SpyGlass® MBIST Submethodology (for GuideWare 2017.12)

Version N-2017.12-SP2, June 2018

SYNOPSYS®

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Report an Error

The SpyGlass Technical Publications team welcomes your feedback and suggestions on this publication. Please provide specific feedback and, if possible, attach a snapshot. Send your feedback to *spyglass_support@synopsys.com*.

Contents

Preface

About This Book

The SpyGlass® MBIST methodology guide describes the flow for using the MBIST methodology.

Contents of This Book

The SpyGlass MBIST methodology guide has the following sections:

Section	Description
The Need for Automatic Insertion of MBIST at an Early Stage of a Design	The need for automatic insertion of MBIST at an early stage of a design

Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	`OUT <= IN;`
Object names	`OUT`
Variables representing objects names	*<sig-name>*
Message	Active low signal name '<sig-name>' must end with _X.
Message location	OUT <= IN;
Reworked example with message removed	OUT_X <= IN;
Important Information	**NOTE:** This rule...

The following table describes the syntax used in this document:

Syntax	Description
[ ] (Square brackets)	An optional entry
{ } (Curly braces)	An entry that can be specified once or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
. . . (Horizontal ellipsis)	Other options that you can specify

The Need for Automatic Insertion of MBIST at an Early Stage of a Design

Overview

The SpyGlass DFT MBIST product automates the insertion of Memory Built-In Self-Test (MBIST) IP into a design at the RTL level. The product also works at the gate level; however, working at the RTL level has many advantages that are crucial in leading-edge SoC designs.

Inserting MBIST at the RTL level reduces the cost of functional verification of the BIST structures. In addition, the synthesis optimization/early floor planning considers the MBIST logic, resulting in much more predictable quality of results and reduced iterations between the RTL-level and gate-level implementations.

Who this Document is for

The document is useful for both novice and advanced users of SpyGlass.

What You Need to have and know to Use this Document

You are expected to have basic knowledge of SpyGlass operations.

The SpyGlass DFT MBIST product requires a clean design into SpyGlass. Please refer to the *SpyGlass Design Read Methodology* document for details. The flow is based on Tcl environment available for SpyGlass operations and therefore the user is referred to the *SpyGlass Tcl Shell Interface User Guide* for more details. The user is also referred to the *SpyGlass DFT MBIST Tcl Flow Reference Guide* for the details of MBIST specific commands.

What this Document is About

This document introduces a method to run the SpyGlass DFT MBIST product.

Tool Versions

- SpyGlass Version: Version N-2017.12-SP2
- SpyGlass DFT MBIST Version: Version N-2017.12-SP2
- GuideWare: 2017.12

References

- SpyGlass Explorer User Guide
- SpyGlass Explorer Reference Guide
- SpyGlass Tcl Shell Interface User Guide
- SpyGlass Design Read-In Methodology
- SpyGlass GuideWare Reference Methodology User Guide
- SpyGlass MBIST Tcl Flow Reference Guide

MBIST Overview

This section describes the basic goals for the MBIST insertion process, various generic components, and restrictions imposed by the technology.

Generic MBIST Components

The SpyGlass DFT MBIST product is designed to work with various BIST components, some of which are as follows:

- **BIST Engines**: BIST engines are hardware components that generate patterns to apply to the memories. They can be either dedicated to a memory or assigned to serve multiple memories.

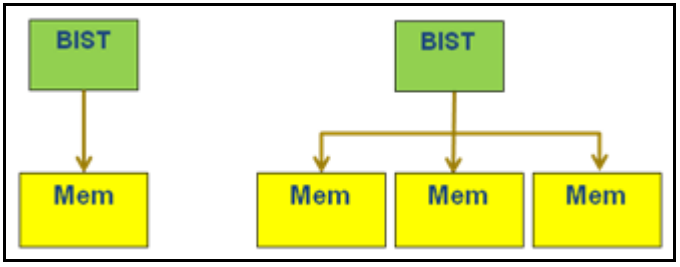


FIGURE 1. BIST Engines

- **BIST Controllers**: BIST controllers control various BIST engines. A hierarchy of BIST controllers, that is, components at various levels of hierarchies, is supported. Such controllers can be connected in chains with memories or other controllers.

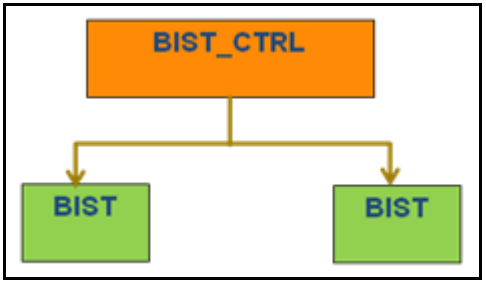


FIGURE 2. BIST Controllers

- **FUSE Controllers**: FUSE controllers detect a 'defective' row/column in a memory and help replace it with redundant rows/columns inside the same memory. A hierarchy of FUSE controllers, that is, components at various levels of hierarchies, is supported. Such controllers can be connected in chains with multiple memories or other controllers.

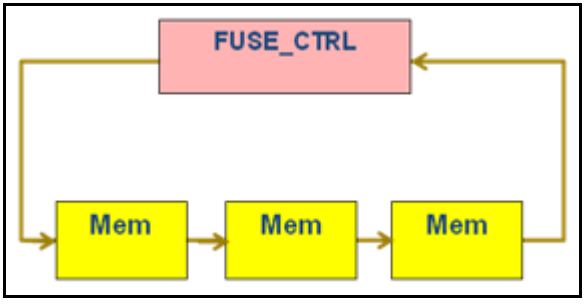


FIGURE 3. FUSE Controllers

- **JTAG controllers or other new instances**: A standard TAP controller (IEEE 1149.1) is inserted and then the MBIST control circuits are connected to this TAP controller for initiating MBIST and collecting the status from the BIST operation on memories. New blocks, that is, blocks that are not part of the original RTL design, can be inserted in various custom connections in order to facilitate MBIST activity.

Restrictions Imposed by the Technology

The architecture choice may be restricted by the specific technology. Examples of such restrictions are as follows:

- A specific shared BIST engine may be restricted to a limited number of memories.
- A specific FUSE controller may be restricted to no more than 32 memories in a chain.

Final View

The SpyGlass DFT MBIST product works in guidance with the vendor's library, applies technology restrictions, and generates a BIST-inserted design. A representative view of the post-BIST design is shown in the following figure:

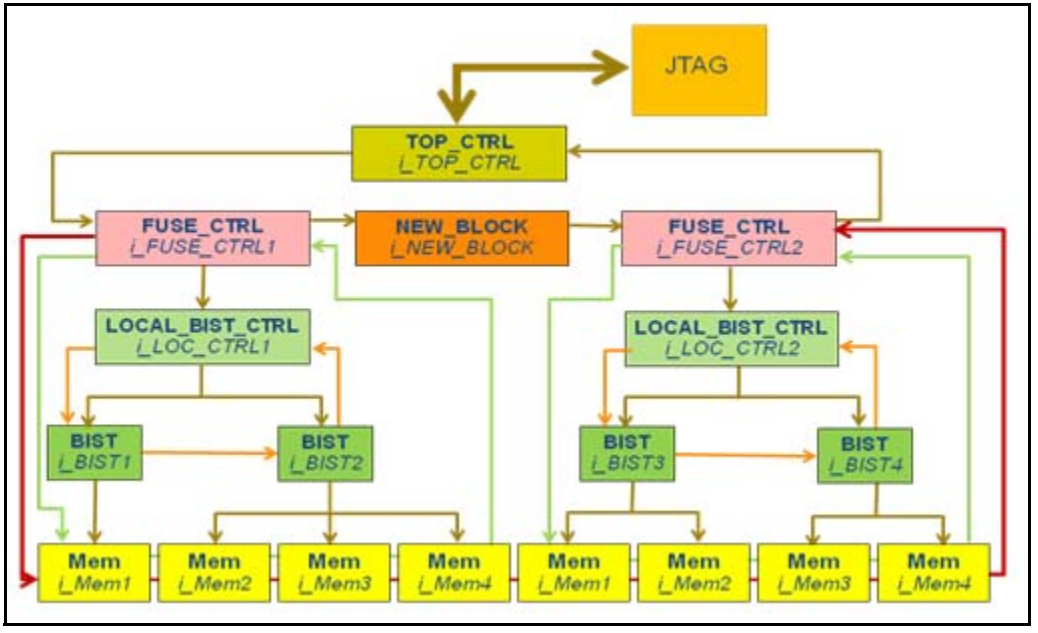


FIGURE 4. A post-BIST Design

Note that in the above figure, the memories in the original RTL exist at various levels of hierarchy. In addition, the other components are inserted

as part of MBIST insertion. Here, the designer selects the insertion level.

Vendor Data and User Data

The SpyGlass DFT MBIST product works with technology-specific data provided by the vendor. The designer provides ‘design-dependent’ data that refers to the vendor data and uses the SpyGlass DFT MBIST product. Thus, the use model can be represented by the following diagram.

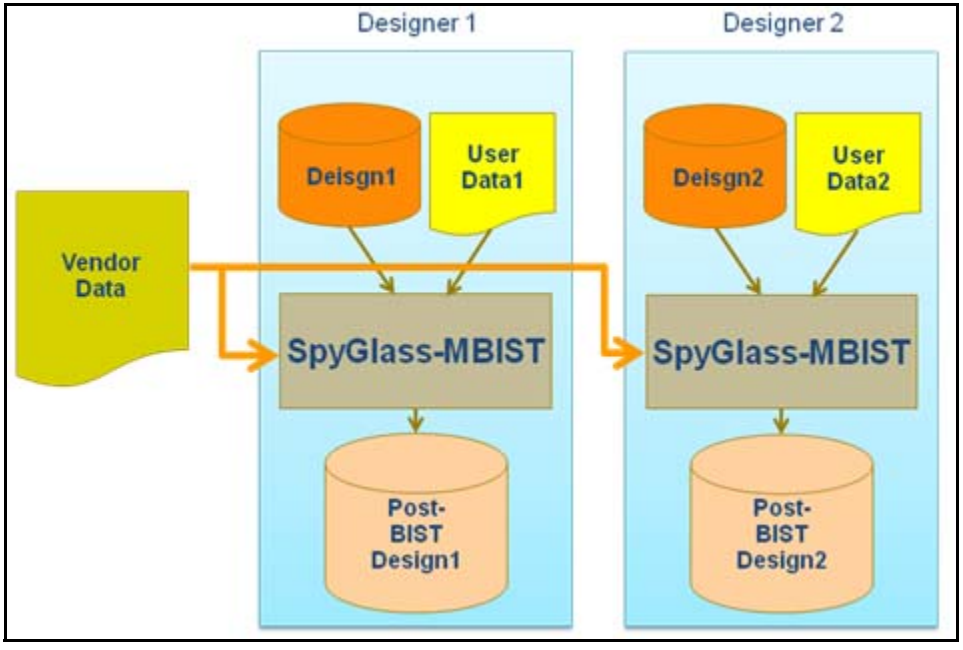


FIGURE 5. Vendor data and user data

Vendor data is design-independent and created by the vendor with technology restrictions in mind. This data will be used in any design targeted for the vendor’s technology.

User data is design-dependent and must be available for each design. The designer, while creating the user data, should have the knowledge of the vendor data and should understand how the vendor information can be used to achieve the MBIST structure. In the following section, we will

describe the essential structure of the user data file.

Approach for Inserting MBIST into a Design

This section describes the approaches/activities involved as part of MBIST insertion. More details about the specific instructions and variations are described in the *Steps for Inserting MBIST into a Design* section.

Overview

The following figure illustrates the recommended flow to perform MBIST insertion into a design by using the SpyGlass DFT MBIST product.

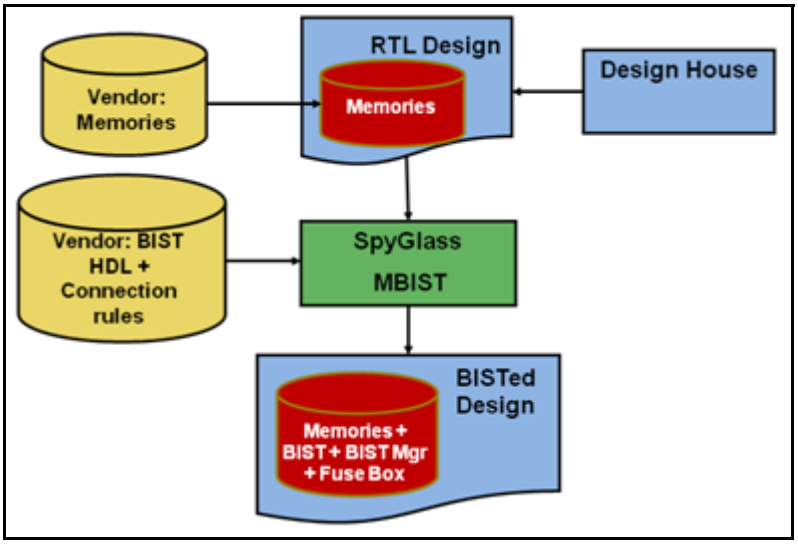


FIGURE 6. Use model of the SpyGlass DFT MBIST product

The SpyGlass DFT MBIST product provides an automated way to insert any vendor-specific BIST components into the designer's RTL. The SpyGlass DFT MBIST product is not tied to any vendor house so that the designer has the flexibility to work with any vendor of his choice.

Stages during SpyGlass DFT MBIST Insertion

This section gives a brief overview of each stage of SpyGlass DFT MBIST insertion.

The SpyGlass DFT MBIST insertion is divided into the following stages:

- *Creating a Vendor File*
- *Collecting Design Data and Preparing for MBIST Run*
- *Creating a User Data File*
- *Verifying the Post-BIST Design*

The following two distinct steps are required before starting the MBIST insertion process:

1. Vendor file creation by the vendor (one time effort), and
2. User file creation by the designer (once for each design)

Creating a Vendor File

To create a vendor file, perform the following steps:

1. *Define classes*
2. *Specify connections*
3. *Disconnect existing connections*
4. *Specify chain connections*
5. *Combine signals through gates*

For detailed information on creating a vendor file using the SpyGlass DFT MBIST product, see *Creating a Vendor File*.

Defining Classes

All the components relevant for MBIST (for example memories, controllers, etc.) are classified in a few named ‘classes’. Then parent-child relationship is established among various class members. A sample scheme of class definition and their relationship is shown in the following diagram.

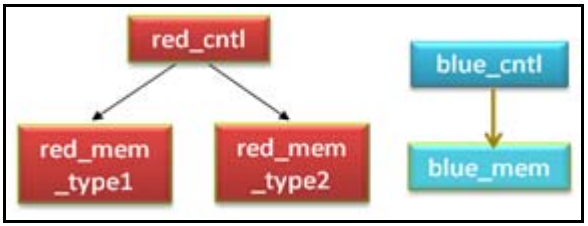


FIGURE 7. Defining classes

Five classes are defined: ‘red_cntl’, ‘red_mem_type1’, ‘red_mem_type2’, ‘blue-cntl’ and ‘blue_mem’. Two relationships are defined: ‘red_mem_type1’ and ‘red_mem_type2’ are the children of ‘red_cntl’. Similarly, class ‘blue_mem’ is a child of ‘blue_cntl’. Note, so far we have defined only the classes and their relationships. We are still not referring to any instances in a design (this will be done in the user file). The advantages of defining such classes and their relationship make it possible to define operations permissible only to such class members.

For defining classes in using the SpyGlass DFT MBIST product, see *Defining Classes*.

Specifying Connections

Once we have defined ‘classes’ and their ‘children’ we are ready to specify connections between them. An example of a ‘required’ connection in BIST-inserted design is for the BIST controller to supply the ‘bist_start’ control signal to the memory, as shown below:

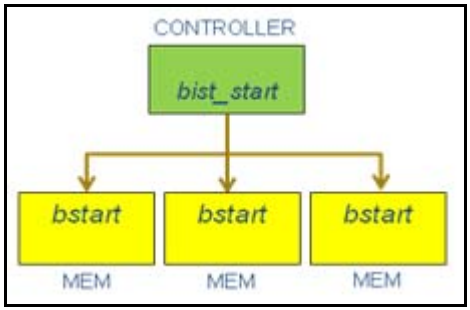


FIGURE 8. Specifying connections

In this scheme, both ‘CONTROLLER’ and ‘MEM’ have been defined as two classes, MEM being the child of CONROLLER. The vendor data should contain the necessary specification for such a connection. Moreover, as we will see in a following section, the connection could be a ‘broadcast, or ‘parallel’ connection.

For specifying connections using the SpyGlass DFT MBIST product, see *Specifying Connections*.

Disconnecting Existing Connections

In addition to establishing connections, it is often useful to disconnect existing connections. The activities may include simply disconnecting an existing connection, or disconnecting an existing connection from a driver (load) followed by establishing new driver (load).

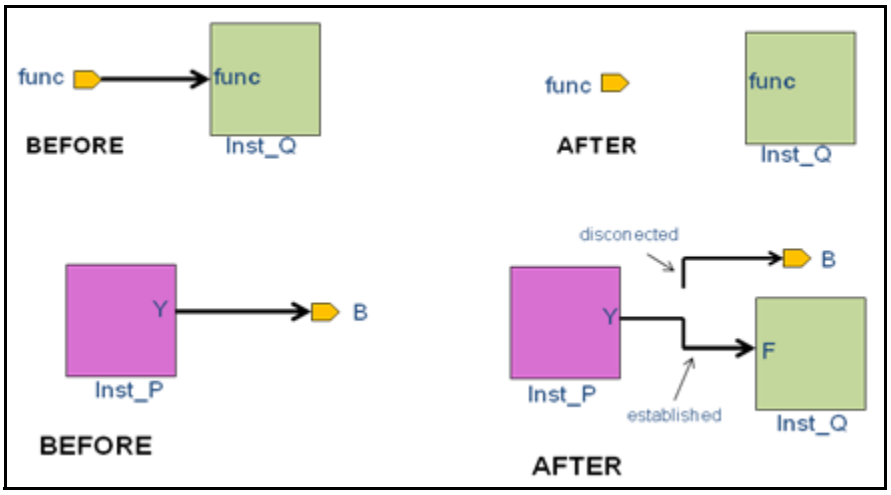


FIGURE 9. Disconnecting existing connections

For disconnecting existing connections using the SpyGlass DFT MBIST product, see *Disconnecting Existing Connections*.

Specifying Chain Connections

It may be necessary to specify required chain connections for various design components. As an example, the vendor may require a fuse controller to connect to a specific class of memories through a chain, as shown below:

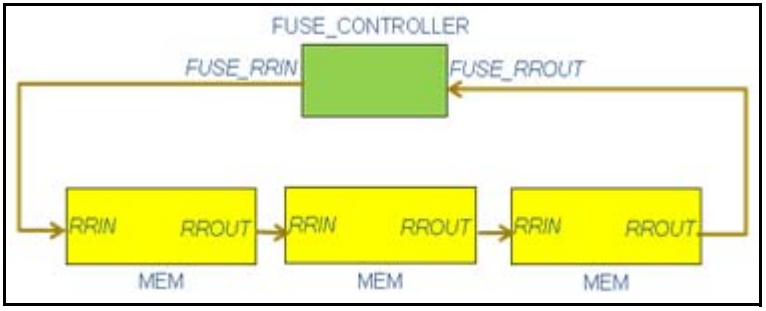


FIGURE 10. Specifying chain connections

In this example, FUSE_CONTROLLER and MEM should be defined as two different classes and MEM defined as a child of FUSE_CONTROLLER. Vendor data requires that a daisy chain be formed from the 'FUSE_RRIN' pin of the FUSE_CONTROLLER to the 'FUSE_RROUT' pin of the same class object through the pins 'RRIN' and 'RROUT' of MEM.

For specifying chain connections using the SpyGlass DFT MBIST product, see *Specifying Chain Connections*.

Combining Signals through Gates

It is often required that status signals from multiple BIST engines/ controllers are combined with gates like AND/OR/EXOR or technology library specific gates to generate global status signals. The following example shows that the 'bbad' (indicating failed test) signals from three BIST engines are combined through an OR gate and the resultant status signal is connected to a BIST controller.

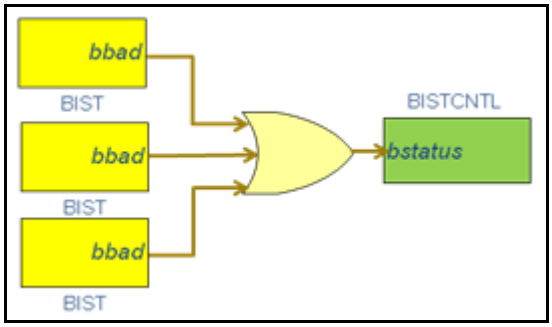


FIGURE 11. Combining signals through gates

In this case, BISTCNTL and BIST should be defined as two classes with BIST being the child of BISTCNTL. The vendor data specifies that the all the signals connected to the ‘bbad’ pin of the BIST instances are combined through an OR gate and the resultant signal drives the ‘bstatus’ pin of BISTCNTL instance. Note, the vendor may also optionally specify the 2-input gate to a specific cell from a technology library.

