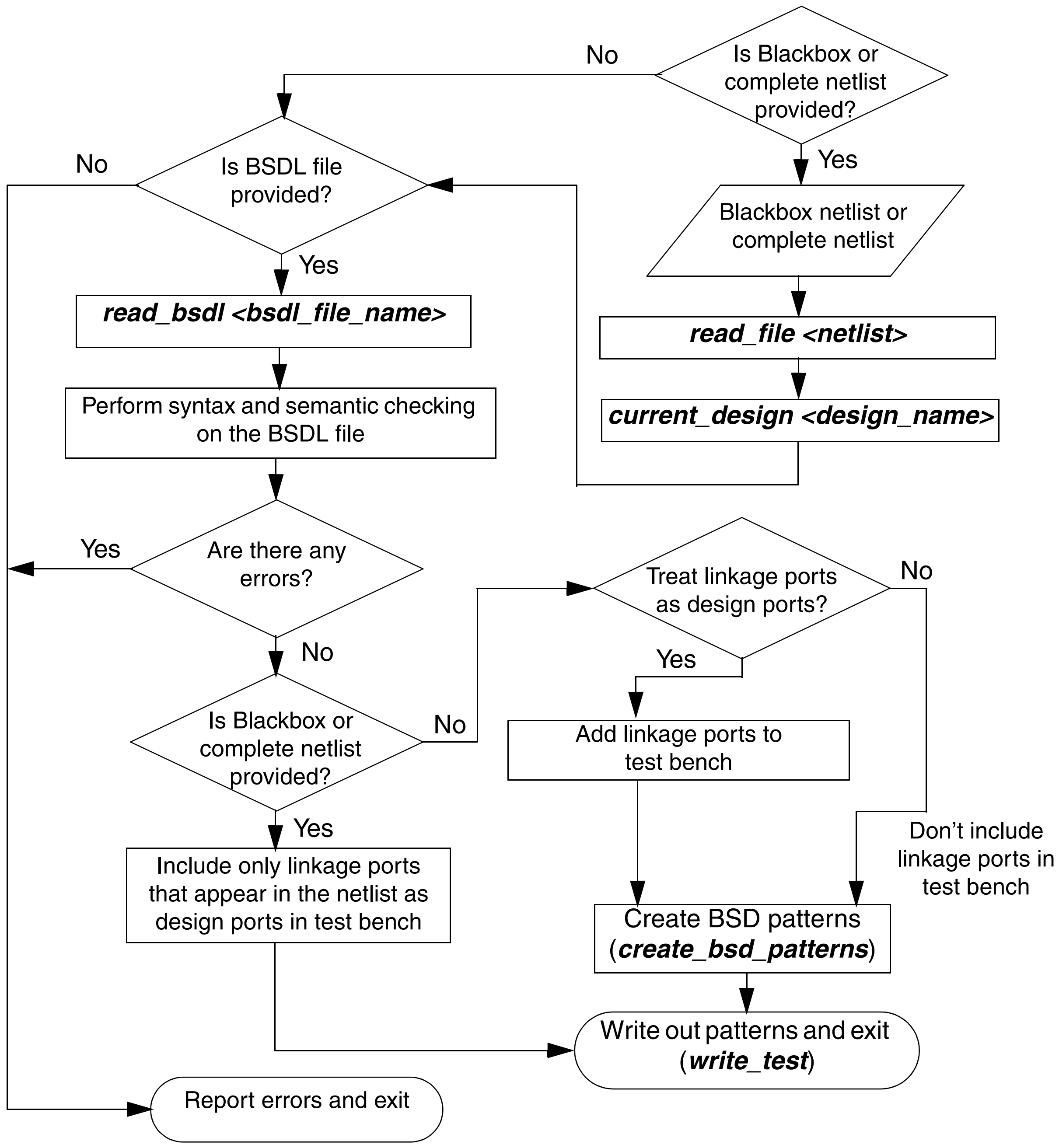
This feature is described in the following subsections:

- BSDL File to Pattern Generation Flow
- Reading the Black Box Description of the Netlist
- Reading the BSDL File
- Checking for Syntax and Semantic Errors in the BSDL File
- Automatic Test Pattern Generation From the BSDL File
- Limitations
- Example Script for Generating Test Patterns Using a Netlist and a BDSL File
- Example Script for Generating Test Patterns From a BSDL File Only
- References

BSDL File to Pattern Generation Flow

Figure 5-2 shows the BSDL file to pattern generation flow.

Figure 5-2 BSDL File to Pattern Generation Flow


Is Blackbox or complete netlist provided?
No
Yes
Blackbox netlist or complete netlist
read_file <netlist>
current_design <design_name>
Is BSDL file provided?
No
Yes
read_bsdl <bsdl_file_name>
Perform syntax and semantic checking on the BSDL file
Are there any errors?
Yes
No
Is Blackbox or complete netlist provided?
No
Yes
Include only linkage ports that appear in the netlist as design ports in test bench
Treat linkage ports as design ports?
Yes
No
Add linkage ports to test bench
Don't include linkage ports in test bench
Create BSD patterns (create_bsd_patterns)
Write out patterns and exit (write_test)
Report errors and exit

Reading the Black Box Description of the Netlist

The first step in the process of generating patterns from a BSDL file or a netlist is to read in the black box description of the netlist. Reading the netlist or the black box description is optional.

- Read in the complete netlist or the black box netlist that has the I/O description of the design, using the `read_file` command. Only the information in the BSDL file is considered in case a netlist is not read.
- Specify the name of the current design using the `set current_design` command. This step is optional.

Note:
If the netlist is also provided along with the BSDL file, only the linkage ports that appear in the netlist are treated as design ports in the testbench.

Reading the BSDL File

Read in the BSDL file using the `read_bsdl` command. The syntax for the command is as follows:

```
read_bsdl bsdl_file_name [ -add_linkage_as_design_port TRUE | FALSE]
```

The default value for the `add_linkage_as_design_port` option is `FALSE`, which will disregard all the linkage ports during the testbench construction when only the BSDL file is read.

If a netlist is not read in, then the linkage ports can be treated using either of the following two methods:

- If you set the `add_linkage_as_design_port` option to `TRUE`, the linkage ports are listed as design ports in the testbench.
- If you set the `add_linkage_as_design_port` option to `FALSE`, the linkage ports are not listed as design ports in the testbench.

Note:
NC ports are treated the same way as linkage ports.

BSD Compiler supports the following BSDL instructions in the BSDL file when reading in the file using the `read_bsdl` command:

- EXTEST
- BYPASS

- SAMPLE
- PRELOAD
- CLAMP
- HIGHZ
- IDCODE
- USERCODE
- EXTEST_PULSE
- EXTEST_TRAIN
- INTEST
- RUNBIST
- User-defined instructions

BSD Compiler supports the following cell types when reading the BSDL file with the `read_bsdl` command:

- BC_0
- BC_1
- BC_2
- BC_4
- BC_7
- AC_1
- AC_2
- AC_7
- AC_SelX
- AC_SelU

Checking for Syntax and Semantic Errors in the BSDL File

BSDL compilation determines if a BSDL description is syntactically correct and makes numerous checks for semantic violations.

The following syntax checks are performed on a BSDL file that is read in using the `read_bsdl` command:

- Missing punctuation
- Misspelled keywords
- Missing opcodes
- Missing IR capture value
- VHDL and Verilog naming conventions

The following semantic error checks are performed on a BSDL file that is read using the `read_bsdl` command:

- Omission of required instruction codes, for example, SAMPLE
- Missing register associations
- Missing boundary-scan register cells
- Unrecognized attributes in the BSDL file that are flagged as warnings
- Unrecognized statements in the BSDL file that are flagged as errors
- Incomplete statements that are flagged as errors
- Missing logical or physical port descriptions
- Missing standard package use or COMPONENT_CONFORMANCE statements
- Missing TAP description
- Missing instruction register length or capture value
- Missing mandatory instruction definitions in accordance with IEEE Std 1149.1
- Instructions without opcodes or test data registers
- Missing boundary-scan register length
- Unrecognized boundary-scan register cell statements
- Unrecognized port names
- Incorrect control cell numbers

If BSD Compiler encounters an error during the syntax or the semantic errors check, it generates an error report and exits.

Some of the reported errors are

- Missing BSDL input file.
  To remedy this error, make sure that the BSDL file is available in the search path and rerun the test.
- Missing IO netlist.
  To remedy this error, make sure that the netlist with IO information is available in the search path and rerun the test.

Any existing User Interface Test messages for BSD Compiler are used as applicable.

Automatic Test Pattern Generation From the BSDL File

If the syntax and semantic error checks are successful, the BSD patterns are created with the `create_bsd_patterns` command.

When the BSD patterns are created, they are written out using the `write_test` command. These patterns can be simulated using VCS. The absence of mismatches implies that the BSDL file parsing mechanism has generated the correct results. The output BSDL file might differ from the input BSDL file in the sequence of BSDL statements. A BSDL file that is written out using the `write_bsdl` command is readable using the `read_bsdl` command.

Note:
When the input and the output BSDL files have the same name, the comments in the input file do not appear in the output file.

The `create_bsd_patterns` and the `write_test` commands are used as is. There are no changes to these commands in this flow.

Example Script for Generating Test Patterns Using a Netlist and a BDSL File

The following script shows you how to generate test patterns using a netlist and a BSDL file.

```
set search_path [list "." ./lib $search_path]
set synthetic_library [list standard.sldb dw_foundation.sldb]
set target_library [list  class.db]
set link_library [concat "*" $target_library $synthetic_library]
read_file -format verilog f_bsd.v
current_design test
link
read_bsdl f_bsd.bsdl
read_pin_map pin_map.txt
create_bsd_patterns
write_test -f stil_testbench -output f_stil_tb
write_test -f verilog -output f_verilog_tb
write_test -f wgl_serial -output f_wgl_serial
```

Example Script for Generating Test Patterns From a BSDL File Only

The following script shows you how to generate test patterns from a BSDL file only:

```
set search_path [list "." ./lib $search_path]
set synthetic_library [list standard.sldb dw_foundation.sldb]
set target_library [list class.db]
set link_library [concat "*" $target_library $synthetic_library]
link
read_bsdl f_bsd.bsdl
create_bsd_patterns
write_test -f stil_testbench -output f_stil_tb
write_test -f verilog -output f_verilog_tb
write_test -f wgl_serial -output f_wgl_serial
```

Limitations

- The following boundary-scan register cell types are not supported: BC_3, BC_5, BC_6, BC_8, BC_9, BC_10, AC_8, AC_9, and AC_10.
- Not all constructs of the BSDL syntax are supported. For example, user supplied packages are not supported by the BSDL reader.
- The following boundary-scan register cells support INTERNAL function: BC_0, BC_1, BC_2, BC_3, and BC_4

References

- 1149.1-1993 IEEE standard test access port and boundary-scan architecture.
- 1149.1-2001 IEEE standard test access port and boundary-scan architecture.
- 1149.6-2003 IEEE Standard for Boundary-Scan Testing of Advanced Digital Networks.

Fault Grading BSD Patterns With TetraMAX

You can fault grade the test vectors you wrote out in BSD Compiler with TetraMAX to get additional test coverage. To get detailed information on configuring your TetraMAX environment, see the TetraMAX documentation.

Formatting BSD Test Vectors in WGL_serial

Using BSD Compiler, format the test vectors in wgl_serial using the following command:

```
write_test -format wgl_serial -output my_pattern_filename
```

Then write the design netlist in Verilog using the following command:

```
write -hier -format verilog -output bsd_design_filename.v
```

Effect of the test_preset_bidi_signal on all_bidirectionals

The setting for the `test_preset_bidi_signal` affects the timing template of the `all_bidirectionals` in the `wgl_serial` patterns. When FALSE is specified, any Z event at time zero is changed to an X for `all_bidirectionals`. When it is TRUE (the default), the waveform description from the timing definition as specified in the incoming SPF is not changed. In this case, any waveform event at time zero that specifies an X is changed to a Z for `all_bidirectionals`. (Other events remain the same, including any P values.)