In summary, the vendor data

- indicates class definitions for memories and controllers
- indicates parent-child relationships among the classes
- indicates permissible connections between the classes
- indicates how chain connections are defined
- indicates how signals produced by class objects can be combined through generic/technology-specific ‘AND’/’OR’/’EXOR’ gates

We will describe more in detail about the specific instructions and variations of these specifications in the section *Steps for Inserting MBIST into a Design*.

Verification of the post-BIST design is critical to ensure the integrity of the insertion and connection process. For example, for the above example, one would like to check that the ‘bbad’ pins of the three BIST instances are indeed connected to the ‘bstatus’ pin of the instance of BISTCNTL. Details to accomplish verification will be presented in section XX.

For combining signals through gates using the SpyGlass DFT MBIST product, see *Combining Signals through Gates*.

Collecting Design Data and Preparing for MBIST Run

The designer is responsible for design-dependent data. It is mandatory that the designer is aware of the various memory instances present in the design. It is also the designer's responsibility to identify the association of memories with BIST engines. One needs to create the instance-specific association among all the BIST components.

As part of the BIST insertion process, the designer must find answers to the following questions:

- What are the memory instances in the design? How they are mapped to the vendor-specified classes?
- Which memory instances are to be associated with which BIST engine instances? Which classes the BIST engine instances belong to?
- Which BIST controller instances will control which BIST engine instances? Which classes the BIST controller instances belong to?

In summary, as the designer is getting ready to run the insertion step, the following design-specific information should be available. This examples assumes a certain set of BIST components to work with (this class hierarchy depends on the vendor).

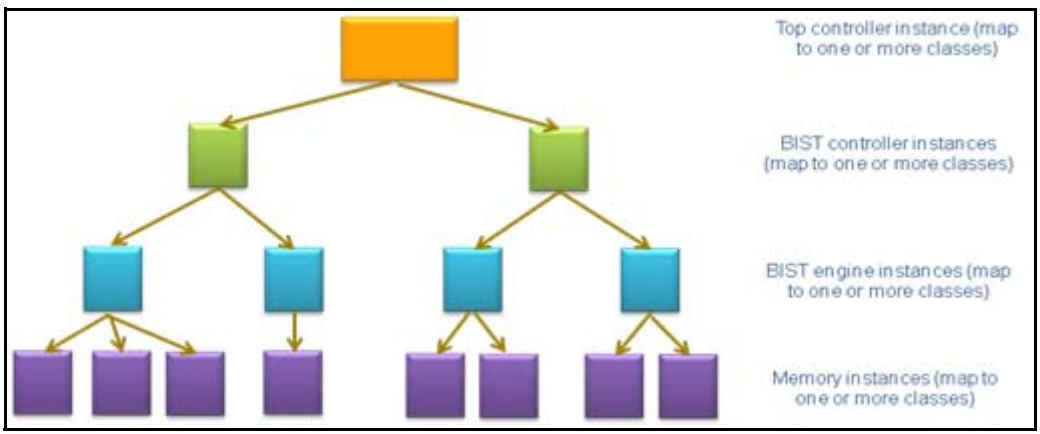


FIGURE 12. Collecting design data and preparing for MBIST run

For more information on how the SpyGlass DFT MBIST product can be used to extract data to help the designer prepare for the MBIST insertion, see

Collecting Design Data and Preparing for MBIST Run.

Creating a User Data File

To create a user data file, perform the following steps:

1. *Prepare the setup*
2. *Define the existing instances*
3. *Insert/replace and define new instances*
4. *Remove the existing instances*
5. *Establish custom connections*
6. *Insert MBIST*
7. (Optional) *Specify the location of modified design files*

The above steps are illustrated in the following diagram:

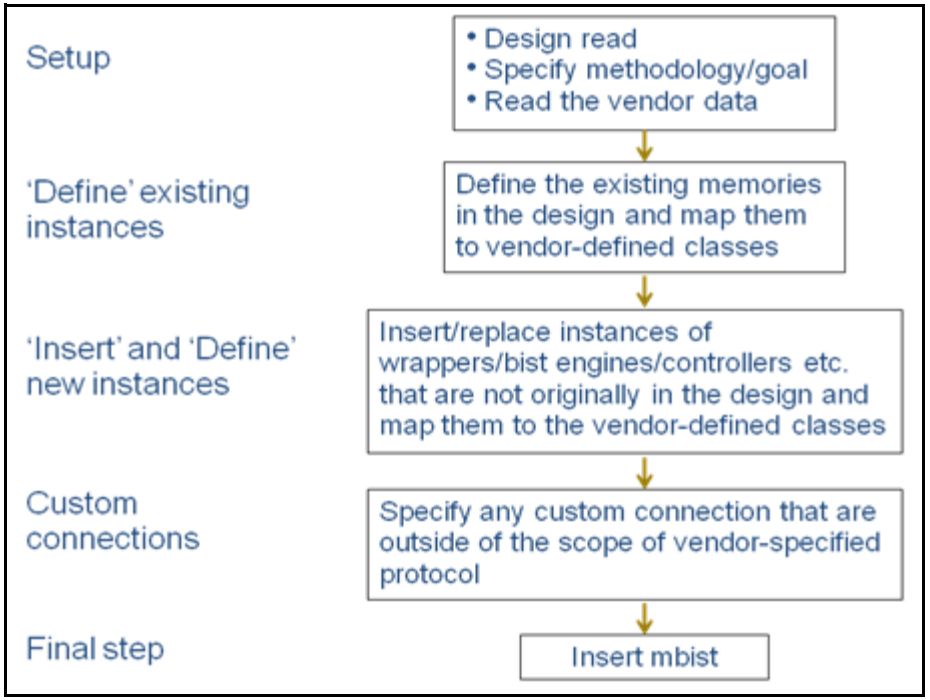


FIGURE 13. Steps for creating a user data file

For detailed information on creating a user data file using the SpyGlass DFT MBIST product, see *Creating a User Data File*.

Preparing the Setup

The SpyGlass DFT MBIST product works in an environment based on Tcl called 'sg_shell'. The vendor data and the user data are in Tcl format and can use all the standard Tcl scripting techniques.

To prepare the setup for running the SpyGlass DFT MBIST product, perform the following steps:

1. *Read the design*
2. *Read the BIST-related components*
3. *Specify the methodology and goal*

4. *Read the vendor data*
5. *Control the output format*

For detailed information on preparing the setup using the SpyGlass DFT MBIST product, see *Preparing the Setup*.

Reading the Design

The first step of the SpyGlass DFT MBIST product is to read the design. The design can be specified in Verilog, VHDL, or mixed language format. It is necessary to create a SpyGlass 'project', supply a list of RTL files, specify path to library directories containing libraries, specify the library files, and supply a list of technology libraries (in case of technology mapped netlist) in '.sglib' format. We will describe commonly encountered requirements in section *Creating a User Data File*. Refer to the 'design read' methodology in SpyGlass user guide.

Reading Compressed Files

SpyGlass DFT MBIST product reads compressed files and will generate compressed files if modifications to any of these files are produced.

Reading the BIST-Related Components

In order to be able to insert the BIST related components you need to read the RTL descriptions of them. Also, note that they are not part of the supplied design yet. They will become part of the design after the BIST insertion. However, at this stage you can supply the description of the BIST components through the usual design read process, like – reading the RTL models explicitly, supplying them through `set_option v` command or `set_option y` command as in the Verilog paradigm. This process will be illustrated in the 'step-by-step' section of this document.

Specifying the Methodology and Goal

This is an essential requirement for SpyGlass operation to specify a 'Methodology' and 'Goal'. For the MBIST purpose we need to select the '$SPYGLASS_HOME/Methodology/MBIST_DFT' methodology and 'mbist_dft' goal.

Reading the Vendor Data

At this step, we read the vendor data or the technology information through a TCL read command.

Controlling the Output Format

During this setup step, we can control the nature of the output data produced by the BIST insertion steps. This consists of the following:

- How the modified module changes its name
- How the modified design file name changes (control one or both of the prefix and suffix of the file name)
- How any modified signal changes its name

Example

Consider the following modules:

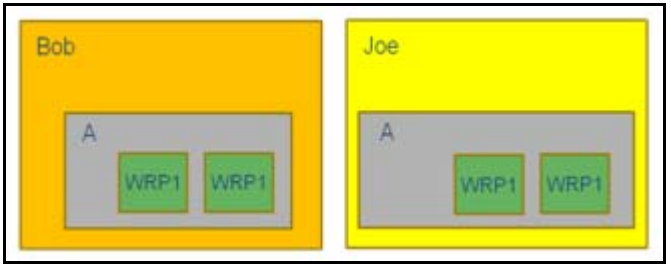


FIGURE 14. Original module

After BIST insertion, the module names are modified as follows:

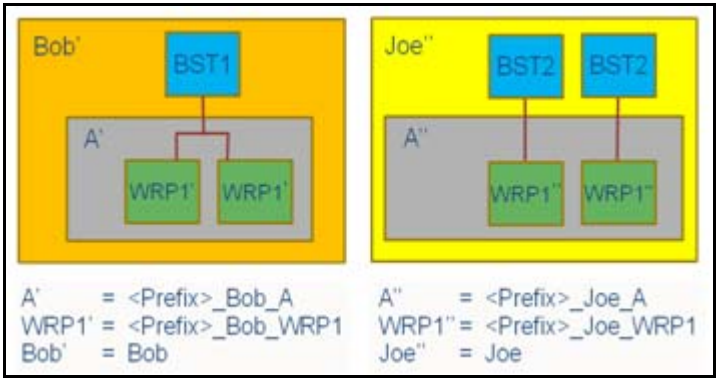


FIGURE 15. Modified module after BIST insertion

NOTE: *The name of the modified top-level module is never changed. This is because at a higher level of the design, the modified top-level module is continued to be referred by the same name.*

SpyGlass, however, maintains the uniqueness of any generated name. This means that whenever there is a name clash, the name is ‘uniquified’. We will find out more details of this user control steps in the “step-by-step” section.

Defining the Existing Instances

The user needs to know the memory instances (with the hierarchical instance names) existing in the design. The vendor data would have already defined specific ‘classes’ for these memories. At this stage, the user identifies such existing instances and maps them to the vendor-specified ‘classes’.

For this purpose and for inserting new instances, the user may need to maintain this instances) in a convenient format such as a table as shown below:

TABLE 1 Memory instances

Sl #	Memory instance	Memory type	Controller instance name	Controller name
1	i_distributed.i_big_mem.i_mem1	MEM_2048x32m8_LTB1	i_distributed.i_big_mem.i_mem1_ctrl	COLLAR_6_Dedicated
2	i_distributed.i_big_mem.i_mem2	MEM_2048x32m8_LTB1	i_distributed.i_big_mem.i_mem2_ctrl	COLLAR_7_Dedicated
3	i_distributed.i_big_mem.i_mem3	MEM_2048x32m8_LTB1	i_distributed.i_big_mem.i_mem3_ctrl	COLLAR_8_Dedicated
4	i_shared.i_big_mem.i_mem1	MEM_2048x32m8_LTB1	i_shared.i_big_mem.CONTROLLER_4	COLLAR_10_Parallel
5	i_shared.i_big_mem.i_mem2	MEM_2048x32m8_LTB1	i_shared.i_big_mem.CONTROLLER_4	COLLAR_10_Parallel
6	i_shared.i_big_mem.i_mem3	MEM_2048x32m8_LTB1	i_shared.i_big_mem.CONTROLLER_4	COLLAR_10_Parallel
7	i_distributed.i_small_mem.i_mem0.i_mem1	MEM_256x8m8_LTB1	i_distributed.i_small_mem.i_mem0.i_mem1_ctrl	COLLAR_3_Dedicated
8	i_distributed.i_small_mem.i_mem0.i_mem2	MEM_256x8m8_LTB1	i_distributed.i_small_mem.i_mem0.i_mem2_ctrl	COLLAR_4_Dedicated
9	i_distributed.i_small_mem.i_mem3	MEM_256x8m8_LTB1	i_distributed.i_small_mem.i_mem3_ctrl	COLLAR_5_Dedicated
10	i_red1.i_mem2	MEM_4096x32m8_LRTB2	CONTROLLER_3	COLLAR_2_ParallelR
11	i_red2.i_mem2	MEM_4096x32m8_LRTB2	CONTROLLER_3	COLLAR_2_ParallelR
12	i_red3.i_mem2	MEM_4096x32m8_LRTB2	CONTROLLER_3	COLLAR_2_ParallelR
13	i_red1.i_mem1	MEM_4096x32m8_LRTB2	CONTROLLER_1	COLLAR_1_ParallelR
14	i_red2.i_mem1	MEM_4096x32m8_LRTB2	CONTROLLER_1	COLLAR_1_ParallelR
15	i_red3.i_mem1	MEM_4096x32m8_LRTB2	CONTROLLER_1	COLLAR_1_ParallelR
16	i_shared.i_small_mem.i_mem0.i_mem1	MEM_256x8m4_L	i_shared.i_small_mem.CONTROLLER_2	COLLAR_9_Parallel
17	i_shared.i_small_mem.i_mem0.i_mem2	MEM_256x8m4_L	i_shared.i_small_mem.CONTROLLER_2	COLLAR_9_Parallel
18	i_shared.i_small_mem.i_mem3	MEM_256x8m4_L	i_shared.i_small_mem.CONTROLLER_2	COLLAR_9_Parallel
19	i_mem1	MEM_512x16m4_LB4	i_mem1_DPREG_ctrl	COLLAR_11_Dedicated
			CONTROLLER_5	TOP_BIST_CTRL

Note, the user maintains a list of memories and their instance names (as shown in col. 2 and 3 in the above table) in the design. The memory types are typically defined as ‘classes’ in the vendor data. At this step, the user maps the instances (with hierarchical names) to the pre-defined ‘classes’.

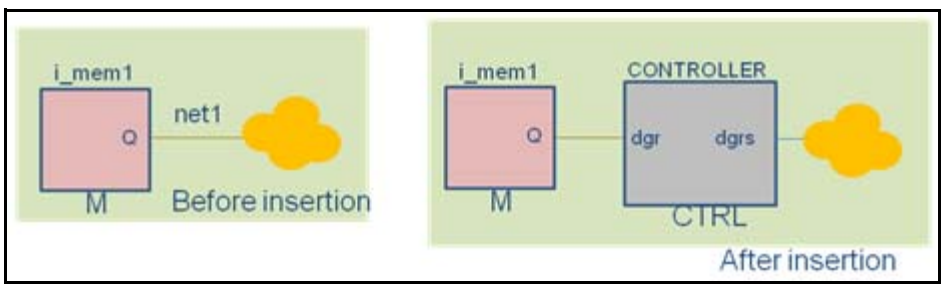
Such a table is not required by the SpyGlass DFT MBIST product. However, it may be useful to facilitate supplying the correct directions to drive the tool.

For defining the existing instances using the SpyGlass DFT MBIST product, see *Defining the Existing Instances*.

Inserting/Replacing and Defining New Instances

Inserting New Instances

At this step, user indicates which new instances are to be inserted. The new instances are the BIST engines, controllers etc. Examples of inserting controllers (‘dedicated’ and ‘shared’ types) are as shown in the following diagrams.



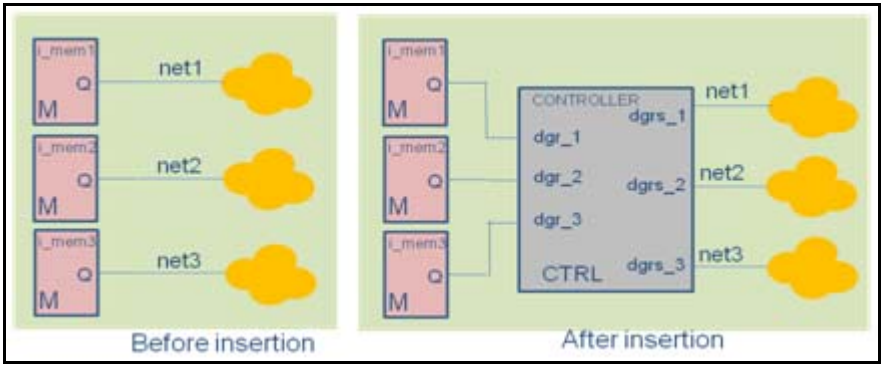


FIGURE 16. New instances inserted during MBIST insertion

As shown in the diagrams above, the controller instance is inserted as intended by the user. The connections between the memory and the controller instance are guided by the connection rules indicated in the vendor data file.

Inserting New Instances of Parameterized Modules

It is often convenient to define instances of parameterized cells where the designer explicitly supplies the parameter values for insertion. See the *Steps for Inserting MBIST into a Design* section for details.

Replacing Existing Instances

Special situations may require replacing an existing instance by another instance. Consider the case as depicted in the following diagram. After BIST insertion, the design will have additional connections (extra pins on the new instance). Thus, it becomes necessary to ‘replace’ the memory instance by the instance of the ‘WRAPPER’.

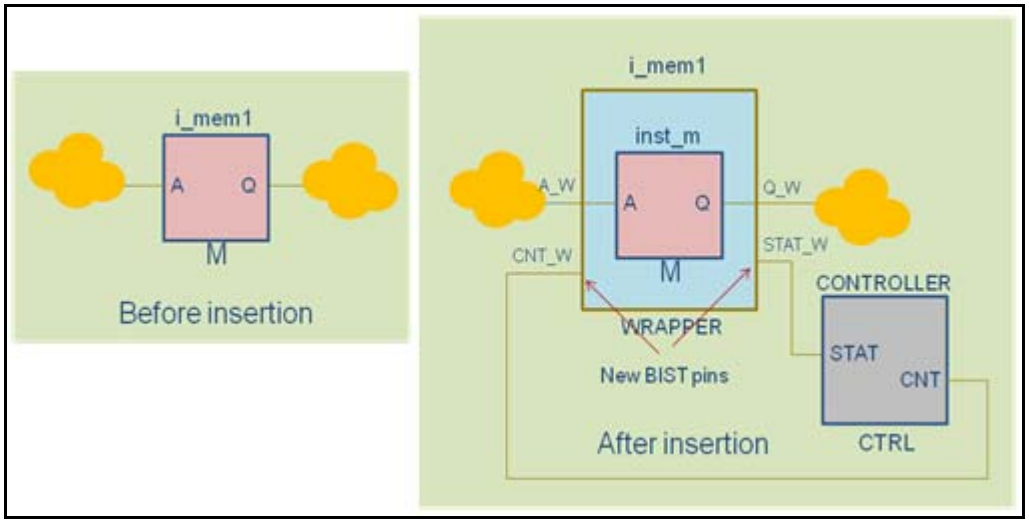


FIGURE 17. Existing instances replaced during MBIST insertion

The user must create a 'class' for the wrapper (class declaration for memory is not needed here). The connection between the wrapper instance and the controller instance will be governed by connection rules specified in the vendor data file.

This process should also be able to accommodate the case where all the ports of the memory do not have corresponding 'substitute' on the wrapper. These memory ports are originally unconnected in the design. One possibility is they are consumed by an internal scan chain inside the wrapper, as shown in the following diagram. You will notice that the pin 'Y' of the memory is connected to an internal logic inside the wrapper and does not have a corresponding pin map on the wrapper. See the 'step-by-step' section for details of how this can be accomplished.

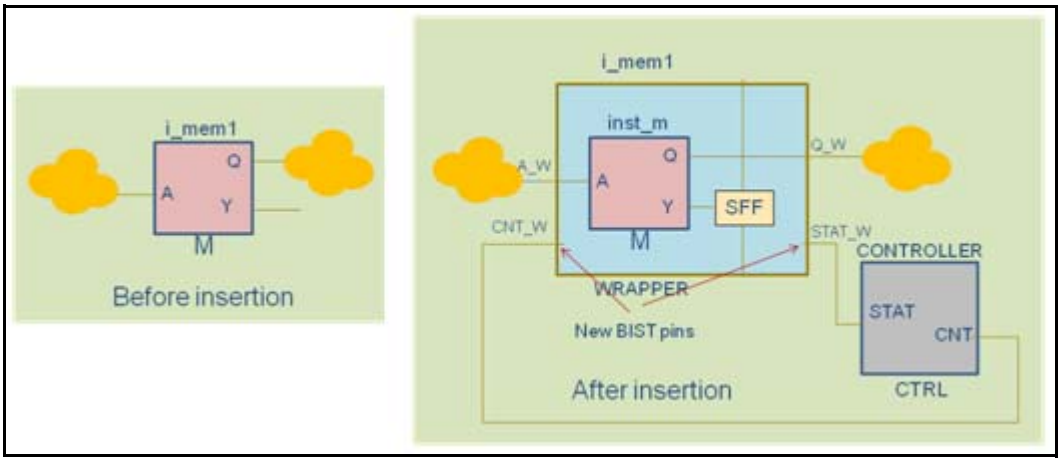


FIGURE 18. Existing instances replaced during MBIST insertion

For inserting/replacing and defining new instances using the SpyGlass DFT MBIST product, see *Inserting/Replacing and Defining New Instances*.

Removing the Existing Instances

Situations exist where it is necessary to remove existing instances. For example, placeholders in the original design may become orphans after the MBIST insertion. The designer may explicitly indicate to remove instances of specific cells in the user data file.