The effect of this variable is notable for patterns that change from output to input mode by passing through an X-transition state. This behavior avoids contentions and mismatches. Similarly, patterns that go from input to output mode transition through a Z state to avoid contentions or mismatches.

Therefore, when the variable is TRUE, Z is used in the WFT, as shown in the following:

```
"all_bidirectionals" {T{'Ons'Z;'40ns'T;}}
"all_bidirectionals" {X{'Ons'Z;'40ns'X;}}
"all_bidirectionals" {H{'Ons'Z;'40ns'H;}}
"all_bidirectionals" {L{'Ons'Z;'40ns'L;}}
```

When the variable is FALSE, X is used in the WFT, as shown in the following:

```
"all_bidirectionals" {T{'Ons'X;'40ns'T;}}
"all_bidirectionals" {X{'Ons'X}}
"all_bidirectionals" {H{'Ons'X;'40ns'H;}}
"all_bidirectionals" {L{'Ons'X;'40ns'L;}}
```

Using BSD Test Vectors in TetraMAX

After you write out your formatted vectors using BSD Compiler, you can fault grade them with TetraMAX.

Invoke TetraMAX, specifying the architecture of your platform in the path for the TetraMAX executable file.

```
% $SYNOPSYS_TMAX/arch/syn/bin/tmax &
```

In TetraMAX you read in libraries and the design netlist, and then compile using the following commands:

```
set_messages -log ./f_tmax.log -r
read_netlist ./lib_sim/vlib/*.v
read_netlist ./netlist/f_bsd.v
run build demo
run build_model demo
```

Next, run design rule checking (DRC), set the pattern's source, and simulate, using the following commands:

```
add clock 0 TCK
add pi constraint 1 TRST
run drc
set patterns external ./pat/f_wgl_tb.wgl
```

Finally, add all the faults and run fault simulation by using the following commands:

```
add faults –all
run fault_sim -seq
report summaries
```

Using Scan-Through Vectors in TetraMAX

For scan-through TAP designs, all capture procedures should use TCK and all registers need to be initialized to 0. Use `set_drc -clock TAP_TCK -init 0`, as shown in the following script example:

```
read_net ./tmax_lib/class.v
read_net ./gates/top_design_bsd.v
run_build TOP
add_clock 0 TAP_TCK
set_drc -clock TAP_TCK -init 0
run_drc top_design_bsd_stt.spf
add_faults -all
run_atpg -auto
```

Simulating BSD Patterns With VCS

You can use BSD Compiler to generate a Verilog DPV testbench for vectors simulation. For VCS simulation of Verilog DPV Testbench, you read and compile the design netlist, testbench, and Verilog DPV simulation libraries. To do a Verilog DPV interactive simulation, use the following command:

```
vcs -RI\
    +acc+2 -notice \
    +delay_mode_zero \
    +nospecify \
    +notimingcheck \
    +udpsched \
    +evalorder \
    +vcs+lic+wait \
    -timescale=1ns/10ps \
    -P $STILDPV_HOME/stildpv_vcs.tab \
    $STILDPV_HOME/libstildpv.a \
    -v ./LIB/*.v \
    ./f_bsd.v \
    ./mypat_dpv.v  \
    -l ./simv.log \
    -o simv

vcs -R f_all_pat.v_0.v test.gate.v \
     -y ./lib/verilog/technology_libs +libext+.tsbvlibp \
     +.tsbvlib+.v+notimingcheck -l f_simvtb.log
```

If the simulation library is in the same directory where the design and testbench are located, this command is correct. Otherwise, use the `-y` or `-v` command argument if the simulation library is located in another directory appropriately. For details see the VCS/VCSi User Guide.

The file mypat_dpv.v is the Verilog DPV file that calls the STIL patterns and the PLIs associated with the Verilog simulator.

The following techniques ensure the best and consistent results:

1. Delete compiled directories from previous simulations (that is, csrc and *.daidir).
2. Simulate and debug the chain test pattern as early as possible. This is a very important first step in making sure the rest of the BSD patterns work correctly.

   If you separate patterns, be sure to include reset sequences to initialize the TAP controller.
3. Minimize the use of VCS switches. Avoid using any VCS optimization switches such as +rad+, +acc+, +2state, or +vcsd, because the accuracy of results can vary with netlist revisions (incremental netlist updates).

4. If simulating with VCS 6.0, use the undocumented +nogateperf option to prevent global optimization on clock signals.

   Note:
   The switch option +gateperf is on by default. This is an optimization switch for better runtime performance with gate-level netlists. It is the default because it improves simulation time by 7x. Always turn this switch off using +nogateperf.

5. Check with the ASIC vendor documentation and library provider for special requirements on when not to use back-annotated timing in a zero delay, unit delay, or functional delay simulation.

6. When not using both asynchronous TRST and power-up reset, add the following initial block to the Verilog DPV Testbench. Without asynchronous TRST and power-up reset, the TAP FSM does not initialize and simulation fails. Adding the initial block prevents simulation failure.

```
initial begin

 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[15] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[14] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[13] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[12] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[11] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[10] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[9] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[8] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[7] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[6] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[5] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[4] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[3] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[2] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[1] .Q = 1'b0;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[0] .Q = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[15] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[14] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[13] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[12] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[11] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[10] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[9] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[8] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[7] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[6] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[5] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[4] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[3] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[2] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[1] .QN = 1'b1;
 force TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[0] .QN = 1'b0;

 #290

 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[15] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[14] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[13] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[12] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[11] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[10] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[9] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[8] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[7] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[6] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[5] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[4] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[3] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[2] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[1] .Q ;

 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[0] .Q ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[15] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[14] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[13] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[12] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[11] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[10] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[9] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[8] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[7] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[6] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[5] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[4] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[3] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[2] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[1] .QN ;
 release TOP_inst.TOP_DW_tap_inst.\U1/current_state_reg[0] .QN ;

 end
```