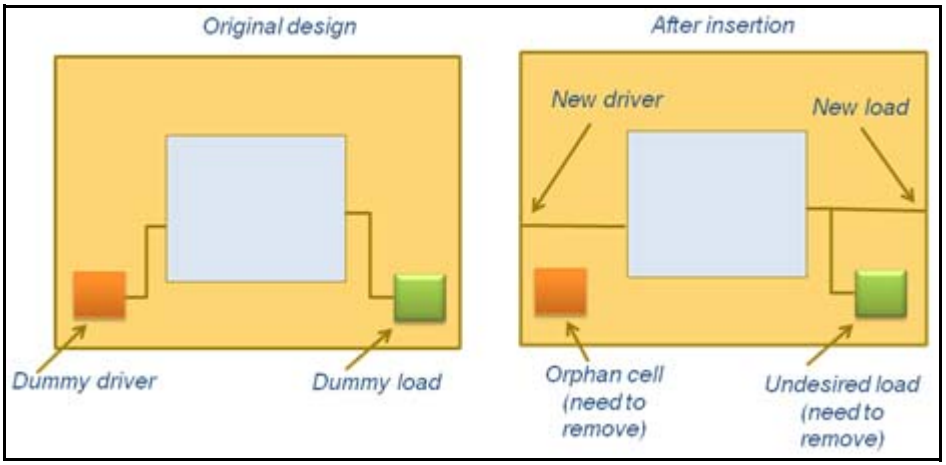


FIGURE 19. Existing instances removed during MBIST insertion

For removing the existing instances using the SpyGlass DFT MBIST product, see *Removing the Existing Instances*.

Establishing Custom Connections

Most of the connections/chains between the memories and the BIST components will be established by the rules provided in the vendor data file. However, there may still be the requirement of creating ‘custom’ connections that are outside of the scope of vendor specification.

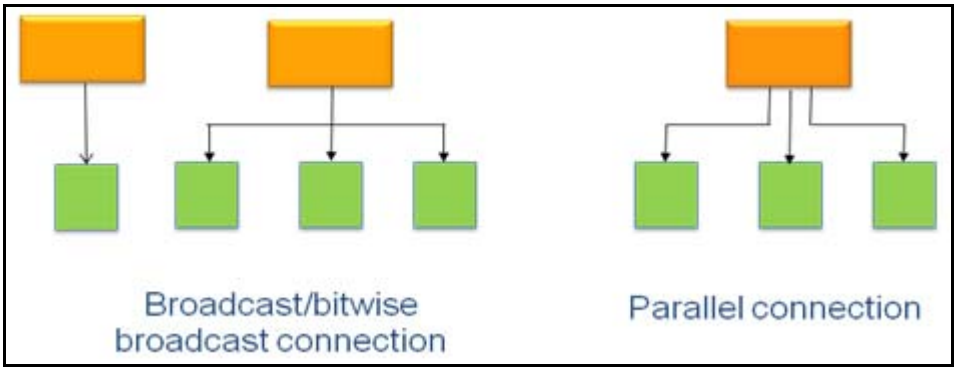


FIGURE 20. Establishing custom connections

The diagram above shows various connections between instances in the design.

MBIST may be used to establish any arbitrary point-to-point connection such as:

- Connecting a pin of an instance (newly inserted or existing) to a primary I/O port
- Connecting a pin of an instance to GND or VCC
- Overriding a vendor-directed connection with a custom connection

For establishing custom connections using the SpyGlass DFT MBIST product, see *Establishing Custom Connections*.

Inserting MBIST

This is the final step of MBIST insertion. With the vendor information file and the user's intention in the user data file, the tool generates the modified description of the design by inserting the MBIST components and making all the prescribed connections.

For inserting MBIST into the design using the SpyGlass DFT MBIST product, see *Inserting MBIST: The mb_insert Tcl Command*.

Specifying the Location of Modified Design Files

The modified design files are deposited in a specific directory. However, the designer can optionally collect all the modified design files in a user-specified location.

For specifying the location of modified design files using the SpyGlass DFT MBIST product, see *Specifying the Location of Modified Design Files*.

Verifying the Post-BIST Design

To verify the post-BIST design, perform the following steps:

1. *View the modified design*
2. *Verify the AND/OR/EXOR logic tree*

Viewing the Modified Design

This is a convenient way of inspecting that the modified RTL is indeed free of syntax errors (syntactically correct Verilog/VHDL/SystemVerilog) and is synthesizable. We recommend that the designer reads the modified design into SpyGlass, runs a generic GuideWare goal (refer to the SpyGlass GuideWare Methodology manual for more information) and view the modified design in the schematic. The designer will be able to view the inserted BIST components, various inter-block connections, and the desired chain connections. We will describe the steps in detail in section 4.

Verifying the AND/OR/EXOR Logic Tree

The BIST insertion process may be required to logic trees with desired particular logic function. However, ‘sneak paths’ may be created with either incorrect logic functions or an inappropriate source, as shown in the following diagram.

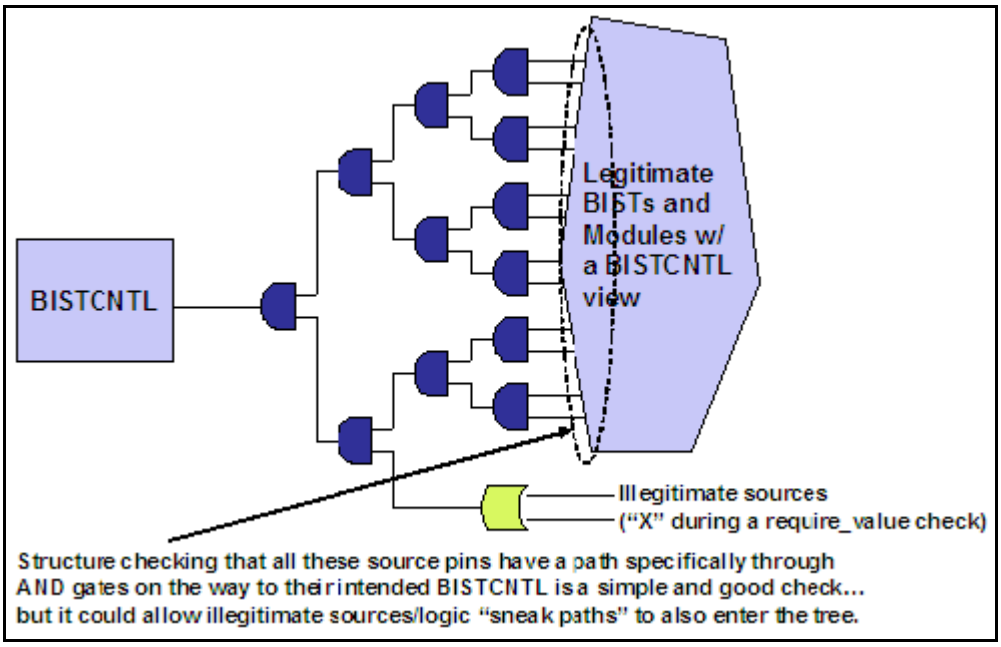


FIGURE 21. Verifying the AND/OR/EXOR logic tree

It is recommended that the designer run checks through SpyGlass to flag such situations.

For verifying the post-BIST design using the SpyGlass DFT MBIST product, see *Verifying the Post-BIST Design*.

Steps for Inserting MBIST into a Design

This section describes the steps for using SpyGlass DFT MBIST product. For a description of the Tcl commands, refer to the *SpyGlass MBIST Tcl Flow Reference Guide*. This section will illustrate the most common use model. The description in this section corresponds to the approach discussed in section *Approach for Inserting MBIST into a Design*.

Creating a Vendor File

This section elaborates how the various steps described in the Approach section can be achieved through Tcl commands in the vendor data file. The intention is to illustrate the most common scenarios. For more details and the exact syntax of the Tcl commands, the user should refer to the *SpyGlass MBIST Tcl Flow Refrence Guide*.

Defining Classes

Components are given a class definition with the command '`mb_define_class`'. This command is also used to describe parent-child relationships.

The class hierarchy structure in our example can be specified as follows:

```
mb_define_class –class red_mem_type1
mb_define_class –class red_mem_type2
mb_define_class –class blue_mem
mb_define_class –class red_cntl –children {red_mem_type1
red_mem_type2}
mb_define_class –class blue_cntl –children blue_mem
```

Specifying Connections

Pin-to-pin connections can be established through the '`mb_configure_connect_net`' command. Connections can be one-to-one or one-to-many.

There are following types of connections:

- **Broadcast connections**

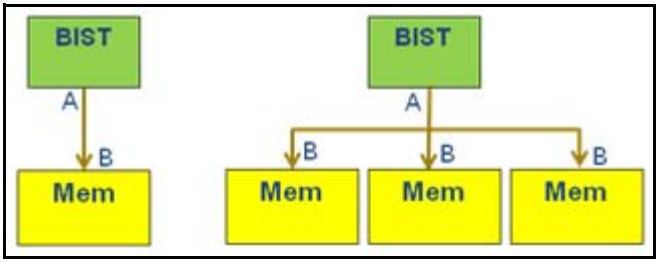


FIGURE 22. Specifying broadcast connections

In this example, scalar pin ‘A’ of an instance of class ‘BIST’ must be connected to the scalar pin ‘B’ of one or more instances of class ‘Mem’. This is accomplished through the command:

```
mb_configure_connect_net -from_class BIST -from_pin A \
         -to_class Mem –to_pin B  -broadcast
```

- **Bit-wise broadcast connections**

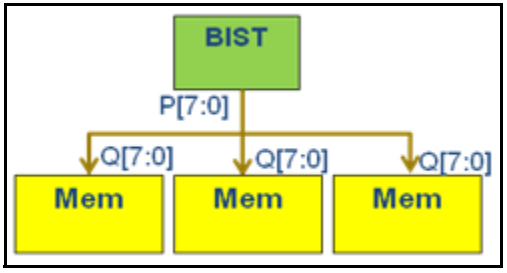


FIGURE 23. Specifying bit-wise broadcast connections

In this example, vector pin ‘P’ of an instance of class ‘BIST’ is connected to the similar sized vector pin ‘Q’ of one or more instances of class ‘Mem’. This is accomplished through the command:

```
mb_configure_connect_net -from_class BIST -from_pin P \
         -to_class Mem –to_pin Q  -bitwise_broadcast
```

- **Parallel connections**

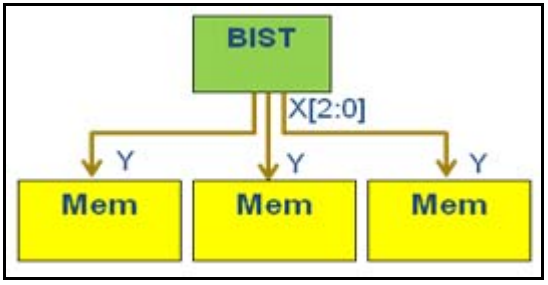


FIGURE 24. Specifying parallel connections

In this example, vector pin 'X' of an instance of class 'BIST' is connected to the scalar pin 'Y' of multiple instances of class 'Mem'. This is accomplished through the following command:

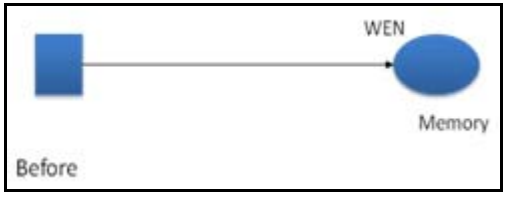
```
mb_configure_connect_net -from_class BIST -from_pin X \
         -to_class Mem –to_pin Y  -parallel
```

Common Connection Requirements

This section describes some of the commonly required connections. Each case begins with a before and after example followed by a description of the associated commands.

- **Multiplexed connections**

  The multiplexed connection is used to switch between functional mode and BIST mode:



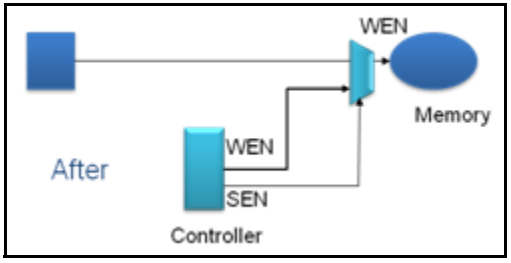


FIGURE 25. Multiplexed connections

This multiplex connection is made with the following:

```
mb_configure_connect_net –from_class CONTROLLER
         –from_pin WEN –to_class MEMORY –to_pin WEN
          –multiplex -select_enable SEN –active HIGH
```

- **Connections through inserted components**

  The following diagrams illustrate new connections because of component insertion:

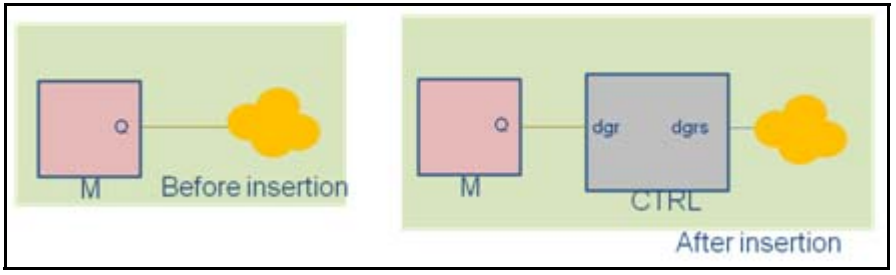


FIGURE 26. Connections through inserted components

After BIST insertion, an instance of class CTRL has been inserted and the above connections have been made. Q pin of M now drives the ‘dgr’ pin of CTRL and the ‘dgrs’ pin of CTRL drives the logic. This represents a connection ‘through’ another class object.

This can be accomplished through the following command:

```
# Vendor
mb_define_class -class M
```

```
mb_define_class -class CTRL -children M
# The following  indicates the legal connections
# between the memories and the controller
mb_configure_connect_net -from_class M -from_pin Q \
        -through_class  CTRL -in_pin dgr -out_pin dgrs
         -parallel
```

- **Connections from multiple components**

  A single insertion can require multiple connections as illustrated below.

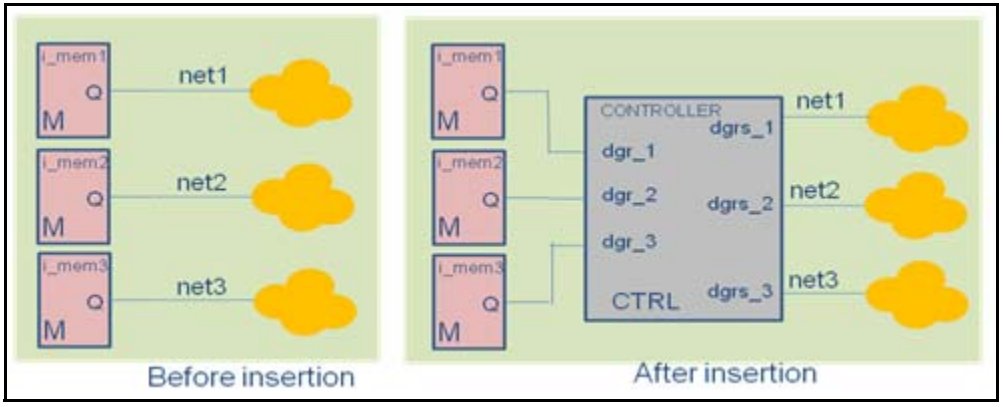


FIGURE 27. Connections from multiple components

In this case, a ‘shared’ controller has been inserted and appropriate connections between the memories and the controller are established. This can be accomplished by the connection rule:

```
# Vendor
mb_define_class -class M
mb_define_class -class CTRL -children M
# The following  indicates the legal connections
# between the memories and the controller
mb_configure_connect_net -from_class M -from_pin {Q}
   -through_class CTRL -in_pin {dgr_1 dgr_2 dgr_3}
   -out_pin {dgrs_1 dgrs_2 dgrs_3} -parallel
```

- **Connections from a single controller to multiple memories**

  In this case, each bus from the controller will be split and connected to

multiple pins of the same memory.

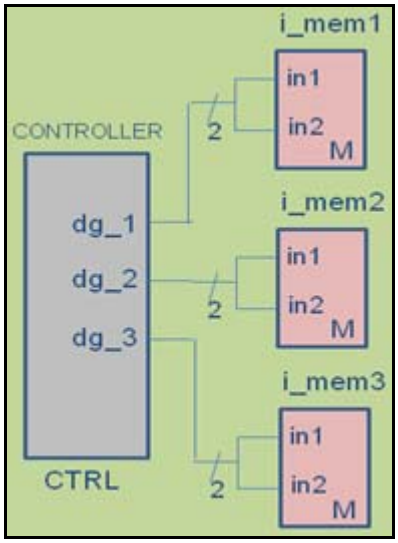


FIGURE 28. Connections from a single controller to multiple memories

This can be achieved with the following connection rule:

```
# Vendor
mb_define_class -class M
mb_define_class -class CTRL -children M
# The following  indicates the legal connections
# between the memories and the controler
foreach {parallel_ctrl memory} { CTRL  M} {
     foreach {frpin topin} {{dg_1 dg_2 dg_3} {in1 in2}} {
       mb_configure_connect_net \
          -to_class [list $memory $memory] -to_pin $topin \
          -from_class $parallel_ctrl -from_pin $frpin
           -parallel
   }
}
```

- **Connections to VCC or GND**

  A vendor directive can be used to connect a specific pin on all objects

belonging to a class to constant nets (VCC or GND), resulting into the following connections:

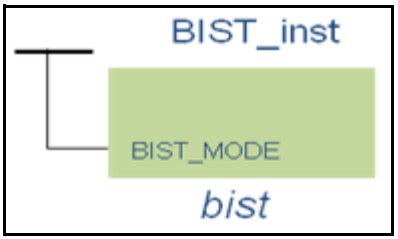


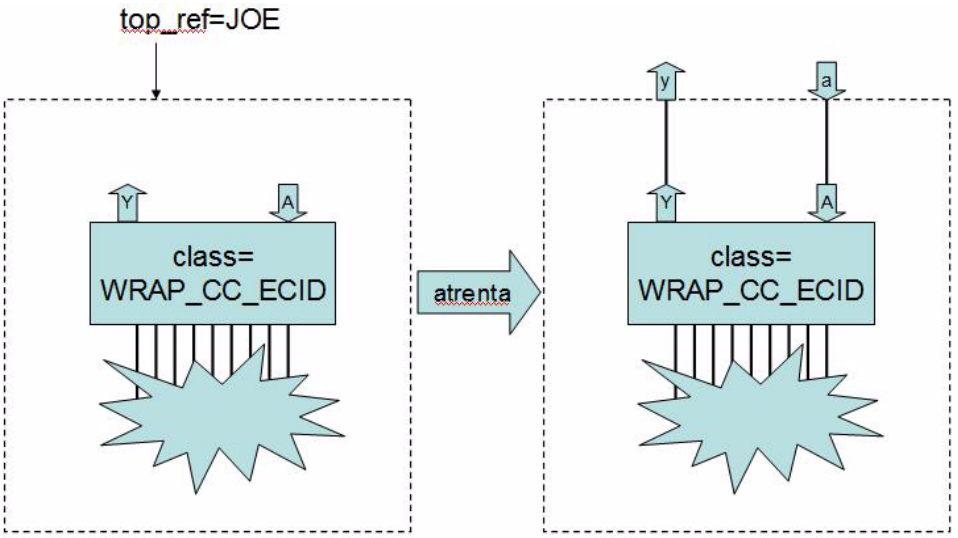
FIGURE 29. Connections to VCC or GND

The vendor directive can be specified as follows:

```
# Vendor rule
mb_configure_connect_net -from_class  TOP -from_pin 1 \
             -to_class BST -to_pin BIST_MODE -broadcast
```

- **Connections to top port**

  Generic class can be used to define connection with top-level ports in the vendor file.



This is accomplished using the following connection rule:

```
mb_configure_connect_net -from_class WRAP_CC_ECID \
-from_pin Y -to_top_port y
```

In this case, if output port, y, on top does not exist, a new output port by name y is created on top. Also, if connection is routed from the intermediate hierarchy and output port by name Y does not exist on the intermediate hierarchy, then new output port by name Y is created on the intermediate hierarchy.

```
mb_configure_connect_net -from_top_port a \
-to_class WRAP_CC_ECID -to_pin A
```

In this case, if input port, a, on top does not exist, a new input port by name a is created on top. Also, if connection is routed from the intermediate hierarchy and input port by name A does not exist on the intermediate hierarchy, then new input port by name A is created on the intermediate hierarchy.

Disconnecting Existing Connections

Existing connections can be removed with the 'mb_configure_connect_net' command.

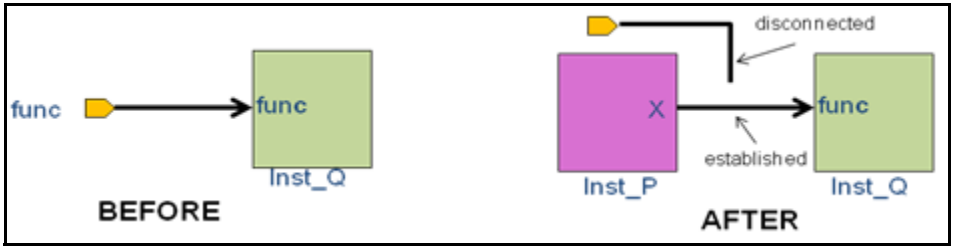
- **Case 1**: Disconnect existing driver and do not establish a new connection to a driver


```
mb_configure_connect_net        \
              -to_class Q -to_pin func -disconnect
```

- **Case 2**: Disconnect existing load and do not establish a new connection to load


```
mb_configure_connect_net          \
                -from_class P -from_pin func -disconnect
```

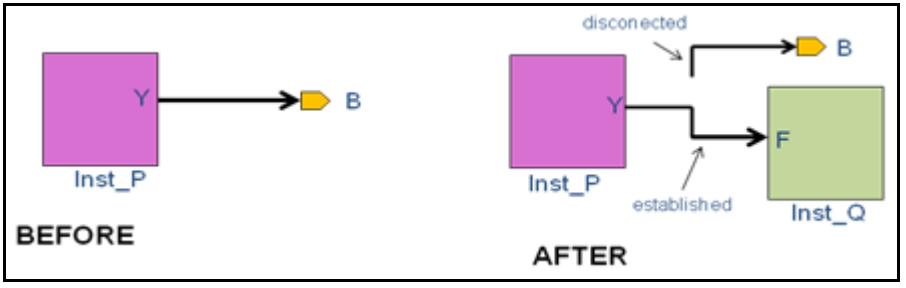
- **Case 3**: Disconnect existing driver and establish new connection to a driver




```
mb_configure_connect_net          \
                -from_class P –from_pin X \
                 -to_class Q -to_pin func  -broadcast
```

NOTE: *In the case above the '-disconnect' switch is not needed. SpyGlass automatically disconnects the existing driver before establishing a 'new' driver.*

- **Case 4**: Disconnect load and establish new load




```
mb_configure_connect_net      \
              -from_class P -from_pin Y \
              -to_class Q –to_pin F     \
```

```
-broadcast  -disconnect
```

Specifying Chain Connections

Chains through various class objects can be achieved through the ‘mb_configure_connect_chain’ command. Consider the following scenario:

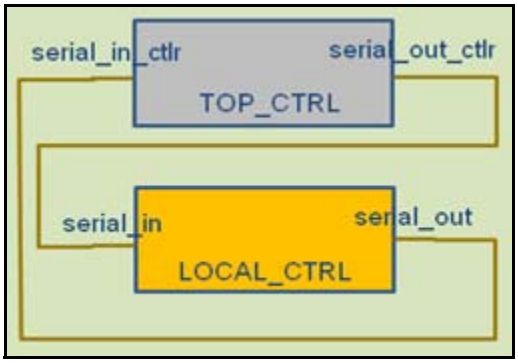


FIGURE 30. Specifying chain connections

This requires a chain connection through two class members – TOP_CTRL and LOCAL_CTRL. The start of the chain is the ‘serial_out_ctrl’ pin and the end point is the ‘serial_in_ctrl’ pin of TOP_CTRL.

This configuration is achieved through the following vendor data file command:

```
#Vendor
mb_define_class  -class LOCAL_CTRL
mb_define_class –class TOP_CTRL –children  LOCAL_CTRL
mb_configure_connect_chain -from_class TOP_CTRL\
         -start_pin serial_out_ctlr -end_pin serial_in_ctlr \
          -through_class LOCAL_CTRL \
          -in_pin serial_in \
           -out_pin serial_out
```

Note that the above vendor data merely specifies a configuration rule. In the actual design more than one instance of the ‘child class’ can be connected in a serial chain with an instance of the ‘parent’ class

Combining Signals through Gates

Signals generated by several class objects can be combined through the ‘mb_configure_connect_gate’ command. Permissible combination gate types are ‘AND’, ‘OR’ and ‘XOR’ only. Consider the following scenario describing the AND of five error signals.

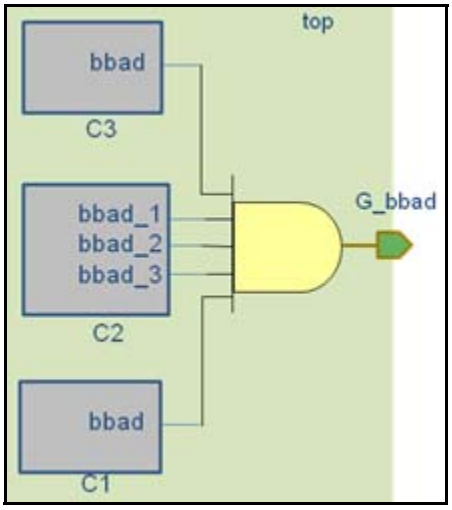


FIGURE 31. Combining signals through gates

This diagram shows three different classes of controllers (C1, C2, and C3) that each produce a ‘bbad’ output status signal. An AND of these signals and brought to the top-level port ‘G_bbad’ with the following command:

```
# Vendor:
mb_define_class –class C1
mb_define_class –class C2
mb_define_class –class C3
mb_define_class –class top –children {C1 C2 C3}
#
# Connect the 'bbad*' ports of the controllers
# through AND gate and bring the status signal to
# the top level port 'G_bbad'
mb_configure_connect_gate -type AND \
```

```
   -from_class { C1 C2 C2 C2 C3} \
   -from_pin   { bbad bbad_1 bbad_2 bbad_3 bbad } \
   -to_class top_class -to_pin G_bbad
```

Note, the above vendor configuration information simply dictates that the permissible connections to one such gate are class objects of C1, C2, and C3 only. In the design, however, the user will instantiate one or many instances of such class objects.

User may prefer to use a specific library cell to combine signals. In that case use the'-cell <cell type>' option to specify the technology cell. However, note that the description of the cell has to be supplied to SpyGlass DFT MBIST product as part of design read and such a cell can only be of type 'AND', 'OR' and 'XOR' functionality (the gate should have 2 inputs only). The above configuration indicating the connection to an AND cell 'AND02_4X' from library can be specified through the modified command:

```
mb_configure_connect_gate -type AND –cell AND02_4X \
   -from_class { C1 C2 C2 C2 C3} \
   -from_pin   { bbad bbad_1 bbad_2 bbad_3 bbad } \
   -to_class top_class -to_pin G_bbad
```

The above will result in insertion of a tree of the AND02_4X cells.

NOTE: *Connections from 'children' classes feeding 'parent' class is the only direction allowed. In case of an attempt to combine in an opposite direction will result in an error message:*

```
Error: Class <to class> is not a parent or grandparent of <from class>
```

The reason for this restriction is that such specifications causes ambiguity in selecting one of many target instances of a child class.

Collecting Design Data and Preparing for MBIST Run

Run SpyGlass to collect information about the memory instances in a design.

mb_report_instances

Run this Tcl command to create a report:

Consider a hierarchical design with instances of specific cells instantiated at various levels:

```
//File: test.v
module test(a1, a2, a3, b1, b2, b3);
input a1, a2, a3;
output b1, b2, b3;
  A inst_A(.a(a1), .b(b1));
  B inst_B(.a(a2), .b(b2));
  C inst_C(.a(a3), .b(b3));
endmodule

module A(a, b);
input a;
output b;
  B inst_B(.a(a), .b(b));
endmodule

module B(a, b);
input a;
output b;
  C inst_C(.a(a), .b(b));
endmodule

module C(a, b);
input a;
output b;
  assign b = ~a;
endmodule
```

Prepare the following Tcl file and run in ‘sg_shell’ environment:

```
# File: test.tcl
# Create a new project
new_project prj –force
# Design read
read_file -type hdl test.v
set_option top test
# Find out all the instances of the cells whose names (cell
names)
```

```
# match ‘C’, ‘A’ and ‘B’

current_methodology $::SPYGLASS_HOME/Methodology/MBIST_DFT
current_goal mbist_dft

mb_report_instances "*C*" "*A*" "*B*"
save_project
close_project
exit
```

It generates the following output:

```
Module - C (under design: test)
    "test.inst_C"
    "test.inst_B.inst_C"
    "test.inst_A.inst_B.inst_C"
Module - A (under design: test)
    "test.inst_A"
Module - B (under design: test)
    "test.inst_B"
    "test.inst_A.inst_B"
```

Designers can write scripts using this Tcl command and prepare the user data to run SpyGlass DFT MBIST product.

A special case arises when the memory instances are instantiated through ‘generate’ statements, as often the case in a Verilog 2001 or SystemVerilog design. Care should be taken to understand how the names of such memory instances are created, and how such names should be used in the user data file. Refer to Appendix – C, at a later part in the document, to get a comprehensive idea of this case.

A variation of this command may be used to print the instances in a specific order, i.e. reporting by hierarchy, as shown below.

mb_report_instances –group_by_hierarchy

This scheme adopts a ‘depth_first_search’ strategy and reports as follows:

1. Search starts at the top level of hierarchy
2. Reports all the ‘matching’ instances at this level
3. Dives down a lower level of hierarchy

4. Repeats the same reporting process (in 2)

An example of a run is as follows:

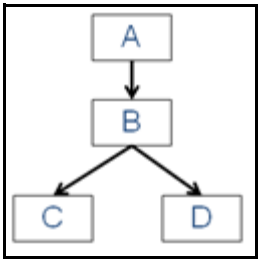
```
mb_report_instances "*wrap2*" "*wrap1*" "*wrap3*" "*wrap4*"
-group_by_hierarchy  -report_file myreport.rpt
```

It is also possible to logically group a number of instances such that they are reported together. This often makes sense if the instances (specified to be in the same logical group) are intended to share the same parent (for example BIST controller, etc.).

An example is as follows:

```
mb_report_instances {*SRAM1D*, *SRAM1P*} *SRAM2T* *CAM*
*DRAM*   -group_by_hierarchy
```

In this example, all the instances of the modules represented by the regexp “*SRAM1D*” and “*SRAM1P*” are meant to share the same parent. In addition, consider the hierarchy in the design as shown:



In this case, the instances are to be reported as:

```
Section for SRAM1D, SRAM1P
     group of instances in the hierarchy A
     group of instances in the hierarchy A/B
     group of instances in the hierarchy A/B/C
     group of instances in the hierarchy A/B/D


Section for SRAM2T
     group of instances in the hierarchy A
     group of instances in the hierarchy A/B
     group of instances in the hierarchy A/B/C
     group of instances in the hierarchy A/B/D
```

```
Section for CAM
    group of instances in the hierarchy A
    group of instances in the hierarchy A/B
    group of instances in the hierarchy A/B/C
    group of instances in the hierarchy A/B/D
Section for DRAM
    group of instances in the hierarchy A
    group of instances in the hierarchy A/B
    group of instances in the hierarchy A/B/C
    group of instances in the hierarchy A/B/D
```

mb_report_instances –dive_in

Normally, mb_report_instances stops at the boundary of a ‘matched’ module and does not dive inside it to find instances of any other matching modules. The ‘-dive_in’ switch allows the search to happen inside an already ‘matched’ module instance.

The following shows a hierarchical design where the module ‘mem’ is instantiated at various levels of hierarchies shown as highlighted.

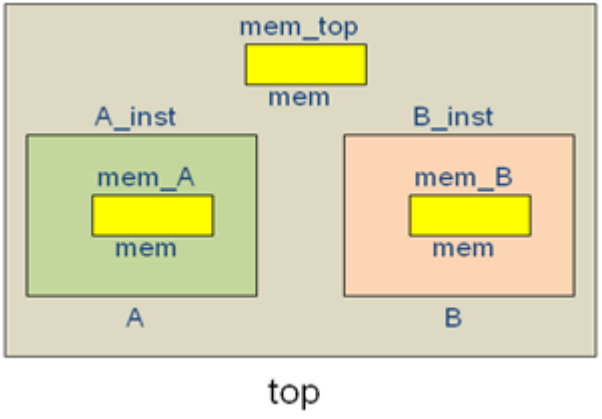


FIGURE 32. A hierarchical design

The following command will produce the report shown below. Note that “mem” inside module A is not reported.

```
mb_report_instances {{*mem*}} "A" \
         -group_by_hierarchy  -report_file myreport.rpt
```

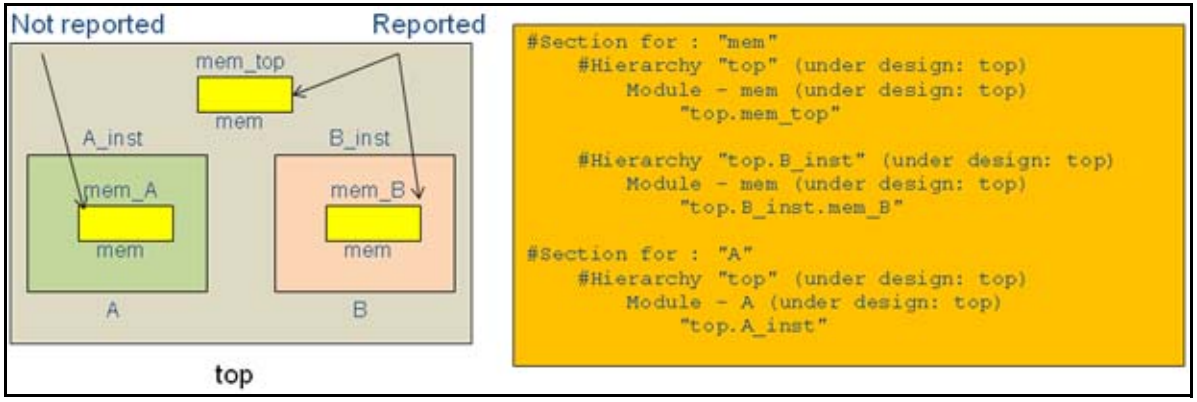


FIGURE 33. The report generated for the above hierarchical design

If the '-dive_in' switch is added to mb_report_instances then the search is continued even inside already 'matched' module instances.

```
mb_report_instances {{*mem*}} "A" \
         -group_by_hierarchy -report_file myreport.rpt \
          -dive_in
```

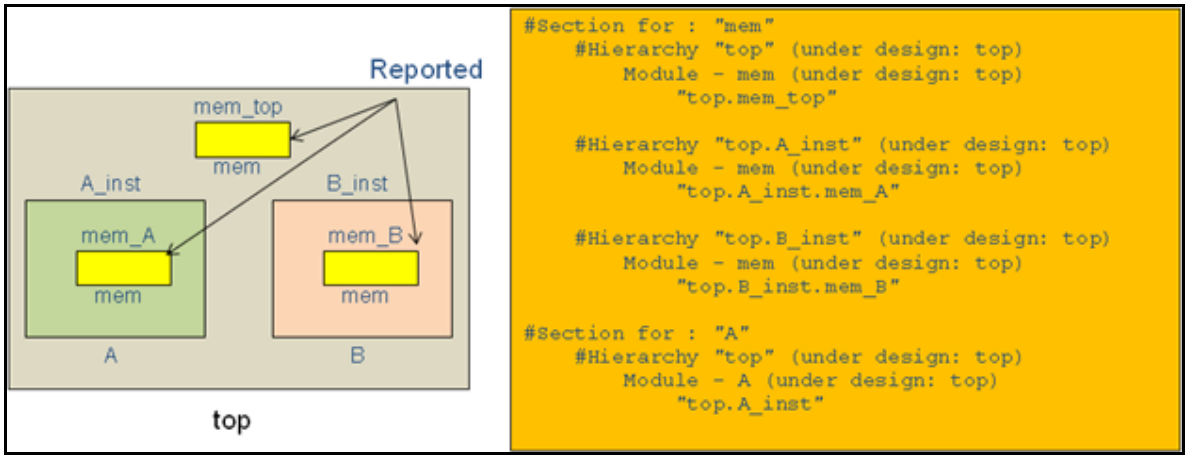


FIGURE 34. The report generated for the above hierarchical design when –dive_in option is used

Creating a User Data File

This section describes the steps that will be followed by the user. We will describe and illustrate the user-driven Tcl commands in detail. However, for the exact syntax and more information please refer to the *SpyGlass MBIST Tcl Flow Reference Guide*.

Preparing the Setup

This is the first step of the setup. This is a common setup step required for any SpyGlass analysis. The goal here is to supply the design information in Verilog/VHDL/mixed language format. Refer to the SpyGlass Explorer Reference Guide for more detailed information. Here we are going to describe a common way of reading a typical Verilog design.

Consider reading a Verilog design consisting of multiple files, header files residing under the ‘include’ directories, libraries spanning across multiple directories specified with ‘-y’. As a SpyGlass design read requirement it is mandatory to create a ‘project’ at the beginning. The Tcl commands for reading the design is illustrated as follows:

```
# Create a new ‘project’ with a name
new_project prj_DESIGN -force
# Command equivalent to ‘-v’
set_option v ./MEMORY_LIB/MEMORY.v
# Command equivalent to ‘+libext+.v’
set_option libext { .v }
# Command equivalent to ‘+incdir+’
set_option incdir { lib/memory_support/verilog/rtl }
# Command equivalent to ‘-y’
set_option y { lib/sram1d/verilog lib/memory_support/
verilog/rtl lib/memory_support/verilog/net rtl }
# Now read the individual files
read_file -type verilog lib/sram1d/verilog/wrappers/
WRAPT2A.v
read_file -type verilog testcase/atrenta/rtl/verilog/
DESIGN.v
```

- **Design read through a consolidated command file**

  Sometimes the designers have already prepared a command file for analyzing their designs through SpyGlass. The same file can be conveniently read for MBIST insertion purpose.

  The following command file represents how a Verilog design can be read into SpyGlass:

  ```
  // File: cmdfile.f
  -incdir 'incdir_abc'
  +libext+.v
  -verilog
  -y 'ydir'
  -v 'lib.v'
  header_b.vh
  test.v
  ```

  The same file can be read as part of the design read step, as:

  ```
  read_file -type sourcelist cmdfile.f
  ```

- **Working with compressed RTL files**

  ‘Compressed’ (gzipped) source files are read as follows:

  ```
  read_file –type hdl <file ame>.gz
  ```

The files ‘included’ in Verilog source code and files supplied through the `v` command can be ‘compressed’ files. The files to be picked up from the directories specified through the y command can also be compressed. In that case, use the following option:

```
set_option libext {.v.gz}
```

If any of the design descriptions supplied through such compressed files need to be modified by SpyGlass then the output of the modified file is also generated as a compressed file.

Example:

```
read_file –type hdl block.v.gz
mb_set_prefix_n_suffix  -file_suffix sg
```

After MBIST insertion in this file, the ‘modified’ file is generated as a compressed file in the name:

```
block.v_sg.gz
```

- **Blackboxing**

  As mentioned earlier, some modules may be ‘unintentionally’ blackboxed. Review the following file and look for the violation messages for the following rules:

  ```
  <project_name>/mbist_dft/spyglass_reports/moresimple.rpt
  ```

You may like to understand the cause of these messages and like to fix them to ensure a clean design read. However, if you are sure that there is no memory inside these modules (and hence BIST insertion is not needed) then you may ignore these messages.

- **ErrorAnalyzeBBox**: This message indicates that the description of the module has not been supplied. The remedy is to supply the HDL for the module.
- **WarnAnalyzeBBox**: This message indicates that either the module description is ‘empty’ or the module is not synthesizable (while SpyGlass can understand the port interface of such module, it cannot navigate inside it).

Sometimes designers may be interested to blackbox parts of the design descriptions, for example a few behavior descriptions of the memory hierarchy. This may be accomplished by explicitly supplying the design name:

```
set_option stop {A_block B_block C_block }
```

or using a wildcard:

```
set_option stop {"A*block" "B*block*"}
```

or using a regular expression:

```
set_option stop { m/^abc[A-Z]+$/ \
                  m/^DLYCLK[0-9]+_[0-9]+_A[TUV][HUX]0[0-9]/ }
```

It is recommended that you confirm that only the desired modules, and nothing else, have been blackboxed. SpyGlass generates the following report:

```
<project name>/mbist_dft/spyglass_reports/SpyGlass/
stop_summary.rpt
```

Review this file to find out the list of modules that have indeed been blackboxed due to the user's directive. The user-specified blackboxes are also indicated by the InfoAnalyzeBBox messages in the file:

```
<project name>/mbist_dft/spyglass_reports/moresimple.rpt
```

Using Waivers

Sometimes one may like to 'waive' certain messages (generated by SpyGlass DFT MBIST product) depending on nature of the design read. Refer to the SpyGlass Explorer User Guide for more detailed information about the 'waiver' mechanism.

An example of usage of waiver is provided in *Appendix B: An Example of Using Waivers*.

Working with Encrypted Designs

It may be a requirement to work with an IP provided by a third-party vendor. For confidentiality, the vendor may wish to encrypt the source code of the IP. If there are memory instances in an encrypted IP then these memories will not be BISTed.

The following is the recommended methodology for working with 'encrypted' designs.

1. Use an `include statement to supply the definition of the encrypted IP. Embrace this `include statement within pragmas, as shown below:

   ```
   // File: top.v
   module top (…..);
   ```

```
:
// This is the instantiation of the encrypted IP.
// The instance will be blackboxed since no definition will
// be visible to SpyGlass
IP ip_inst(.A(w1), .B(w2), …);
:
endmodule
// vendor_string translate_off
// This encrypted file contains the description of IP
`include “core.vp”
// vendor_string translate_on
```

2. Use the following option during design read to SpyGlass:

```
set_option pragma vendor_string
```

Here *vendor_string* is any arbitrary string suitably chosen by the RTL developer. Because of the ‘set_option’ directive, SpyGlass will not read the content enclosed in between the pragmas. The advantage to this approach is that the reference to the definition (of the encrypted IP) will be available in the modified RTL (post-MBIST insertion) for other tools to use.