You can use BSD Compiler to generate the Verilog testbench for vector simulation. For VCS simulation of Verilog Testbench you read and compile the design netlist, Verilog testbench, and simulation libraries. To do a Verilog interactive simulation use the following command:

```
vcs -Mupdate -RI \
    +v2k \
    +acc+2 -notice \
    +delay_mode_zero \
    +nospecify \
    +notimingcheck \
    +udpsched \
    +evalorder \
    +vcs+lic+wait \
    -timescale=1ns/10ps \
f_verilog_tb.v \
f_bsd.v \
../lib/my_class.v \
    -l f_verilog_tb.log
```

If simulation yields a mismatch, consider the following tips to determine the possible cause:

- Test vectors from BSD Compiler should be simulated with zero delay. When using VCS to simulate BSD generated vectors, use the +delay_mode_zero option during compile time. Also, use this option when simulating vectors generated with JTAG inserted gate level netlist and gate delays.
- Simulation models should match the functionality of the synthesis library models in the .lib files. Differences in these two libraries can result in a simulation mismatch. Determine which simulation cell causes the simulation mismatch and verify that the basic functionality of this cell is the same in both libraries.
- Verify that the correct I/O pad constraints are applied to enable or disable pad pull-ups and pad pull-downs, or to the MUX selected signals in the pad cell.
- Changes made to the netlist as post-processing can affect the simulation results if BSD patterns are not re-generated.
- Check the correct environment setting for your Verilog simulator; the root path might be specified incorrectly.
- If your design was inserted with BSD Compiler, make sure all the pad issues were resolved during `preview_dft` and `check_bsd` before writing the patterns for simulation.

When using VCS to debug simulation mismatches, the first step is to set up a good configuration file for the simulator to display the instruction bus so that you know what instruction is active, and the TAP FSM bus, so that you know what TAP state you are in when debugging the JTAG logic. Then you can display ports failing the simulation and can trace back to pad cell signals to see where you are losing your simulated patterns.

A

Custom Boundary-Scan Design

This appendix describes ways you can customize your boundary-scan specifications. For example, you can customize parameters for TAP signals, boundary-scan cell types, test data registers, and instructions.

This appendix includes the following sections:

Defining Custom Boundary-Scan Components

You can use any design as a custom BSR cell or TAP controller. There are no requirements on the state of the design; it can be gate-level, RTL, mixed, or black box (empty cell). After you characterize the design component to define its interface with the rest of the boundary-scan logic, you can insert the component into the boundary-scan design.

No verification is performed on the design at this stage to determine if it functions properly as a BSR cell or TAP controller. The only check performed on the design is in the compliance checking phase after the boundary-scan design is completed and mapped.

You can use the `define_dft_design` command and the `set_boundary_cell` command to define custom boundary-scan cells.

Using Custom BSR Cells

You might want to use your own boundary-scan register cells in place of the DesignWare foundation library cells that are provided by default during boundary-scan insertion. You do this by specifying a design to be used, identifying the type of boundary-scan register cells it replaces, and then associating BSR signals with design pins. The custom cell must be loaded in the Design Compiler database, and the access interface must match that of the DesignWare foundation components.

Use the `define_dft_design` command as follows to customize your boundary-scan register configuration.

```
define_dft_design -design_name design_name
                   [-type AC1|AC2|AC7|AC_SELU|AC_SELX|BC1|BC2|BC4|BC7]
                   [-interface access_list]
```

If a default BSR cell was not defined implicitly or explicitly, BSD Compiler uses the equivalent DesignWare cell of that type as the default cell.

For example, you might want to use your own custom design to replace the DesignWare foundation library element BC1 that is normally inserted into the boundary-scan design. You would do this with a command similar to that shown in Example A-1.

Example A-1 define_dft_design Example

```
dc_shell> define_dft_design -type BC1 \
           -interface [list si TDI H data_in DI H shift_dr SHIFTDR H \
           capture_clk CL CLKDR H update_clk UPDATEDR H mode MODE_1 H \
           data_out DO H so TDO H] \
           -design_name user_bsr_bc1
```

See the DesignWare documentation for additional information on the BSR.

BSD Compiler supports multiple implementations of the default DesignWare BSR cells of the same type for boundary-scan synthesis. You can use DesignWare cells at multiple ports or a mix of DesignWare and custom BSR cells.

For example, you might want to use the DesignWare foundation element BC1 along with the element BC2. You would do this with commands similar to those shown in Example A-2. A flow using custom BSR cells is show in Example A-3.

Example A-2 define_dft_design Example

```
dc_shell> define_dft_design -design_name BC1_SLOW -type BC1 \
           -interface {capture_clk my_capture_clk h \
                      update_clk my_update_clk h \
                      data_in my_data_in h \
                      data_out my_data_out h  \
                      shift_dr my_shift_dr h \
                      si my_si h \
                      so my_so h \
                      mode my_mode h}

dc_shell> define_dft_design -design_name BC1_FAST -type BC1 \
             -interface {...}

dc_shell> define_dft_design -design_name BC2_FAST -type BC2 \
             -interface {...}

dc_shell> define_dft_design -design_name B2_SLOW -type BC2 \
           -interface {...}
```

Example A-3 Custom BSR Flow With define_dft_design & set_boundary_cell

```
…
#specify the custom BSR that was read in the script.
define_dft_design -type BC4 \
     -interface [list si TDI h data_in DI h shift_dr SHIFTDR h \
          capture_clk CLOCKDR h so TDO h] \
     -design_name my_bsr_bc4
define_dft_design -type BC1 \
     -interface [list si TDI h data_in DI h shift_dr SHIFTDR h \
          capture_clk CLOCKDR h update_clk UPDATEDR h \
          mode MODE_11 h data_out DO h so TDO h] \
     -design_name my_bsr_bc1
#specify where you want to place the custom BSR
set_boundary_cell -class bsd -design_name my_bsr_bc4 -function input \
     -ports {list input_port_name}
set_boundary_cell -class bsd -design_name my_bsr_bc1 -function output \
     -ports {list output_port_name}

set_boundary_cell -class bsd -design_name my_bsr_bc2 -name MY_BC2 \
     -function control -ports [get_ports sdo0]
```

The ports not on the list use the BSRs from the DesignWare foundation.