Specifying the Description of the BIST components

The description of the components to be inserted (for example the BIST components that are not part of the original design) by any of the following methods:

- Explicitly supply the description of the file:

```
read_file -type hdl lib/sram1d/verilog/bist/BIST9A.v
```

- Specify the description of the components through the y command:

```
set_option libext { .v }
set_option y { lib/bist_lib/verilog lib/memory_support/
verilog/rtl }
```

- Specify the description of the components through the v command:

```
set_option v ./BIST_LIB/BIST_COMPONENTS.v
```

The description of such components, supplied through one or more of the above methods, will be picked up by SpyGlass DFT MBIST product during insertion.

Specify the Methodology and Goal

To run SpyGlass DFT MBIST product, choose the Methodology as 'MBIST_DFT' and the goal as 'mbist_dft'

```
# $SPYGLASS_HOME may be conveniently set environment variable
# to point to the SPYGLASS_HOME directory in the software
# release tree
current_methodology $SPYGLASS_HOME/Methodology/MBIST_DFT/
current_goal mbist_dft -alltop
```

Once the 'current goal' is specified, it is mandatory to set the 'top level' of the design for the purpose of MBIST insertion. This is especially important in the case of SpyGlass DFT MBIST product because all the BIST components that are not yet part of the design will create a multiple top situation. This is prevented by the following command:

```
set_parameter mbist_top_level_name DESIGN
```

Read the Vendor Data

The user is now required to read the vendor specific data (class definition, connection specification etc.) in the sg_shell environment. It is often convenient to encapsulate this vendor-specific information in a file and read it.

```
# As a convenience the vendor specific commands are present
# in the file 'vendor.data'
source vendor.data
```

Controlling the Output Format

Various aspects of the output format are controlled by the following command:

```
mb_set_prefix_n_suffix
    [-module_prefix <string>]
    [-file_prefix <string>]
    [-file_suffix <string>]
    [-wire_prefix <string>]
    [-help]
```

NOTE: *The following two conditions must be satisfied for changing the file name:*

- *The file name and module names must match in the pre-BIST RTL*

- *The file must not contain more than one module*

If either or both of the above conditions are not met, the file name is not changed.

Also note that it is the user's responsibility to supply appropriate module and file prefixes so that the module and file names match in the post-BIST RTL.

Prefixes/suffixes are applied only to the modified/added files/modules.

The default values of the user-configurable options are as follows:

```
FILE PREFIX
      default value = "atrenta_modified_"

FILE SUFFIX
      default_value = "" (null)

MODULE PREFIX
      default value = "" (null)

WIRE PREFIX
       default value = "Atrenta_wire_"
```

Defining the Existing Instances

The memory instances that already exist in the design must be mapped to the defined 'classes'.

```
# Note: the classes should have already been defined through
# 'mb_define_class' in the vendor file
mb_define_instance -instance i_distributed.i_big_mem.i_mem1\
                -class MEM_2048x32m8_LTB1
mb_define_instance -instance i_distributed.i_big_mem.i_mem2\
                -class MEM_2048x32m8_LTB1
mb_define_instance \
          -instance i_distributed.i_small_mem.i_mem0.i_mem2\
                -class MEM_256x8m8_LTB1
mb_define_instance -instance i_red1.i_mem1\
                -class MEM_4096x32m8_LRTB2
```

Inserting/Replacing and Defining New Instances

Inserting New Instances

The user now inserts new instances into the design (typically, BIST related components, for example, BIST engines, controllers, etc.). This requires inserting an instance of a ‘cell’ or module definition. This is followed by mapping this instance to a predefined class and indicating which instances are its children. A sanity check is in place to ensure that the inserted instance has only ‘legal’ children (as indicated in the class relationship in the vendor data).

```
# Vendor data: Defines class structure and thereby recommends
# a legal use model
#
# Define two basic memory classes
#
mb_define_class -class MEM0
mb_define_class -class MEM1
#
# Now define a bist class that can control only memory
# class objects.
# According to this rule the bist class can control none or
# any number of instances belonging to these two memory
# classes
#
mb_define_class -class BIST1 -children {MEM0 MEM1}
#
# ------ End vendor file

# User data: User is now ready to work with the design as per
# the guidelines set in the vendor file
# The following are the two instances of memory classes
#
mb_define_instance  -instance I7 -class MEM0
mb_define_instance  -instance I6 -class MEM1
#
# Now insert an instance of bist class (CONTROLLER_1) to
# control I7 and I6.  This is permissible in accordance with
```

```
# the guidelines # set in the vendor file.
#
mb_insert_instance -instance CONTROLLER_1 -cell BIST1
mb_define_instance -instance CONTROLLER_1 -class BIST1
-children {I7 I6}
```

Inserting New Instances of Parameterized Modules

During the insertion step, use the ‘mb_insert_instance’ command with specific parameter values through the ‘-parameter_map’ switch. For example

```
mb_insert_instance -instance I2 -cell WRAP_FUSECNTL_T09 \
                  -parameter_map {NUM_BAYS 23 MEM_ADDR 10}
mb_define_instance -instance I2 -class fuse_class
```

Replacing Existing Instances

The designer can ‘replace’ an existing instance in the design with a block (maybe a ‘wrapper’). As a result of the replacement, every functional connection will be established to at least one pin of the new replacing block. It is expected that every pin of the ‘replaced’ block be mapped to a pin of the ‘replacing’ block. However, in case the designer decides to ‘ignore’ mapping a list of pins of the replaced block, that can be done as we will see later in the illustration. This is accomplished through the basic command:

```
mb_replace_instance -instance
    -cell <replacing block name>
    [-pin_map {<new_pin old_pin>*}]
    [-ignore_pin <old_pin>]
```

Keep in mind that the designer can omit the explicit pin mapping. In that case, the same pin names will be assumed. The following diagram illustrates a replacement and the associated commands.

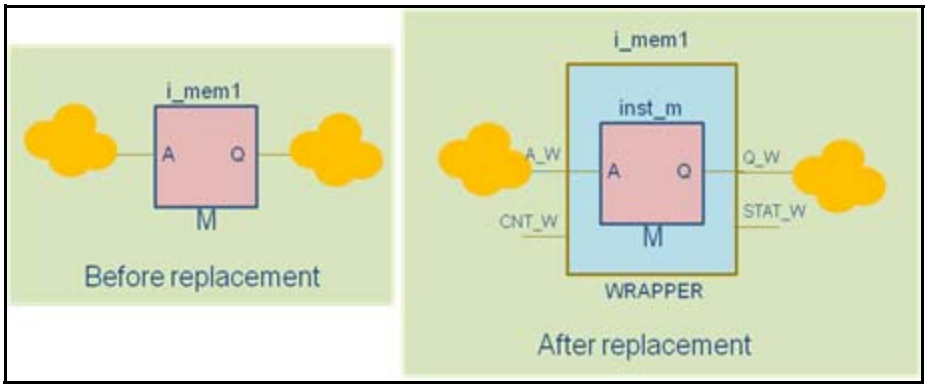


FIGURE 35. Existing instances replaced during MBIST insertion

```
# Vendor data:
# The replacing module should belong to a ‘pre-defined’
# class. # So, define a suitable class
#
mb_define_class –class wrapper
#
# -- End vendor data
#
# User data:
# Replace i_mem1 with an instance of WRAPPER
mb_replace_instance -instance i_mem1 -cell WRAPPER \
                 -pin_map {A_W  A  Q_W  Q}
mb_define_instance  -instance i_mem1 -class wrapper
```

Note, there is no need to define a class for the ‘replaced’ object. Also, note that after the replacement of the instance, the extra pins on the new block are now available for connection to other (BIST) components.

In case there are extra pins on the memory that do not have a corresponding pin map on the wrapper, then such pins can be ‘ignored’ during replacement with the following command:

```
mb_replace_instance -instance i_mem1 -cell WRAPPER \
          pin_map {A_W  A  Q_W  Q}    -ignore_pin { Y }
```

The above may be the case where the memory internally drives a scan flop

and the scan flop connections are brought out through one or more pins on the wrapper.

Removing the Existing Instances

Use the ‘mb_remove_instance’ Tcl command for this purpose.

This command can be either part of the vendor or the user data file. The use model is illustrated as follows:

```
# Remove all the instances of the cell where the cell name
# contains the substring “TW”
mb_remove_instance -cell {{*TW*}}
# Remove all the instances of the cell where the cell name
# contains the substring “ABC”
mb_remove_instance -cell {{*ABC*}}
# Remove all the instances of the cell where the cell name
# contains the substring “TERM” and the instance name
# contains the substring “term_return”
mb_remove_instance -cell {{*TERM*}}
        -instance {{*term_return*}}
```

Note, this command can be part of the vendor file. However, the design may not instantiate any cell that satisfies string-matching criteria. In this case, a warning message is generated and the user is allowed to move on.

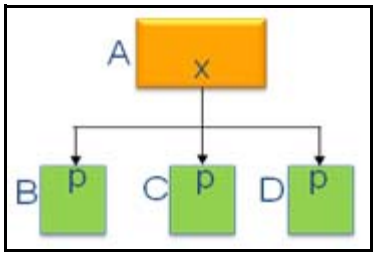
Establishing Custom Connections

Most of the connections will be established by the connection rules specified in the vendor file. If there is a need for custom connections (for example from a pin of an instance to a pin of another instance) then they can be accomplished through the ‘mb_connect_net’ command:

```
mb_connect_net
    [-from <pin>*]
    [-to <pin>]*
    [-tie_extra_receivers <0|1>]
    [-through_in_pin <pin> -through_out_pin <pin>]*
    [-broadcast | -parallel | -bitwise_broadcast]
    [-multiplex  -select_enable <string>  -active <high|low>]
```

```
[-id <string>]
[-help]
```

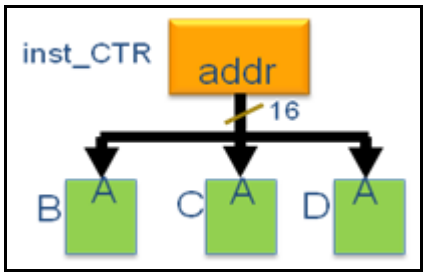
For example, the following diagram represents a broadcast topology where a scalar pin ‘x’ on the instance A must be connected to each of the pin ‘p’ of the instances B, C, and D.



This custom connection is accomplished by:

```
mb_connect_net –from A.x –to {B.p C.p D.p} –broadcast
```

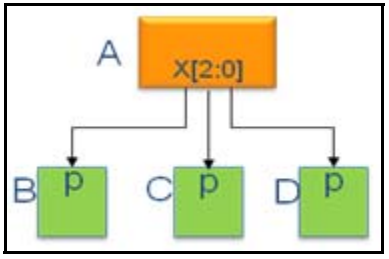
Note, the above connection could also be specified as a ‘bitwise_broadcast’ connection. However, it is mandatory to use the ‘-bitwise_broadcast’ switch if the pins are a vector, as follows:



The connection being a bus, one needs to use the command:

```
mb_connect_net –from inst_CTR.addr –to {B.A C.A D.A} –
bitwise_broadcast
```

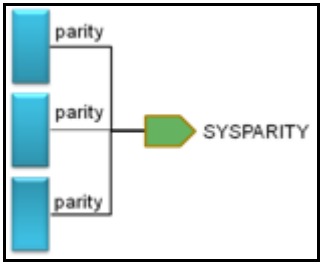
The diagram below, shows a scalar pin’p’ on the instances B, C, and D receiving data from a three bit bus. Such connections are established through the following command:




```
mb_connect_net –from A.X –to {B.p C.p D.p} –parallel
```

Examples of some custom connections are shown below:

- Connecting a pin of an instance (newly inserted or existing) to a primary I/O port




```
mb_connect_net -from { inst1.parity inst2.parity
inst3.parity } \
            -to SYSPARITY \
            -add_port \
             -parallel
```

- Connecting a pin of an instance to GND or VCC

  Use a command such as:

```
mb_connect_net -from 0 -to FUSE_inst1/FUSE_MODE -parallel
```

- Overriding a vendor-directed connection with a custom connection

  Any connection specified using the *mb_connect_net* command will automatically override the default connection configuration specified through *mb_configure_command_net* command.

Consider the vendor rule for connection setup:

```
mb_configure_connect_net -from_class abs_TOP -from_pin 1 \
    -to_class BST -to_pin BIST_MODE -broadcast \
    -id CCN_BIST_MODE_TO_1
```

By default, all the class objects matching these classes are connected according to this rule. However, you can define the following custom connection in user data file to override the above default vendor connection:

```
mb_connect_net -from sys_logic_inst/TM_OUT \
    -to BIST_inst1/BIST_MODE -parallel
```

Another mechanism to override vendor connection is based on skipping connection configuration using connection IDs as defined below:

```
mb_define_instance -instance BIST_inst1 -class BST \
   -children mem_inst1 -skip_connection CCN_BIST_MODE_TO_1
```

Inserting MBIST: The mb_insert Tcl Command

This tcl command initiates application of vendor commands to user-identified instances and generates the “expanded” commands that will be applied to a design, and will invoke SpyGlass DFT MBIST product for design change.

```
mb_insert
    [-preview]
    [-output_control_file <output control data file>]
    [-input_control_file <input control data file>]
    [-force]
    [-report <report file name>]
    [-help]
```

It is recommended that the tool is run in ‘preview’ mode to ensure that the ‘expanded’ commands are properly generated. This step is analogous to the ‘elaboration’ step for a compilation activity. This enables the user to review the errors and fix them before undergoing the final step of inserting the BIST logic.

The following command will activate the ‘-preview’ switch:

```
mb_insert –preview
```

The following in the stdout indicates that the expanded commands have been generated successfully without encountering any error from both the vendor file and user file.

```
#Found 0 error(s) while processing Vendor MBIST configuration
data
#Found 0 error(s) while processing User MBIST configuration
data

:

Generated MBIST control file 'top.mbist.sgdc'
0
sg_shell>
```

It also produces useful information about the class hierarchy as depicted in the design, for example

```
#Info: No parent found for instance '<CURRENT_MBIST_TOP>'
# <CURRENT_MBIST_TOP> (class: top_class)
#     |--- TOP_CONTROLLER (class: BIST_CON_FBOX)
#     |    |--- CONTROLLER_1 (class: BIST1)
#     |    |    |--- I7 (class: RAM_BLOCK_BIST1_u_sram0)
#     |    |    |--- I6 (class: RAM_BLOCK_BIST1_u_sram1)
#     |    |
#     |    |--- CONTROLLER_2 (class: BIST2)
#     |    |    |--- I3 (class: RAM_BLOCK_BIST2_u_sram2)
#     |    |
#     |    |--- CONTROLLER_3 (class: BIST3)
#     |    |    |--- I1 (class: RAM_BLOCK_BIST3_u_sram4)
#     |    |    |--- I2 (class: RAM_BLOCK_BIST3_u_sram3)
#     |    |
#     |    |--- CONTROLLER_4 (class: BIST4)
#     |    |    |--- I4 (class: RAM_BLOCK_BIST4_u_sram6)
#     |    |    |--- I5 (class: RAM_BLOCK_BIST4_u_sram5)
#     |    |
#     |
#
```

As part of this ‘expansion or elaboration’ process, a SGDC file is created

(by default it is <top module name>.sgdc or it can be specified through the '-output_control_file' switch of this command. This sgdc file contains the expanded version of all the directives that SpyGlass DFT MBIST product will act on.

Run the following command to invoke the design modification process:

```
mb_insert
```

Specifying the Location of Modified Design Files

The modified design files are deposited under a specified directory and can be accessed through the following file:

```
<project name>/mbist_dft/spyglass_reports/rme/mbist-dft/
atrenta_modified_rtl_files/atrenta_generated_design.f
```

The designer has the option to collect all the modified files in a user-specified location with the command 'mb_collect_modified_files':

```
mb_insert
mb_collect_modified_files <user specified directory>
```

NOTE: *This is a post-insertion step.*

Verifying the Post-BIST Design

In this section, we discuss some steps for working with the post-BIST design.

Viewing the Modified Design

It is useful to view the modified design in the schematic. A convenient way to accomplish this task is through the following steps:

1. Read the modified design as part of the standard design read process.
2. Synthesize the design by running a pre-defined 'goal', such as GuideWare/New_RTL/initial_rtl/lint/synthesis.
3. View the synthesized design in SpyGlass Console.

The following two approaches are used to accomplish this:

- **Project file based**

- Create a project file as follows:

```
# file: test.prj
##Data Import Section
read_file -type sourcelist <project name>/mbist_dft/
spyglass_reports/rme/mbist-dft/
atrenta_modified_rtl_files/atrenta_generated_design.f

##Common Options Section

set_option language_mode mixed
set_option projectwdir .
set_option projectcwd .
set_option active_methodology $::SPYGLASS_HOME/
GuideWare/New_RTL

##Goal Setup Section
current_goal initial_rtl/lint/synthesis –top top
```

- Run spyglass to synthesize the design

```
spyglass –project test.prj –goal initial_rtl/lint/
synthesis -batch
```

- Invoke SpyGlass Console to view the design in the schematic

```
spyglass –project test.prj &
```

- **Tcl script based**

  - Create a Tcl script as follows:

```
# File: test.tcl
set design top
new_project prj_view_modified_$design -force

# Read the post-BIST design
read_file -type sourcelist prj_top/mbist_dft/
spyglass_reports/rme/mbist-dft/
atrenta_modified_rtl_files/atrenta_generated_design.f

current_methodology $::SPYGLASS_HOME/GuideWare/New_RTL
```

```
# Goal setup setup section
current_goal initial_rtl/lint/synthesis –top $design

run_goal
save_project

# Invoke the GUI to view the schematic and results
gui_start
```

- Invoke this script as follows:

```
sg_shell
   source test.tcl
```

Verifying the AND/OR/EXOR Logic Tree

As discussed during the ‘Approach’ section, the motivation is to check the sanity of created logic tree generated as a result of ‘mb_configure_connect_gate’ command or ‘rule’ described in the vendor data.

The designer reads the post-BIST design and runs the ‘mb_assert_function’ command. Refer to the ‘SpyGlass DFT MBIST Tcl Flow User Manual’ for the detailed syntax of this command. In a very basic form, it can be specified as follows:

```
mb_assert_function
     -from <list of start_nodes>
     -to <end_node>
     -function_type <func_name>
```

The type of functions that can be checked (specified through the ‘-function_type’ switch are ‘or’, ‘and’, and ‘exor’.

An example of this command is illustrated as follows.

```
set design test
new_project prj_assert_fn_$design -force

# Read the modified RTL for the design (test)
# Note the design has been modified in an earlier step and
# the the database is now residing under the ‘prj_$design’
```

```
# directory.
# We are going to verify a desired structure in that design.
read_file -type sourcelist prj_$design/mbist_dft/
spyglass_reports/rme/mbist-dft/atrenta_modified_rtl_files/
atrenta_generated_design.f \
set_option top $design

current_methodology $::SPYGLASS_HOME/Methodology/MBIST_DFT
current_goal mbist_dft

mb_reset_assertions

# ------------------------------------------
# Verify that the two source nodes are indeed connected to
# the destination node through a XOR tree
# ------------------------------------------
mb_assert_function -from {W31_1/INTWRAP/PARITY W31_2/
INTWRAP/PARITY} \
          -to PARITYTREE_WRAP_FUSECNTL_BISTCNTL_ref1       \
          -function_type XOR

mb_check_assertions

save_project
close_project
```

The result appears on the stdout:

```
RESULT: mb_assert checks PASSED
        Look for Info messages from following rule(s) in
'prj_assert_fn_test/test/custom_dft_block_check/
spyglass_reports/moresimple.rpt':
                'Soc_07_Info'
sg_shell>
```

Verifying the Connectivity of Critical Paths

The connectivity of critical paths of interest can be validated through the command ‘mb_assert_path’, as illustrated below:

```
mb_reset_assertions

mb_assert_path \
   -from a\[2:0\] -to b\[2:0\] \
   -path_type sensitizable -parallel -no_invert

mb_check_assertions
```

The confirmation appears on the stdout:

```
RESULT: mb_assert checks PASSED
        Look for Info messages from following rule(s) in
'prj_test/test/custom_dft_block_check/spyglass_reports/
moresimple.rpt':
                'Soc_02_Info'
sg_shell>
```

Ignore non-existent nodes during connectivity check (mb_check-node_existence)

When a design is intended for reuse, it may be useful to ‘ignore’ user-specified nodes that do not exist in the design (by default a FATAL error message is reported if the node does not exist). This can be accomplished with the user-specified option ‘mb_check-node_existence’.