Using a Custom TAP Controller

You can use your own TAP controller in place of the DesignWare foundation library TAP controller that is provided by default during boundary-scan insertion. You do this by specifying a design to be used and then associating TAP controller signals with design pins. The design must be loaded in the Design Compiler database.

Use the `define_dft_design` command, as follows, to customize your TAP controller configuration.

```
define_dft_design -design_name design_name
                  [-type TAP | TAP_UC]
                  [-interface access_list]
```

To use your own custom design to replace the DesignWare foundation library TAP controller that is normally inserted into the boundary-scan design, use a command similar to that shown in Example A-4.

It uses the instruction bus from the TAP to decode the op-codes for all the instructions and placed the decoded in the mode block.

Example A-4 is an example for custom TAP using an equivalent DW_tap_uc interface of type TAP_UC.

Example A-4 define_dft_design Command Example for Synchronous Custom TAP Controller

```
#/* defining Custom TAP controller */
define_dft_design -type TAP_UC  \
   -interface [ list tck tck H \
                     tdi tdi H \
                     tms tms H \
                     so so H \
                     tdo tdo H \
                     tdo_en tdo_en H \
                     trst_n trst_n H \
                     shift_dr shift_dr H \
                     sync_capture_en sync_capture_en H \
                     bypass_sel bypass_sel H \
                     sync_update_dr sync_update_dr H \
                     instructions instructions H] \
   -design_name my_tap_uc
```

Note:

The access interface must match that of the DW_tap IPs: `DW_tap` and `DW_tap_uc`.

When using a custom TAP equivalent to a DW_tap access interface use variable: `bsd_use_old_tap true` and –`type TAP`.

When using a custom TAP equivalent to a `DW_tap_uc` access interface set the variable to false (default) and use –`type TAP_UC`.

See the *BSD Compiler Reference Manual* for additional information on the TAP access pin semantics.

Customizing the Boundary-Scan Register

You can customize the boundary-scan register before you perform boundary-scan insertion on your design. This customization allows you to change the default boundary-scan register configuration to suit your design requirements.

Specifying Control Cells

You can use the `set_boundary_cell` command to declare a control cell name and assign to it a specific control cell type to enable the specified tristate ports and bidirectional ports.

This command allows you to specify the control cells to be shared between specific tristate output ports. Use the `set_scan_path` command to share a control cell with other ports. See "Ordering the Boundary-Scan Register With the set_scan_path Command" on page A-8.

To specify control cells, use the `set_boundary_cell` command as follows:

```
set_boundary_cell
  -class bsd
  [-design_name design_name]
  [-function input|output|control|bidir|observe|input_inverted|
      output_inverted|bidir_inverted|receiver_p|receiver_n|ac_select]
  [-ports port_list]
  [-type BC1|BC2|BC4|BC7|BC8|BC9|AC1|AC2|AC7|AC_SELU|AC_SELX|none]
  [-share true | false]
  [-name bcell_name]
```

The command has the options shown in Table A-1.

Table A-1 set_boundary_cell Command Options

Option	Description	Choices
`-class`	Specifies the name of the class for which the configuration is applicable.	Valid values are `core_wrapper`, `shadow_wrapper`, and `bsd`.
`-function`	Specifies the function of the specified ports to which the local configuration applies.	`input` | `output` | `control` | `bidir` | `observe` | `input_inverted` | `output_inverted` | `bidir_inverted` | `receiver_p` | `receiver_n` | `ac_select`

Table A-1 set_boundary_cell Command Options (Continued)

Option	Description	Choices
`-ports` *`port_list`*	Specifies the list of ports for which the configuration applies.	There is no default value for this option. Either the -ports option or the -core option is required.
`-type` *`cell_type`*	Specifies the cell type to be used for the boundary cell in DFT insertion.	BC1 | BC2 | BC4 | BC7 | AC1 | AC2 | AC7 | AC_SELU | AC_SELX There is no default value for this option. If no option is not specified, the tool uses the set_wrapper_configuration command to get the type of DFT design to use for the specified ports.
`-design_name` *`design_name`*	Specifies the name of the design to be used for boundary cell in DFT insertion.	
`-share`	Specifies whether the boundary cells should be shared. Sharing boundary cells is allowed only for control boundary cells. If the value of the option is set to true, the boundary cells for the specified ports are shared. If the value of the option is set to false, the boundary cells for the specified ports are not shared.	TRUE FALSE The default value of this option is true for function value control and false for all other types of boundary cells.
`-name` *`bcell_name`*	Specifies the name of the boundary cell.	There is no default value for this option. The name of the boundary cell is mandatory if the -share option is set to true.

Example A-5 shows a boundary-scan control register, control, of type BC1, controlling the tristate output ports out0, out1, and out2.

Example A-5 Using the set_boundary_cell Command

```
dc_shell> set_boundary_cell
           -class bsd -type BC1 \
           -function control \
           -ports {port1, port2} \
           -name CTRL1 \
           -share false
```

Specifying Data Cells

You can use the `set_boundary_cell` command to assign a specific data cell type to enable specified data ports.

To specify data cells, use the `set_boundary_cell` command as follows:

```
set_boundary_cell
  -class bsd
  [-type AC1|AC2|AC7|BC1|BC2|BC4|BC7|NONE]
  [-design_name design_name]
  [-function input|output|bidir|observe|input_inverted|
      output_inverted|bidir_inverted|receiver_p|
      receiver_n|ac_select]
  [-port port_name]
```

The command has the options shown in Table A-2.

Table A-2 set_boundary_cell Command Options

Option	Description	Choices
`-class`	Specifies the name of the class for which the configuration is applicable.	
`-type` *`cell_type`*	Specifies the type of boundary-scan cell to be used for the declared control cell. This is an optional switch.	AC1 AC2 AC7 BC1 BC2 BC4 BC7
`-design_name` *`design_name`*	Specifies the name of the design to be used for boundary cell in DFT insertion.	

Table A-2 set_boundary_cell Command Options (Continued)

Option	Description	Choices
`-function` *`function_type`*	Specifies the function of the specified ports to which the local configuration is applicable.	
`-port_list` *`port_list`*	Identifies design tristate ports that are to be enabled by the declared control cell.	*port_list* is checked so that it contains only tristate output ports.

Note:
The recommended way to avoid inserting a BSR cell on the port is to use the `set_bsd_linkage_port` command instead of the `set_boundary_cell -type none` command. See “Inserting Boundary-Scan Components” in Chapter 2”.

An example of the use of the `set_boundary_cell` command is shown below. This example assigns in input and output ports for cell BC1.

```
dc_shell> set_boundary_cell -class bsd -type BC1 \
     -function input -port {out in in1 in2 }
dc_shell> set_boundary_cell -class bsd -type BC1 \
     -function output -port {out out1}
```

Ordering the Boundary-Scan Register With the set_scan_path Command

By default, BSD Compiler orders the BSR cells in alphabetical order according to their port declarations in the design.

The `set_scan_path` command specifies the order of the control and data cells in the boundary-scan register. The order you specify indicates the sequence in which cells are chained from the TDI port to the TDO port. Choose names for the cells that relate to the ports or to the identifiers you use to characterize a control boundary-scan register cell with the `set_boundary_cell` command.

If you specify only a partial list of the cells with the `set_scan_path` command, the last cell in the list is positioned just before TDO. The cells not specified by `set_scan_path` are ordered alphabetically, and they precede the first cell listed with the `set_scan_path` command. The order specified by `set_scan_path` overrides the order inferred from the port-to-pin mapping file.

Use the command as follows to order BSR cells:

```
set_scan_path -class bsd boundary \
     -ordered_elements ordered_list
```

If you do not specify your control cells with the `set_boundary_cell` command, they are placed, by default, next to data cells. The following example orders the cells with the scan chain named “boundary.”

Example A-6

```
set_scan_path -class bsd boundary \
     -ordered_elements {list en clk in2 in1 CTRL1 out0 out1 out2}
```

If you want to specify the position of your boundary-scan register control cells, first define the control cells by using the `set_boundary_cell` command. The identifier you assign to the cells with this command allows you to use the `set_scan_path` command to specify the sequence. See the *BSD Compiler Reference Manual* for more information on ordering with `set_scan_path`.

B

Setting Timing Attributes

This appendix provides information about setting timing attributes for both BSD Compiler and DFT Compiler.

This appendix contains the following sections:

Global Timing Attributes

When you choose not to use the default timing settings in BSD Compiler, you must set the timing using the variables shown in Example B-1. You need to use a strobe-after-clock-edge for TDO compliance with IEEE Std 1149.1. If your design's timing attributes are the same as the variables' default values, you do not need to make any changes.

Test timing attributes are associated with particular variables. The associations are shown in Table B-1.

Table B-1 Test Timing Attributes and Associated Variables

Attribute	**Variable**	**Default value**
`default_period`	`test_default_period`	100
bsd`_default_delay`	`test_bsd_default_delay`	0
bsd`_default_bidir_delay`	`test_bsd_default_bidir_delay`	0
bsd`_default_strobe`	`test_bsd_default_strobe`	95
bsd`_default_strobe_width`	`test_bsd_default_strobe_width`	0

For BSD Compiler, use a strobe-after-clock-edge, and set the timing values described in Example B-1.

If you intend to use a strobe-after-clock protocol, the timing attributes shown in Example B-1 appear in the inferred test protocol for the multiplexed flip-flop design example:

Example B-1 Timing Attributes for a Strobe-After-Clock Design

```
set test_default_period 100
set test_bsd_default_strobe 95
set test_bsd_default_strobe_width 0
set test_bsd_default_delay 0
set test_bsd_default_bidir_delay 0
```

Table B-2 lists the test timing attributes along with the keyword used and the definition of the attribute. The value of these attributes must be a positive real number. The time unit is nanoseconds (ns).

Table B-2 Test Timing Attributes

Attribute	Keyword	Definition
`default_period`	`period`	Duration of a tester cycle.
`bsd_default_delay`	`delay`	The time, relative to the start of the tester cycle, at which data is applied to all nonclock inputs.
`bsd_default_bidir_delay`	`bidir_delay`	The time, relative to the start of the tester cycle, at which data is applied to all bidirectional ports in input mode and is released from all bidirectional ports changing from input mode to output mode.
`bsd_default_strobe`	`strobe`	The time, relative to the start of the tester cycle, at which the output strobe occurs.
`bsd_default_strobe_width`	`strobe_width`	The width of the strobe pulse (a width of 0 indicates instantaneous strobe).

Setting Global Timing Attributes

When you know you are not using the BSD Compiler default timing, you must set the following variables before you run the `insert_dft` command.

```
test_default_period
test_bsd_default_delay
test_bsd_default_bidir_delay
test_bsd_default_strobe
test_bsd_default_strobe_width
```

The setting of test timing variables is discussed in subsequent sections.

You can define these variables every time you create a new design or you can add these variable values to your local .synopsys_dc.setup file or in your script.

test_default_period

The `test_default_period` variable defines the default value (in ns) for the period for compliance checking. The period value must be a positive real number.

The syntax for setting the variable is

```
set test_default_period period
```

For example,

```
dc_shell> set test_default_period 100.0
```

For BSD Compiler, the `test_default_period` default value is 100.0.

test_bsd_default_delay

The `test_bsd_default_delay` variable defines the default value (in ns) for the input delay for compliance checking. The delay value must be a nonnegative real number less than the strobe value (see the default timing in Figure B-1 on page B-6).

The syntax for setting the variable is

```
set test_bsd_default_delay delay
```

For example,

```
dc_shell> set test_bsd_default_delay 0.0
```

For BSD Compiler the `test_bsd_default_delay` default value is 0.0

test_bsd_default_bidir_delay

The `test_default_bidir_delay` variable defines the default value (in ns) for the bidirectional delay for compliance checking. The *bidir_delay* must be a positive real number less than the strobe value and can be less than, greater than, or equal to the delay value (see the default timing in Figure B-1 on page B-6).