Example:

```
mb_assert_path \
    -from inst_A/inst_AA/O5 -to inst_B/inst_BB/I6 -parallel \
    -path_type buffered

# The following prevents FATAL to produce due to non-existent
# nodes

mb_check_node_existence off

# The following node 'O7' does not exist
mb_assert_path \
    -from inst_A/inst_AA/O7 -to inst_B/inst_BB/I6 -parallel \
    -path_type buffered

# The following node 'I8' does not exist
```

```
mb_assert_path \
    -from inst_A/inst_AA/O5 -to inst_B/inst_BB/I8 -parallel \
    -path_type buffered

mb_check_node_existence on
```

Verifying the Connectivity of Critical Nodes to Ground or Power

The connections to ground or power for specific nodes in the post-BIST design can be validated through the command ‘mb_assert_value’, as illustrated below:

```
mb_reset_assertions

mb_assert_value -name inst_B/a -value 0 -use_shift

mb_check_assertions
```

The validation status appears on the stdout as follows:

```
RESULT: mb_assert checks PASSED
        Look for Info messages from following rule(s) in
'prj_check_fn_top/top/custom_dft_block_check/
spyglass_reports/moresimple.rpt':
                'Soc_01_Info'
sg_shell>
```

Summary: Usage of the mb_assert* Commands

Any combination of the ‘mb_assert*’ commands may be used to validate the post-BIST designs illustrated below:

```
#Reset previously defined checks
  mb_reset_assertions

# Specify what you want to check
  mb_assert_path ....
  mb_assert_path ....
# Optionally ‘ignore’ some non-exitent nodes if you want
  mb_check_node_existence off
```

```
  mb_assert_value ....
  mb_assert_path ....
  mb_assert_path ....
  mb_assert_value ....
  mb_assert_function  ...
# Stop ‘ignoring’ non-exitent nodes
  mb_check_node_exitence on
  mb_assert_function  ...

# Finally check all of the above
mb_check_assertions
```

Bottom-up Methodology

So far, we have described the basic concepts and some details of inserting BIST components in a block level design. In this section, we will learn how an already BIST-inserted block can be used as a component of a higher-level block while the rest of the higher-level block is inserted with BIST components.

The Bottom-up Flow

Assume that the design in *Figure 36* represents a SoC consisting of three blocks. The designer performs BIST insertion in block A through a stand-alone SpyGlass DFT MBIST run. The BISTed version of block A can then be inserted into the higher level of assembly. The modified description of the block A (the BISTed version) is read along with the description of the rest of the SoC. It now requires to insert BIST components in other parts of the SoC, leave block A as 'untouched' and hook up the test ports of block A to the rest of the design.

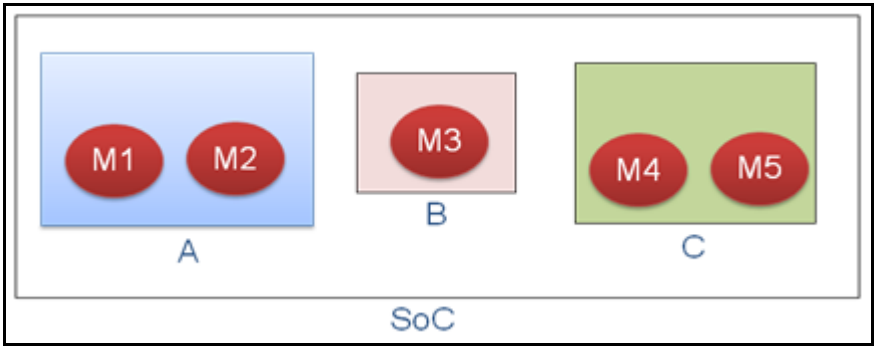
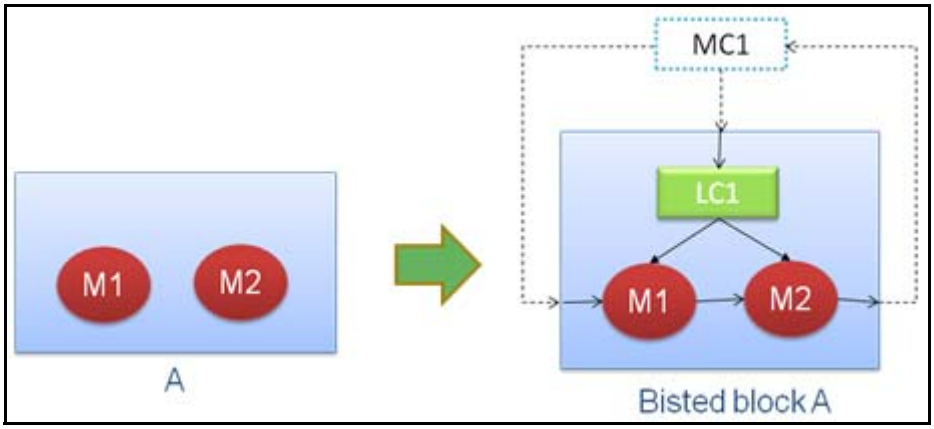


FIGURE 36. A SoC

The following steps should be performed by the designer:

1. Insert BIST components in block A

   This will result in inserting various components inside this block and creating additional test ports for communicating (in future) with controllers that will be placed outside of the block.



Notice that additional test ports are added after the modification of the design. The dotted lines represent future communication paths to controllers at the SoC level ('MC1' with a dotted boundary represents a master controller that will be inserted in the SoC).

2. Insert BIST for the rest of the SoC

   This involves reading the modified design for block A along with the rest of the SoC. Then, insert BIST in the rest of the SoC (for example inserting LC2 and LC3 inside blocks B and C) while keeping the block A (BISTed version) as 'untouched' and finally hook up the test ports of block A to the rest of the design.

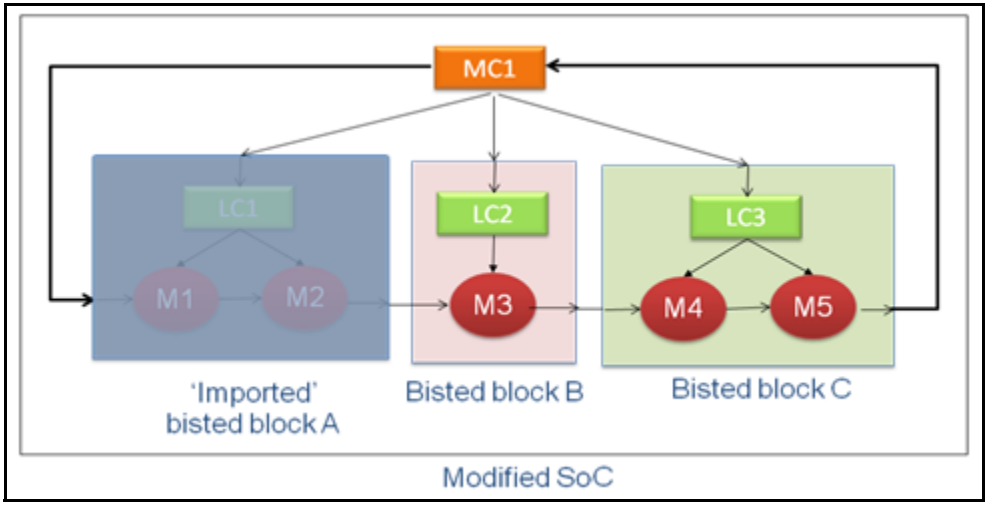


FIGURE 37. Modified SoC

To summarize, the steps of the bottom-up flow are as follows:

1. At the bottom level: Bring connections from lower objects to the block boundary
2. At the next higher level: Connect test ports at boundary of lower block to upper level controllers

The lower level block is represented by an 'abstract class' to automate connection. The concept of the 'abstract class' is discussed in section *Abstract Classes*.

Design Read for Bottom-up Flow

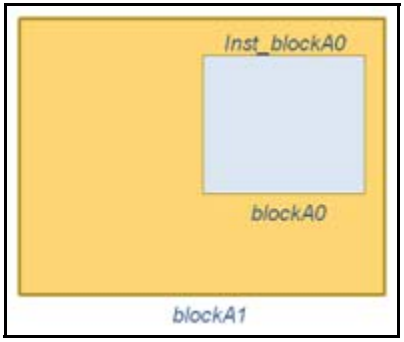
The design read principles we have discussed in a previous section also apply for the bottom-up flow. However, the only exception is that we are now trying to read the modified (BISTed) design for the lower-level blocks.

During any single SpyGlass run, a file named 'atrenta_generated_design.f' is created at the following path:

```
<project name>/mbist_dft/spyglass_reports/rme/mbist-dft/
atrenta_modified_rtl_files
```

This file contains the entire description of the modified design. This file will be inserted at higher levels.

In the following SoC design block ‘blockA0’ is BIST-inserted in a stand-alone MBIST run.



The user data file created for processing this block (blockA0) is as follows:

```
############################
# User data file for blockA0
############################
set design blockA0
new_project prj_$design -force

read_file -type hdl $design.v
<other commands>
mb_insert
save_project
close_project
```

Upon completion of the blockA0 run, the following file is created, which contains the entire description of the modified blockA0:

```
prj_blockA0/mbist_dft/spyglass_reports/rme/mbist-dft/
atrenta_modified_rtl_files/atrenta_generated_design.f
```

This blockA0 file is read in the user data for the processing of the higher-level block (blockA1):

```
########################
# User data file for blockA1
########################
```

```
set design blockA1
new_project prj_$design -force

read_file -type hdl $design.v
read_file -type sourcelist prj_blockA0/mbist_dft/
spyglass_reports/rme/mbist-dft/atrenta_modified_rtl_files/
atrenta_generated_design.f
<other commands>
```

Abstract Classes

An abstract class is defined in the vendor file. This is done to enable the bottom-up flow through a process of ‘abstraction’. Once a block is BISTed then usually this block is defined as an instance of a previously defined abstract class. This will enable us to refer to the BISTed block in SoC in future during a bottom up flow. In our previous example, we will have an abstract class definition (say ‘ABS_A’) in the vendor file, as follows:

- Vendor data:

  ```
  mb_define_class -class ABS_A -children {ABS_A LC1} \
                -abstract -auto_configure
  ```

  Note, the switch ‘-abstract’ indicates that the class is an abstract class. We will defer discussion about its ‘children’ and the other switch ‘-auto_configure’ until later.

  Once the block A is BISTed, then we will need to define the ‘top’ level of the ‘modified block A’ as an instance of the abstract class ‘ABS_A’, as follows:

- User data (while working with block A)

  ```
  mb_define_instance –top –class ABS_A –children <instance
  of LC1>
  ```

  Subsequently, it is also necessary that one defines an instance of abstract class ABS_A to refer to the instance of modified block A at the SoC level. Hence, the user data (while working with SoC) should reflect this.

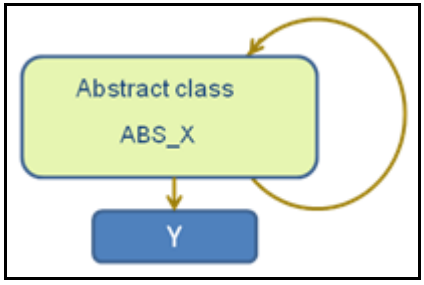
- User data (while working with SoC)

  ```
  mb_define_instance -instance <instance name of A in SoC> -
  ```

```
class ABS_A
```

More on Abstract Classes

There may be a necessity of instantiating an abstract class object by an object of the same abstract class. Hence, it is a special property of an abstract class that it can be its own 'children'. The following diagram shows that 'ABS_X' is an abstract class; Y is a class that is a child of this abstract class. Moreover, ABS_X is its own 'child'.

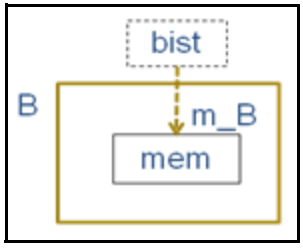



```
mb_define_class -class ABS_X -abstract \
             -children {ABS_X  Y} -auto_configure
```

Example of abstract class

The following example illustrates the concept of the abstract class and its use in a bottom-up flow.

Consider a block (B) that instantiates a memory (mem). The class relationship is defined in the vendor data file. We will eventually create an abstract class out of B for future use at a higher level.




```
# Vendor data file
mb_define_class -class MEMORY
```

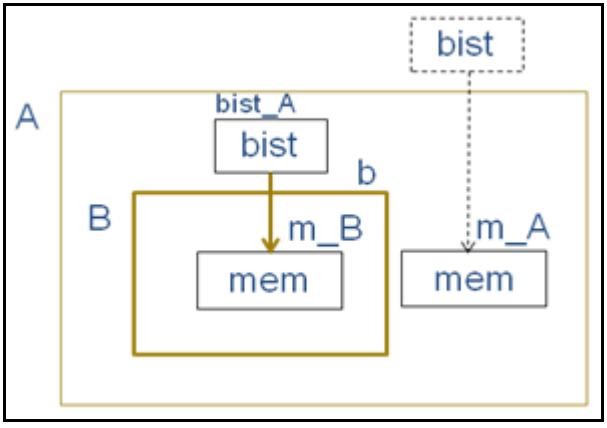
```
mb_define_class -class BIST -children MEMORY
mb_define_class -class ABS_BIST -abstract \
             -children {ABS_BIST MEMORY} -auto_configure
```

- **Step 1**: Create an abstract class (ABS_BIST) out of B. Note, the top level of this design (indicated by '-top' in the command) is created as the abstract class.

  ```
  # User File
  mb_define_instance -instance m_B -class MEMORY
  mb_define_instance -top -class ABS_BIST -view viewB -
  children m_B
  ```

  (The concept of '-view' is described in the next section)

  Step 2: Instantiate an abstract class (ABS_BIST) object (B) inside A and create an abstract class (ABS_BIST) object out of A.




```
# User File
mb_define_instance -instance m_A -class MEMORY
mb_define_instance -instance b -class ABS_BIST -view viewB
mb_insert_instance -instance bist_A  -cell bist
mb_define_instance -instance bist_A  -class BIST -children
{b -view viewB}
// Create an object of class ABS_BIST out of A
mb_define_instance -top -class ABS_BIST -view viewA -
children m_A
```

Introducing Views

'View' (as we have noticed in the previous example with the usage of switch '-view' in the 'define_isntance' command) facilitates associating specific instances inside an abstract class object. This will further facilitate referring to these instances from outside the abstract class object, at a higher level in the SoC.

Let us once again refer to the previous example.

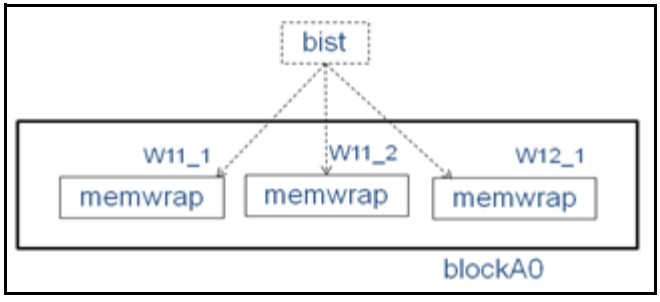
In Step 1, 'viewB' of the ABS_BIST class object refers to the instance of the memory 'm_B'.

In Step 2, this instance is referred to at level A. Notice, the BIST instance 'bist_A' associates with this instance (viewB) as its child as:

```
mb_define_instance -instance bist_A  -class BIST -children {b
-view viewB}
```

Example of usage of 'view'

Consider a block (blockA0) with three different memory wrappers. When instantiated in higher levels, one or more BIST engines can be associated with these wrappers. The 'dotted' lines and the box represent future connections and new components.



The vendor data file would be as follows. Note the class relationship and the definition of the abstract class ABS_BIST.

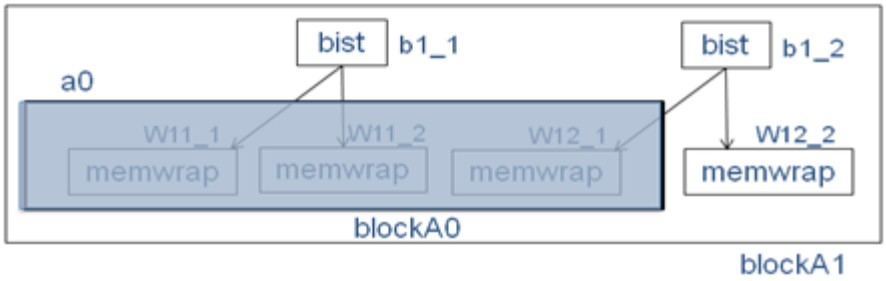
```
# vendor File
mb_define_class -class WRAP
mb_define_class -class BIST -children WRAP
mb_define_class -class ABS_BIST -abstract \
                    -children {ABS_BIST WRAP} -auto_configure
```

The designer creates an abstract class object at the level of blockA0. Note, the three views are created for the three memory wrapper instances inside this abstract class object.

```
# User File
mb_define_instance -instance w11_1 -class WRAP
mb_define_instance -instance w11_2 -class WRAP
mb_define_instance -instance w12_1 -class WRAP
mb_define_instance -top -class ABS_BIST -children w11_1 -view
v1
mb_define_instance -top -class ABS_BIST -children w11_2 -view
v2
mb_define_instance -top -class ABS_BIST -children w12_1 -view
v3
```

Now we are ready to use this modified block at the next level (while inserting BIST for blockA1). At this step, the designer performs the following tasks:

1. Instantiate, at a higher level, an abstract class object with several ‘views’,
2. Insert BIST components at this level
3. Associate instances from within the abstract class object to the newly inserted BIST components.



The user data file looks like the following:

```
# User File
mb_define_instance -instance w12_2 -class WRAP
mb_define_instance -instance a0 -class ABS_BIST  -view v1
mb_define_instance -instance a0 -class ABS_BIST  -view v2
mb_define_instance -instance a0 -class ABS_BIST  -view v3
```

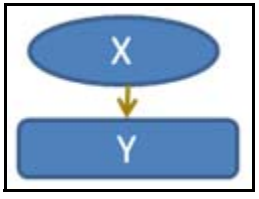
```
mb_insert_instance -instance b1_1 -cell bist
mb_insert_instance -instance b1_2 -cell bist
mb_define_instance -instance b1_1 -class BIST \
              -children {{a0 -view v1} {a0 -view v2}}
mb_define_instance -instance b1_2 -class BIST \
                -children {{a0 -view v3} w12_2}
```

Abstract Classes with auto_configure

'auto_configure' is an option to an abstract class definition and helps in extending the configuration of abstract classes. At the end of the definition of all abstract classes you should use the command 'mb_auto_configure_abstract_classes' to, actually, extend the configuration of the abstract classes and prepare a BIST-inserted block for connection in a higher level block in the future.

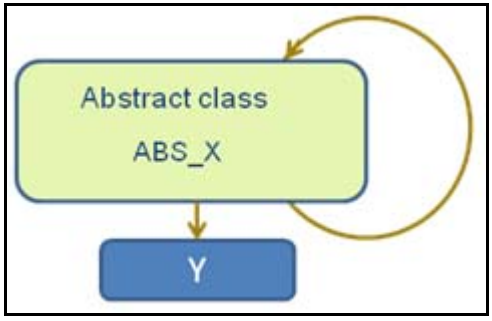
Consider the following class relationship, represented by the vendor file as:

```
# Vendor data
mb_define_class –class Y
mb_define_class –class X –children Y
```



Let us now consider a block containing an object of class Y. Insert BIST into this block and define an abstract class for this modified block:

```
# Vendor data
mb_define_class –class ABS_X –children  {ABS_X Y} \
             –abstract –auto_configure
mb_auto_configure_abstract_classes -verbose
```



The command ‘mb_auto_configure_abstract_classes’ extends the configuration of this class relationship and the resultant architecture is reported as:

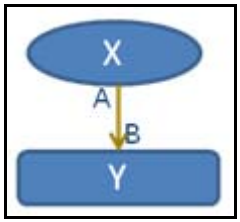
```
#Reporting MBIST Architecture ...
  mb_define_class -class Y -children {}
  mb_define_class -class ABS_X -children {ABS_X Y} -abstract
; # -auto_configure
  mb_define_class -class X -children {Y ABS_X}
```

You will notice that the extended configuration establishes a relationship between class X and ABS_X in addition to the relationship between X and Y.

The rules, defined among various classes, get extended to the abstract classes. To illustrate the concept, consider one connection rule (mb_configure_connect_net) defined among two classes, as shown in the following vendor file:

```
# Vendor data
mb_define_class –class X –children Y
mb_configure_connect_net  –from_class X  –from_pin A
                       -to_class Y    –to_pin B    -broadcast
mb_define_class –class X’ –children  {X’ Y} –abstract –
auto_configure
mb_auto_configure_abstract_classes
```

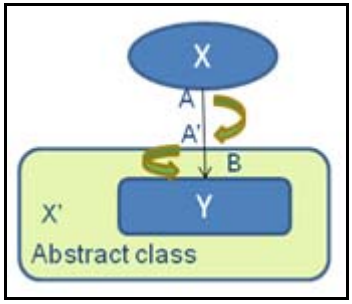
The vendor file establishes a connection rule between the parent class X and child class Y as shown in the diagram below:



Because of the elaboration (caused by the command 'mb_auto_configure_abstract_classes') the connection rule is extended to the abstract class X', as follows:

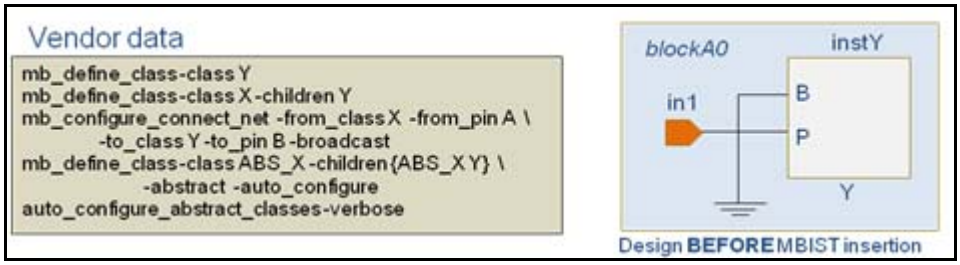
```
# Vendor commands after elaboration
mb_define_class –class X –children {X′ Y}
mb_configure_connect_net –from_class X   –from_pin A
     -to_class Y    –to_pin B
     -to_class X′   –to_pin A′   -broadcast
mb_configure_connect_net –from_class X′ –from_pin A′
      -to_class Y     –to_pin B     -broadcast
```

The effect of this elaboration can be visualized in the following diagram. During the bottom-up flow the new port A' is created on the boundary of the lower level block. At a higher level, if the object of class X needs to connect to the object of class Y inside the BIST-inserted block then they communicate through the new port A'. This is shown in the above elaborated vendor data. In addition, if an object of class X must be connected to an object of class Y at the higher level (not inside the abstract class object) then they can communicated directly (between port A and B) as defined in the 'mb_configure_connect_net' rule.