The syntax for setting the variable is

```
set test_bsd_default_bidir_delay bidir_delay
```

For example,

```
dc_shell> set test_bsd_default_bidir_delay 0.0
```

For BSD Compiler the `test_bsd_default_bidir_delay` default value is 0.0.

test_bsd_default_strobe

The `test_bsd_default_strobe` variable defines the default value (in ns) for the strobe for compliance checking. The strobe value must be a positive real number less than the period value and greater than the `test_default_delay` value (see the default timing in Figure B-1 on page B-6).

The syntax for setting the variable is

```
set test_bsd_default_strobe strobe
```

For example,

```
dc_shell> set test_bsd_default_strobe 95.0
```

For BSD Compiler the `test_bsd_default_strobe` default value is 95.0.

test_bsd_default_strobe_width

The `test_bsd_default_strobe_width` variable defines the default value (in ns) for the strobe width for compliance checking. The strobe width value must be a positive real number. The strobe value plus the strobe width value must be less than or equal to the period value (see the default timing in Figure B-1 on page B-6).

The syntax for setting the variable is

```
set test_bsd_default_strobe_width strobe_width
```

For example,

```
dc_shell> test_bsd_default_strobe_width 5.0
```

For BSD Compiler the `test_bsd_default_strobe_width` default value is 0.0.

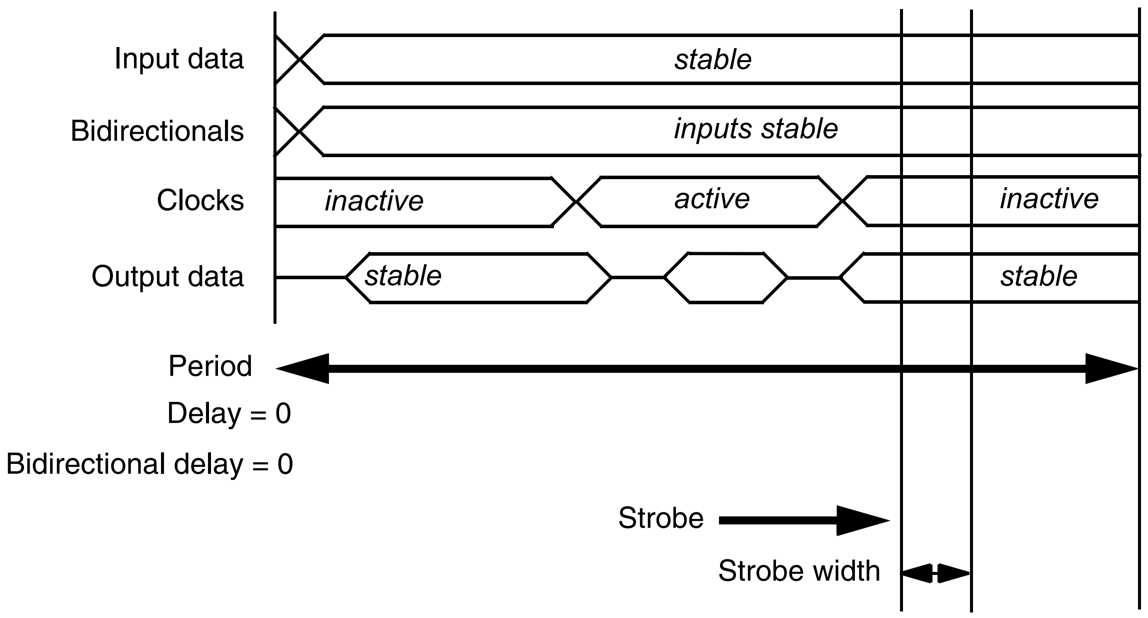
Note:
> When `test_bsd_default_strobe_width` is 0.0 ns, the strobe width is equal to one of two values: the difference between the strobe time and the end of the period, or the difference between the strobe time and the first input event after the strobe occurs, whichever occurs first.

Inferring Timing Attributes

The timing attributes for scan testing of the design are inferred during test design rule checking and defined in the test protocol group. These timing attributes influence test design rule checking and determine the timing in the test vector files produced by the `write_test` command. Timing attribute values ensure the accuracy of the rule checking process. They also provide important information that makes the test protocol file a complete test program template.

The timing diagram in Figure B-1 shows the effect of these timing attributes on vector formatting.

Figure B-1 Effect of Timing Attributes on Vector Formatting



Default Test Timing Information

Your semiconductor vendor’s requirements, together with the basic scan test requirements, drive the specification of test timing parameters.

Default timing parameter values include

- Test period

  By default, BSD Compiler uses a 100-ns test period. If your semiconductor vendor uses a different test period, specify the required test period using the `test_default_period` variable.

- Input timing

  By default, BSD Compiler applies data to all nonclock input ports 5 ns after the start of the cycle. If your semiconductor vendor requires different input timing, specify the required input delay using the `test_bsd_default_delay` variable.

- Bidirectional timing

  By default, BSD Compiler applies data to all bidirectional ports in input mode 0 ns after the start of the parallel measure cycle. In any cycle where a bidirectional port changes from input mode to output mode, BSD Compiler releases data from the bidirectional port 0 ns after the start of the cycle. If your semiconductor vendor requires different bidirectional timing, specify the required bidirectional delay using the `test_bsd_default_bidir_delay` variable.

  The risks associated with incorrect specification of the bidirectional delay time include:

  - Test design rule violations
  - Bus contention
  - Simulation mismatches

  Minimize these risks by carefully specifying the bidirectional delay time.

  BSD Compiler uses the bidirectional delay time as

  - The data application time for bidirectional ports in input mode during the parallel measure cycle (and during scan in for bidirectional ports used as scan input or scan-enable signal)
  - The data release time for bidirectional ports in input mode during cycles in which the bidirectional port changes from input mode to output mode

  BSD Compiler performs relative timing checks during test design rule checking. The following requirements must be met:

  - The bidirectional delay time must be less than the strobe time.

    If you change the strobe time from the default value, confirm that the bidirectional delay value meets this requirement.

  - If the bidirectional port drives sequential logic, the bidirectional delay time must be equal to or greater than the active edge of the clock.

- Output strobe timing

  By default, BSD Compiler compares data at all output ports 95 ns after the start of the cycle. If your semiconductor vendor requires different strobe timing, specify the strobe time using the `test_bsd_default_strobe` variable.