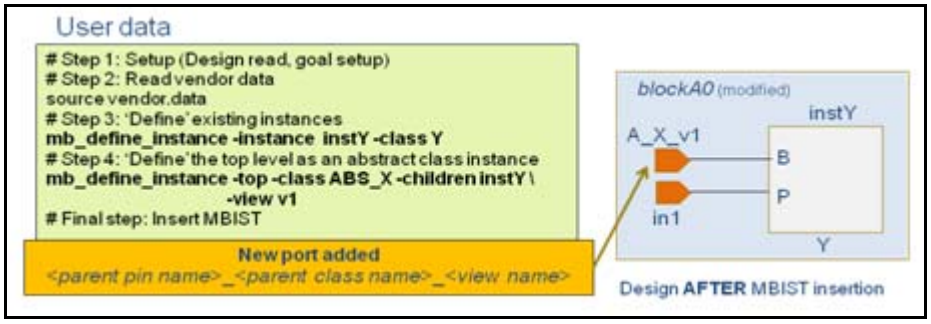


It may be useful to understand how the intermediate ports are created at the boundary of the abstract class objects during the bottom-up flow. The

following example shows a block (blockA0) which will be BIST-inserted according the rules defined in the vendor file shown below.



This block (blockA0) is now BIST-inserted per the user data shown below.

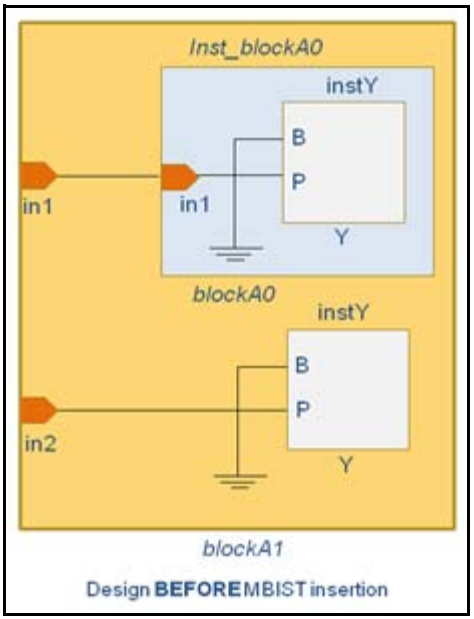


Note that an additional port (A_X_v1) is created to make provision for future connection of port B (of the child class instance in the abstract class object) to pin A of the parent class (X) object at a higher level. Notice the naming convention for the additional test port of the modified design is:

<port name of the parent class>_<parent class name>_<view name>

This mechanism allows higher level objects to 'uniquely' refer to the instances inside the lower level BISTed block (abstract class object).

The following example illustrates how the above BIST-inserted block (modified blockA0) can be used at a higher level inside the block 'blockA1'. The original SoC (before BIST insertion at any level of hierarchy) looks like the following:



The bottom-up BIST insertion flow consists of the following steps:

1. Insert BIST on the lower level block ‘blockA0’. Consider that this has already been done as we have shown before.
2. Read the description of ‘blockA1’. At the same time read the description of the ‘modified blockA0’.
3. Insert BIST at the level of ‘blockA1’.

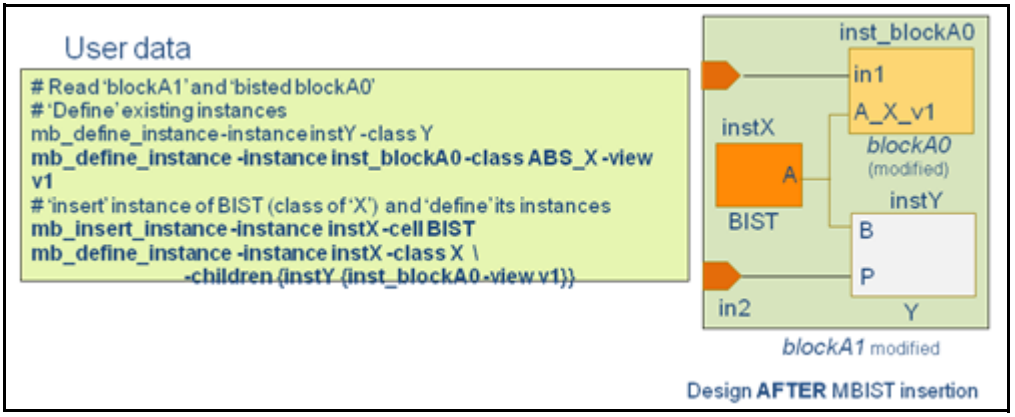
We will use the same ‘vendor file’ as before (use the same class relationships and connection rules).

```
mb_define_class-class Y
mb_define_class-class X-children Y
mb_configure_connect_net -from_class X -from_pin A\
          -to_class Y -to_pin B –broadcast
mb_define_class-class ABS_X-children {ABS_X Y} \
                -abstract -auto_configure

auto_configure_abstract_classes-verbose
```

We will now insert an object of class X (an instance of module ‘BIST’) at the level of ‘blockA1’. The connections to the various class objects are

automatically established by the tool. The ‘user data’ and the final structure of the modified blockA1 will look as follows:

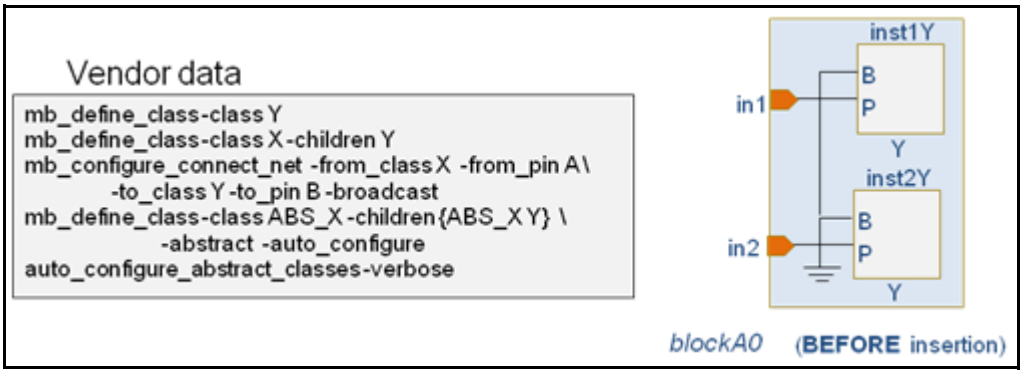


Notice that the desired connection between the object of class X (instX) and object of class Y (blockA1.instY and inst_blockA1.blockA0.instY) have been established between ports A (of the parent class object) and port B (of the child class object).

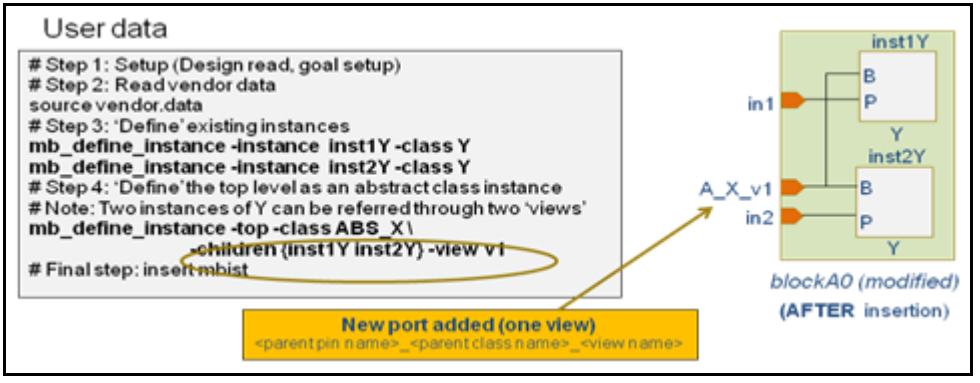
Implication of a View as Defined for an Abstract Class Object

While discussing ‘view’ in a previous section, we understood that it is a convenient mechanism for a parent class object at a higher level to refer to specific instances inside an abstract class object. In this section, we will further extend this concept and understand the creation of the additional test ports for future communication with the parent class objects placed at the higher level.

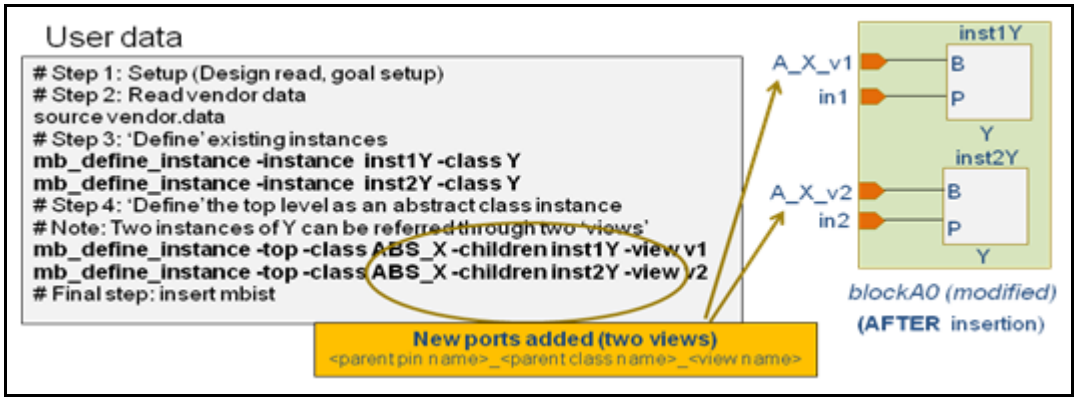
Let us modify theblockA0 example to instantiate a second instance of Y. The same vendor file with the same connection rule between the parent class X and child class Y is used.



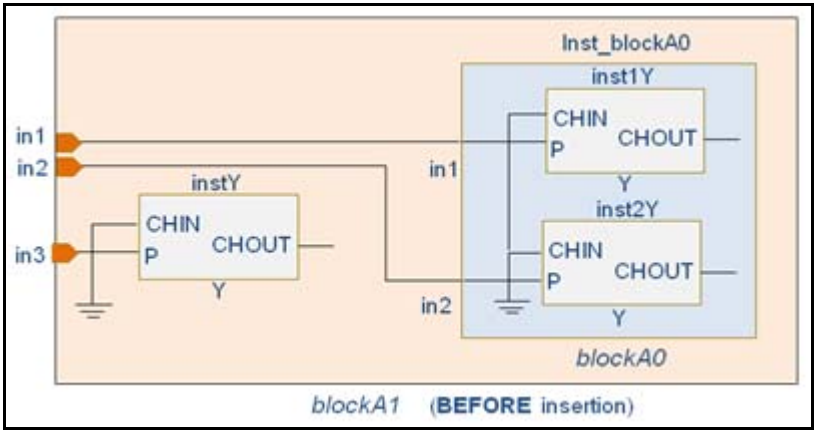
To process this block, the designer defines the abstract class instance with a single view for both the instances. Note the 'mb_define_instance' in the user data file. The resultant structure is shown in the schematic. Notice that only one additional port is created.



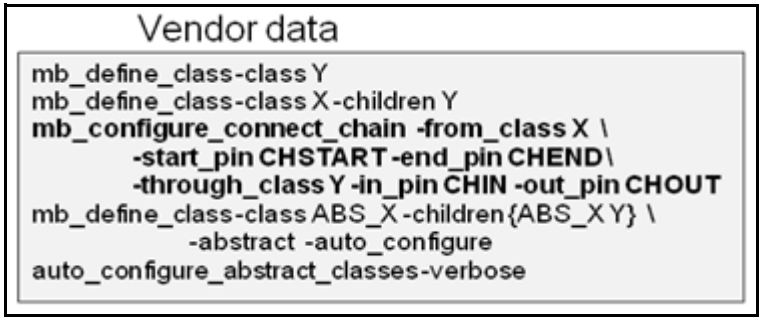
Contrast this scenario when the designer decides to create two views, instead of a single view, for each of the instances of Y. Notice the user data file with two views and the resultant modified blockA0 with two additional ports instead of one, as follows:



We can also cite another illustration where we will establish a chain connection through a lower level block. Consider the following SoC level design. The topmost level module is 'blockA1' that instantiates module 'blockA0', as shown in the following schematic.



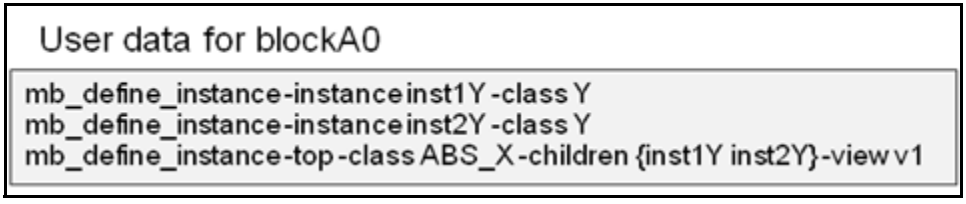
The vendor data defines a daisy chain in this SoC with the 'mb_configure_connect_chain' command in the following vendor data file. This command sets up a rule for establishing a daisy chain connection among various class objects. The same vendor file will be used for the entire bottom up flow:

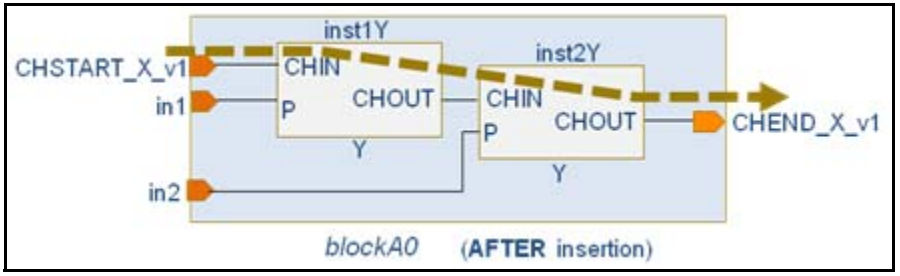


The steps for the execution of the bottom-up flow are:

1. Process ‘blockA0’
2. Read the description of ‘blockA1’
3. Read the ‘modified’ description for ‘blockA0’
4. Insert an instance of class of X
5. This is the final step. Tool automatically establishes the chain as desired in the vendor file.

As we have noticed earlier, in a bottom-up flow the number of views will determine the way the additional port will be created. As a result, this will also influence the way chains will be connected. Let us assume that the designer wants to create a single view for both instances inside blockA0 (note the user data file for this block). As a result, two additional ports are created for the two endpoints of the chain for this block after insertion.



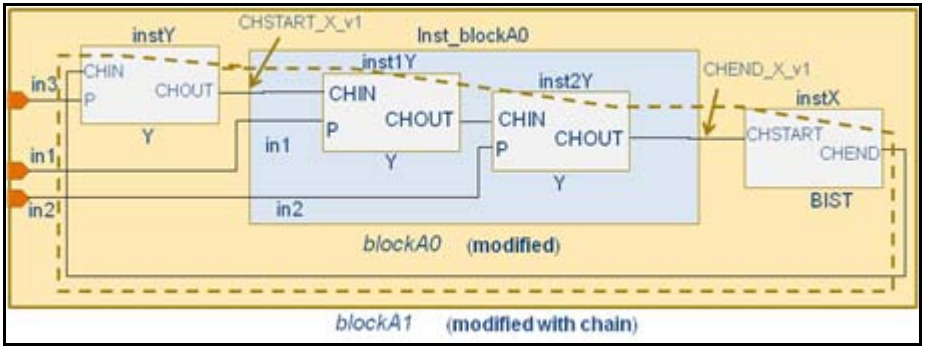


The top level (blockA1) is processed next with the following user data file. Notice, one view of the modified blockA0 is considered.

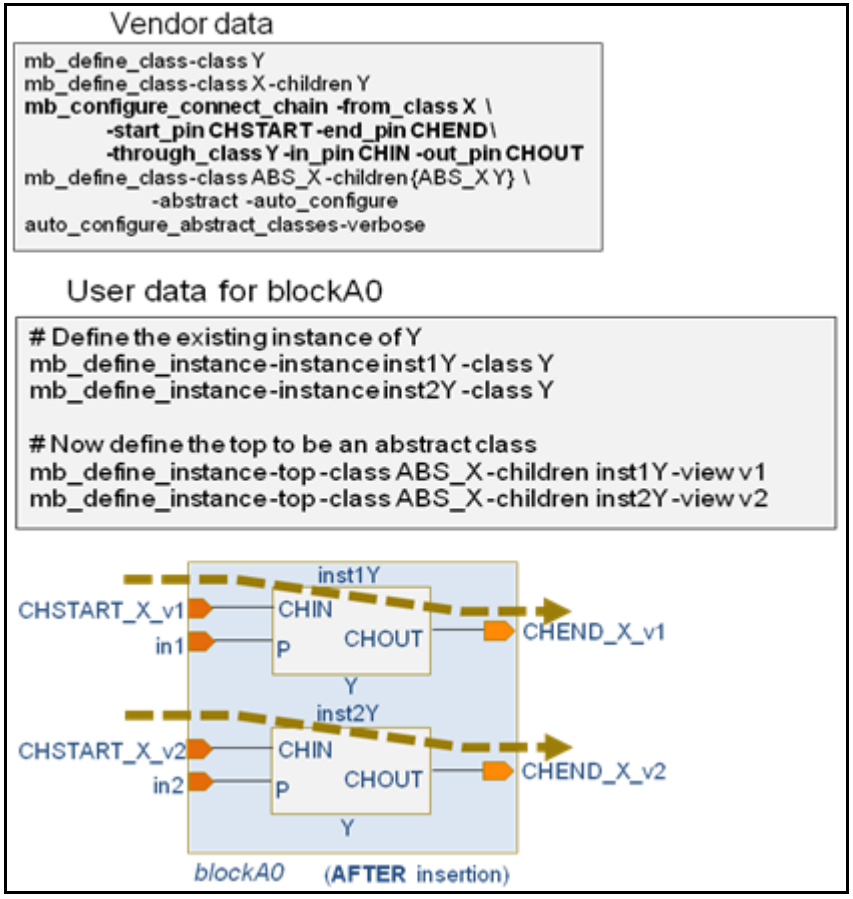
User data for blockA1

```
# Define the existing instance of Y
mb_define_instance -instance instY -class Y
# Define the  instance of 'modified' blockA0 with 'one' view
mb_define_instance -instance inst_blockA0 -class ABS_X -view v1
# Insert the instance of the class X, and define its instances
mb_insert_instance -instance instX -cell BIST
mb_define_instance -instance instX -class X -children {instY {inst_blockA0 -view v1}}
```

The following is the resultant structure for modified blockA1. Note the chain configuration.



Contrast this when the designer decides to consider two (2) views for the instances inside blockA0. The following is the result of the design modification in the two stages of bottom-up flow. We use the same vendor data as before.



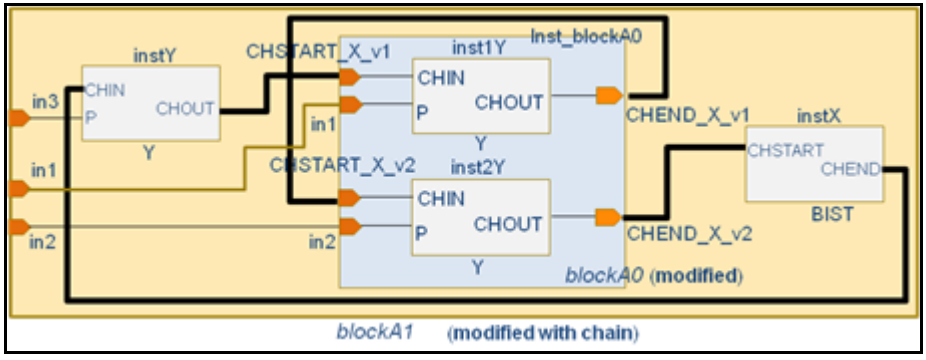
You may note that two additional ports have been created for the two possible chains in order to access two different views from outside this block.