- Clock waveform requirements

  Clocking requirements specified by semiconductor vendors include

  - Clock waveform timing
  - Maximum number of unique clock waveforms
  - Minimum delay between different clock waveforms (to allow for clock skew on the tester)

  BSD Compiler provides the capability to specify clock waveform timing but does not place any restrictions on the number of unique waveforms that can be defined or the minimum time between clock waveforms. Determine what restrictions the semiconductor vendor places on these timing parameters, and define clock waveforms that meet the restrictions.

  When BSD Compiler infers clock ports during `dft_drc`, the clock type determines the default timing for each clock edge. Table B-3 provides the default clock timing for each clock type.

Table B-3 Default Clock Timing for Each Clock Type

Clock type	First edge	Second edge
Edge-triggered or D-latch enable	45.0	55.0
Master clock	30.0	40.0
Slave clock	60.0	70.0
Edge-triggered	45.0	60.0
Master clock1	50.0	60.0
Slave clock[a]	40.0	70.0

a. *Default timing for auxiliary-clock LSSD test clocks only.*

BSD Compiler determines the polarity of the first edge (rise or fall) so the first clock edge triggers a majority of cells on a clock. The timing arcs in the technology library specify each cell's trigger polarity. The polarity of the second edge is the opposite of the polarity of the first edge (if the first edge is rising, the second edge is falling; if the first edge is falling, the second edge is rising).

Use the following command

```
set_dft_signal -view view_type -type TCK -port TCK -timing variable
```

to specify clock waveforms if your semiconductor vendor's requirements differ from the default timing.

The `set_dft_signal` command has a period associated with it. That period has to be identical to the `test_default_period` value. If you change the value of one, you must be sure to check the value of the other.

```
set_dft_signal -view existing_dft -type TCK -port pad_tck \
-timing { 45 55}
```

set_dft_signal Default Clock Timing

When inferring a test protocol, the `dft_drc` command uses default values for the clock timing based on the clock type (see Table B-4) unless you explicitly specify clock timing with the `-timing` option to the `set_dft_signal` command.

Table B-4 Default Clock Timing

Clock type	First edge (ns)	Second edge (ns)
Edge-triggered	45.0	55.0
Master clock	30.0	40.0
Slave clock	60.0	70.0
Edge-triggered[a]	45.0	60.0
Master clock[a]	50.0	60.0
Slave clock[a]	40.0	70.0

a. Auxiliary-clock LSSD scan style only. In this scan style, the system clock is not used, the edge-triggered test clock (IH) is used for capture, and the scan-A master clock and scan-B slave clock are used for scan shift.

Note:
The polarity (rise or fall) of the first edge is determined from the technology library timing description for the sequential cells. The `dft_drc` command selects the polarity of the first edge so that the majority of the cells are triggered off the first edge. The polarity of the second edge is determined by the polarity of the first edge. For example, if the first edge is rising, the second edge is falling.

Specifying Clock Timing Attributes

If the default clock timing inferred by the `dft_drc` command does not meet your requirements, you can explicitly specify the clock timing using the `set_dft_signal` command. This command sets the following timing attributes on the clock ports you specify:

- `test_clock_rise_time`
- `test_clock_fall_time`

To verify the values of these timing attributes for all clock ports, use the `report_attribute -port` command.

Use the `set_dft_signal` command as follows

```
set_dft_signal -view existing_dft -type TCK \
               [-port port_list] \
               [-timing rise_time fall_time]
```

Note:
The `set_dft_signal` command has a period associated with it. That period has to be identical to the `test_default_period` value. If you change the value of one, you must be sure to check the value of the other.

The Test Protocol Clock Group

The arguments to `set_dft_signal` are the same as the values specified in the statements that make up the test protocol clock group. The `clock_port_list` argument becomes the value of the sources statement in the test protocol clock group. The rise argument becomes the value of the rise argument in the waveform statement in the test protocol clock group. The fall argument becomes the value of the fall argument in the waveform statement in the test protocol clock group. The `period_value` argument becomes the value of the period statement in the test protocol clock group.

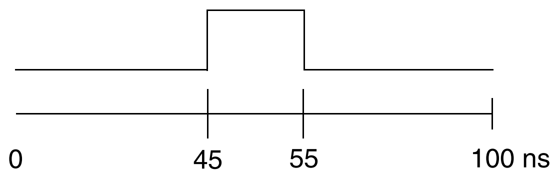
Specifying the Clock Period

If you do not specify the timing in the `set_dft_signal` command, the default value is the period value in the test protocol. If you specify the timing, it must be the same as the period value in the test protocol. If the timing specified in the `set_dft_signal` command differs from the period value in the test protocol, the `dft_drc` command displays an error message.

Multiplexed Flip-Flop Design Example

The clock timing for the multiplexed flip-flop design example shown in Figure B-2 is a positive pulse clock with a period of 100.0 ns. The rising edge occurs at 45.0 ns and the falling edge occurs at 55.0 ns. This clock waveform is shown in Figure B-2.

Figure B-2 Default Clock Timing for Multiplexed Flip-Flop Example



The `dft_drc` command automatically infers this clock timing, because the design contains an edge-triggered, active rising clock (see Table B-4 on page B-9). You can explicitly specify this clock waveform using the following `set_dft_signal` command:

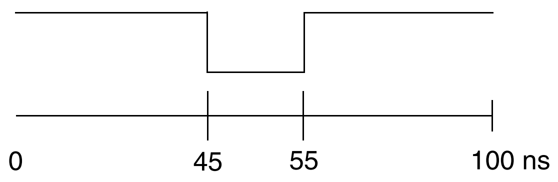
```
dc_shell> set_dft_signal -view existing_dft -type CLK \
           -timing {45.0 55.0}
```

If a return-to-one clock is required instead of the default clock, you can use the following `set_dft_signal` command:

```
dc_shell> set_dft_signal -view existing_dft -type CLK  \
           -timing {55.0 45.0}
```

Figure B-3 shows the waveform diagram.

Figure B-3 Return-to-One Waveform Diagram



Index