User data for blockA1

```
# Define the existing instance of Y
mb_define_instance -instance instY -class Y
mb_define_instance -instance inst_blockA0 -class ABS_X -view v1
mb_define_instance -instance inst_blockA0 -class ABS_X -view v2

# Insert the instance of the class X
mb_insert_instance -instance instX -cell BIST
mb_define_instance -instance instX -class X \
                   -children {instY {inst_blockA0 -view v1} {inst_blockA0 -view v2}}
```

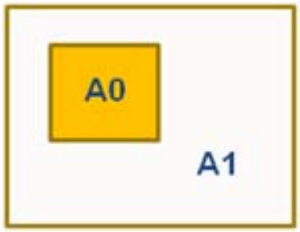
The blockA1 is modified to establish the daisy chain as shown in the schematic below.



Shell Creation during Bottom-up Flow

Once the lower level block is processed, it does not need to be processed again during the MBIST insertion at the higher level. In order to expedite the processing of the higher level block, a ‘shell’ view for the lower level block can be used. This ‘shell’ view of the lower level block is created during the insertion of the lower level block.

Consider the following design hierarchy:



In a bottom-up flow, block A0 is processed as the first step and block A1 is processed as part of a second step. The sequence of steps and the activities that happen during these steps are described below:

Step 1: Processing of block A0

1. Design read: Read the source file for the RTL for A0
2. Insert MBIST in block A0
3. During this step, by default, the shell view (post-BIST) for block A0 is created in a file named 'atrenta_generated_abstract_block.f'. This file is intended to be used during the design read of A1.

NOTE: *The file 'atrenta_generated_design.f' is always generated. This file contains the list of all the files representing the post-BIST block A0.*

Step 2: Processing of block A1

1. Design read: Read all the source files for A1. This may or may not include the 'original;' description of A0. During this step read the 'shell' view of the processed block A0

   ```
   read_file –type sourcelist prj_A0/mbist_dft/
   spyglass_reports/rme/atrenta_modified_rtl_files/
   atrenta_generated_abstract_block.f
   ```

2. Apply a blackbox directive on A0

   ```
   set_option stop A0
   ```

3. Insert MBIST in block A1
4. During this step, the shell view for processed A1 will be generated. This 'shell' view may be used during the processing at a higher level

This methodology has demonstrated significant run time improvement in a bottom-up flow. A complete example can be found in *Appendix D: An Example of Shell View Created during Bottom-up Flow*.

Working with Gate-Level Designs for MBIST Insertion

The SpyGlass DFT MBIST product described so far may also be used for gate-level designs.

Although there is no special treatment necessary, we recommend the use of the following optional parameters for achieving run time efficiency.

- **mbist_flow**

  This parameter tells the tool whether the tool is run at "gate" or "rtl".

  ```
  set_parameter mbist_flow gate
  ```

  This parameter will direct the auto-fix engine to eliminate unnecessary bookkeeping and API calls as normally necessary at the RTL. As an example, at the gate-level, the Verilog language construct "generate" is not expected. If the parameter is specified then the design modification engine skips checking for the ‘generate’ constructs in a gate-level design.

  Another parameter is useful in case of working with a gate-level design. At smaller gate-level designs, there may be negligible difference in run-time with or without the parameter.

- **mbist_generate_single_netlist**

  By default, the tool generates a single netlist at the gate-level, which means that all BIST and design definitions are written out in a single output file. This feature can be disabled with the script:

  ```
  set_parameter mbist_generate_single_netlist off
  ```

- **mbist_write_all_parameters**

  This parameter is used to write all the parameters while instantiation. By default, the value of the *mbist_write_all_parameters* parameter is set to off. In this case, only the parameters, which are overridden through the `parameter_map` option in the `mb_insert_instance` command are written.

  ```
  set_parameter mbist_write_all_parameters on
  ```

TIPs and FAQs

Q: How do I get more information about a specific Tcl command while I am inside the SpyGlass Tcl shell?

Ans: Use the '-help' switch available to every Tcl command, for example

```
sg_shell> mb_configure_connect_gate -help
Command Usage:
   -type <string>       : gate type (OR | AND | XOR)
   [-cell <string>]     : specific cell to be used for logic
   [-use_as_buffer ]    : if necessary use the cell as a
buffer
   -from_class <string>*       : connect_gate from this class
   -from_pin <string>*  : from pin
   -to_class <string>   : connect_gate to this class
   -to_pin <string>     : to pin
   [-max_children <integer>]    : maximum number of child
instances to be connected
   [-id <string>]       : configuration rule ID
   [-help ]     : prints this help message
0
sg_shell>
```

More information is available in the *SpyGlass DFT MBIST Tcl Flow User*

Manual.

Q: How to capture the stdout into a report?

Ans: Use the "capture" command of SG_SHELL

eg: **capture -stdout error.log {source test.tcl}**

“error.log” is the user-defined file. Note: when this command is used, then no messages/output are thrown on the stdout. (similar to ">" operator in UNIX).

```
sg_shell
sg_shell> capture –stdout error.log {source test.tcl}
```

Q: If I have the following line in the vendor file:

```
# any comment with \
foreach {fr_pin} {O} { \
mb_configure_connect_net -from_class M -from_pin O \
        -through_class CTRL2 -in_pin {in1 in2} -out_pin
{out1 out2} \
        -parallel \
}
```

I found that the entire line is considered as a comment (i.e. the mb_configure_connect_net command is not visible)?

Ans: One subtle effect to watch out for is that a backslash continues a comment line onto the next line of the script. A semicolon inside a comment is not significant. Only a newline terminates comments:

```
# Here is the start of a Tcl comment \
and some more of it; still in the comment
```

source: *http://www.beedub.com/book/2nd/tclintro.doc.html*

http://www.beedub.com/book/2nd/os.doc.html

Q: I have the following command in the vendor data file:

```
mb_configure_connect_net –from_class bist –from_pin
ti\[9:5\] \
```

```
        -to_class ram_blk –to_pin mem_pin\[4 : 0\] –
bitwise_broadcast
```

and get the following error:

```
Error: Invalid option ‘:’
```

The error goes away if I remove the extra space around ‘:’ in the command:

```
mb_configure_connect_net –from_class bist –from_pin
ti\[9:5\] \
        -to_class ram_blk –to_pin mem_pin\[4:0\] –
bitwise_broadcast
```

Ans: TCL parser is sensitive to space, so the field `mem_pin\[4 : 0\]` is not a single string. It is broken into three different strings (`mem_pin\[4` , `:` and `0\]`) by the TCL parser. Hence, when the arguments are passed by the TCL parser to the MBIST code, the intention of the user is lost. Also `{mem_pin[4 : 0]}` would not work for the same reason.

SpyGlass uses the standard open-source TCL parser (version 8.5).

A few valid ways to use part select are:

```
-from_pin mem_pin\[2\]
-from_pin {mem_pin[3:5]}
-from_pin {mem_pin[1:2]  mem_pin[3]}
-from_pin {mem_pin[5] mem_pin[2]}
```

Q: My vendor data looks like:

```
mb_define_class -class mem_class
mb_define_class -class bist_class -children mem_class

mb_configure_connect_net -from_class {mem_class } -from_pin
mem_pin \
          -to_class {bist_class } -to_pin bist_pin –parallel
```

I get the following error:

```
#Undefined -from_class 'mem_class '

#Undefined -from_class 'bist_class '
```

Ans: One needs to be careful about using extra white spaces with the

object names in Tcl.

The string "mem_class" and "mem_class " are not equal because of the extra white space in the mb_configure_connect_net statement. This is related to the Tcl parser and not MBIST code.

Q: How do I specify a list of arguments in the user data file with double braces? For example, I am trying to supply an argument value map in the ‘mb_insert_instance’ command, as follows:

```
mb_insert_instance -instance I2 -cell WRAP_FUSECNTL_T09 \
            -parameter_map {{NUM_BAYS 23 MEM_ADDR 10}}
```

Ans: The following are examples that will work (not the usage of double braces):

```
-parameter_map {a 1}
-parameter_map {a 1 b 2}
-parameter_map {{a 1}}
-parameter_map {{a 1} {b 2}}
-parameter_map [list a 1 b 2]
```

The following usage will not work and error message will be generated:

```
          -parameter_map {{a 1 b 2}}
          -parameter_map {{{a 1 b 2}}}  … or more braces
```

Appendix A describes the recommended use of braces in the Tcl file.

Q: My RTL has memories instantiated inside ‘generate’ statements. How do I represent them in the user data file?

Ans: Suppose you have the following RTL:

```
generate
    for (j = 0; j < (2**U_ADDR_BIT); j = j + 1) begin : MEM
      assign dec[j] = (a[(D_ADDR_BIT+U_ADDR_BIT)-
1:D_ADDR_BIT] == j);
      assign _ce[j] = ~(~ce & dec[j]);

      MEM inst (.CK(ck), .CE(_ce[j]), .WE(we),
                .A(a[D_ADDR_BIT-1:0]), .I(i),
```

```
                .O(_o[DATA_BIT*(j+1)-1:DATA_BIT*j]),
                .BWE(bwe));
    end
endgenerate
```

This will result in SpyGlass DFT MBIST product finding the memories in the following names:

“\MEM[0].inst “, “\MEM[1].inst “, “\MEM[2].inst “, etc.

The recommended way to use these names in the user data file is as follows. Note, in order to preserve the space character (required character due to the escape character ‘\’) or “ ”, mention one item in one list:

```
mb_define_instance -instance {sram1p/\MEM[0].inst } -class
1PMEM
mb_define_instance -instance {sram1p/\MEM[1].inst } -class
1PMEM
mb_define_instance -instance {sram1p/\MEM[2].inst } -class
1PMEM
mb_define_instance -instance {sram1p/\MEM[3].inst } -class
1PMEM
:
mb_insert_instance -instance sram1p/collar_2 -cell
collar_1p_2
mb_define_instance -instance sram1p/collar_2 -class
COLLAR_1P_2 \
        -children { \
                  {sram1p/\MEM[2].inst } \
                  {sram1p/\MEM[3].inst } \
                   }
```

Q: I have a memory in the design for which I want to swap certain connections with the existing logic after insertion of a collar. The pseudo-code to specify this connection is as follows:

```
# Insert the collar (“CTRL")
add_instance {ctrl_inst1} -cell CTRL

# move the driver of mem_inst1/A and connect it to the
collar's 'sys_A‘ pin
```

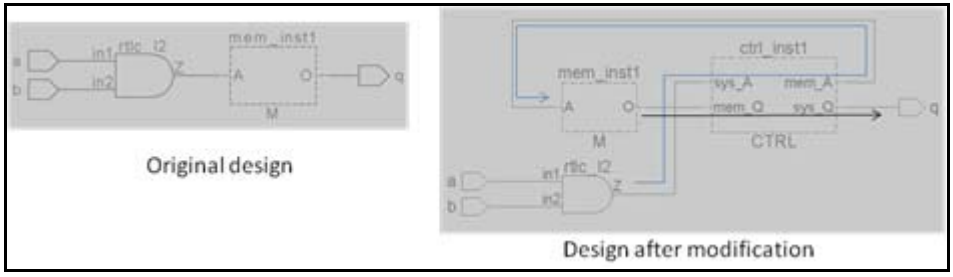
```
move_driver_of /mem_inst1/A -to /ctrl_inst1/sys1_A

# Finally make the connection between collar and memory
add_connection -from /ctrl_inst1/mem_A -to /mem_inst1/A

# Similarly on the output ('load') side of the memory
move_load_of /mem_inst1/O -to /ctrl_inst1/sys_Q
add_connection -from /mem_inst1/O -to /ctrl_inst1/mem_Q
```

How do I accomplish such connections during BIST insertion?

Ans: Per your directive, the following is what you want to achieve:



The following SpyGlass-MBIST Tcl commands will help achieve this goal:

```
# On the input side
mb_configure_connect_net -from_class M -from_pin A \
         -to_class CTRL -to_pin sys_A \
         -parallel \
mb_configure_connect_net -from_class CTRL -from_pin mem_A \
         -to_class M -to_pin A \
         -parallel \

# On the output side
mb_configure_connect_net -from_class M -from_pin O \
         -through_class CTRL -in_pin mem_Q -out_pin sys_Q \
         -parallel \
```

Q: We have some BIST engines (DRAM BIST, ROM BIST) that run-off of the customer's at-speed functional clock

Is there any way that we can have the tool connect the BIST input clock to

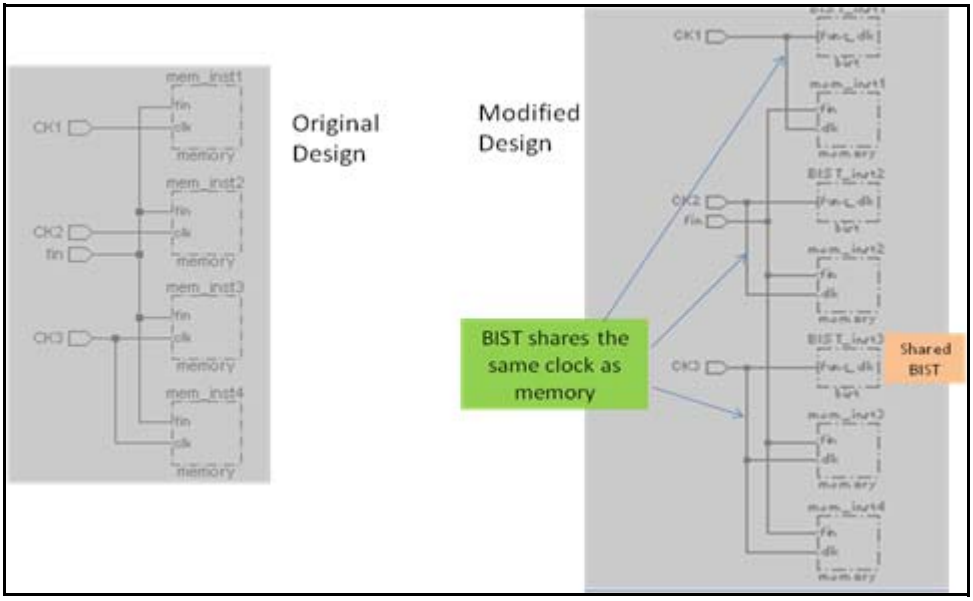
the same functional clock and the associated memories? In other words, the following are the objectives:

- The customer connects their functional clock to the memory, and then SpyGlass adds the BIST engine and we want SpyGlass to connect the BIST input clock to the same net that is connected to the memory functional clock pin.
- If the BIST is shared/associated with multiple memories, then all the memories would need to be connected to the same functional clock.

Ans: Use the 'mb_configure_connect_net' command with the '-single_connection' switch. The vendor file should look like:

```
# Vendor data
mb_define_class -class MEMORY
mb_define_class -class BST -children MEMORY
mb_configure_connect_net -from_class MEMORY -to_class BST \
                      -from_pin clk      -to_pin func_clk \
                        -parallel –single_connection
```

Now consider a pre-BIST design with two memories with two dedicated clocks and two memories that share the same clock. In the following diagram, 'mem_inst1' and 'mem_inst2' are the two memory instances with dedicated clocks. Memories 'mem_inst3' and 'mem_inst4' both share one functional clock.



The desired ‘modified’ structure is shown in the above diagram. Note that the inserted BIST components are connected to the specific functional clocks of the respective memories. The user data is shown below:

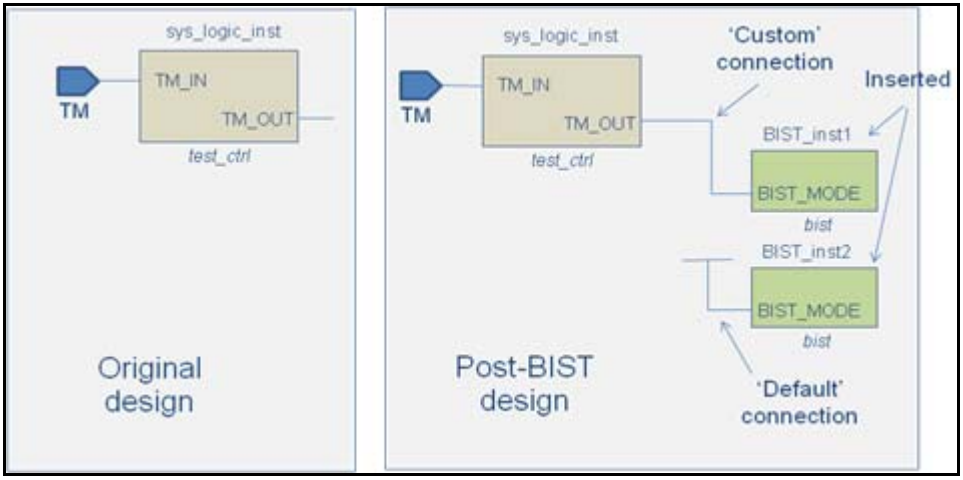
```
# User data
# This bist instance has one memory
mb_insert_instance -instance BIST_inst1 -cell bist
mb_define_instance -instance BIST_inst1 -class BST -children
mem_inst1

# This bist instance has one memory
mb_insert_instance -instance BIST_inst2 -cell bist
mb_define_instance -instance BIST_inst2 -class BST -children
mem_inst2

# This bist instance is a ‘shared’ one and controls two
memories
mb_insert_instance -instance BIST_inst3 -cell bist
mb_define_instance -instance BIST_inst3 -class BST \
                     -children {mem_inst3 mem_inst4}
```

Q: How to selectively override vendor-directed connections by custom connections? We have BIST engines that, when inserted, will have their BIST_MODE pins connected to VCC. This is the ‘default’ connections. However, we want to connect these pins for a few ‘selected’ instances to system test logic.

The intention is illustrated in the following diagram:



The instance ‘sys_logic_inst’ is system test logic that is already present in the original design. As part of the insertion process, the designer wants to insert two instances of ‘bist’ and have their ‘BIST_MODE’ pins connected to VCC, by default. However, the designer wants to override this ‘default’ connection of the instance ‘BIST_inst1’ with a custom connection from ‘sys_logic_inst’.

Ans: The ‘default’connection is the result of a vendor directive as follows. Note, this command has an ‘Id’ that will be referenced in the scripts.

```
# Vendor data; This rule directs the BIST_MODE pin of all the
# instances to be connected to ‘1’
mb_configure_connect_net -from_class abs_TOP -from_pin 1 \
                  -to_class BST -to_pin BIST_MODE -broadcast \
                    -id CCN_BIST_MODE_TO_1
```

Any connection specified using the mb_connect_net command automatically overrides the default connection configuration specified through *mb_configure_connect_net* command.

```
# Custom connection to override the vendor-specified
connection

mb_connect_net -from sys_logic_inst/TM_OUT \
-to BIST_inst1/BIST_MODE -parallel mb_insert
```

Another mechanism to override the vendor connection is to rewrite the user data file to skip the rule-driven connection for a specified instance with the -skip_connection switch in the mb_define_instance command.

Following describes the user data file that will accomplish the desired objective:

```
mb_insert_instance -instance BIST_inst1 -cell bist
################################################
# The connection rule specified by 'mb_configure_connect_net'
# with the 'id' will be skipped for this instance only.
################################################
mb_define_instance -instance BIST_inst1 -class BST \
-skip_connection CCN_BIST_MODE_TO_1
mb_insert_instance -instance BIST_inst2 -cell bist
mb_define_instance -instance BIST_inst2 -class BST
# Custom connection to override the vendor-specified
connection
mb_connect_net -from sys_logic_inst/TM_OUT \
-to BIST_inst1/BIST_MODE -parallel mb_insert
```

Q: We use ‘sg_shell’ from a perl script to drive the MBIST insertion, as follows:

```
myScript.pl –tcl test.tcl
```

However, if there is any error encountered inside ‘test.tcl’ (for example a bad ‘mb_configure_connect_net’ command, or a ‘mb_define_instance’ applied on a non-existent instance, or a ‘mb_connect_net’ on a non-existent node, etc.), then the control does not come back to my perl script. I see the ‘sg_shell>’ prompt and exiting from this requires user intervention. Is there a way to avoid this situation?

Ans: There are two ways a user can use ‘sg_shell’ in a script.

sg_shell < test.tcl (re-direction)

(will always exit out, regardless of pass or fail).

- sg_shell -tcl test.tcl

(this is for a startup file scenario) so control would remain in sg_shell until exit is provided explicitly.

In case some errors occur while processing the test.tcl file then the processing is stopped and the remaining test.tcl file is not parsed.

Q: I notice that a module name has been changed (to reflect the user’s preference). However, the name of the file containing the module has NOT changed. To summarize, the user’s preference is:

```
mb_set_prefix_n_suffix
-module_prefix atrenta_MYTOP \
-file_prefix atrenta_MYTOP \
-file_suffix sg
```

The following file has been created with the content shown below. Notice that the module name has changed, but the file name has not.

```
cd
prjdir/prj_MYTOP/mbist_dft/spyglass_reports/rme/mbist-dft/
atrenta_modified_rtl_files
more WRAPPER_SRAM2SFBM.v_sg        fl- Note the file name

module atrenta_MYTOP_WRAPPER_SRAM2SFBM(    fl Note the module
name
//Atrenta inserted ports , done on 8-Febuary-2011  SpyGlass
4.5.0 policy mbist-dft 4.5.0
   Bist_CNTL,
   Bist_EN,
   Bist_RETURN,
   :
```

Ans: Note that original file contains more than one module. Therefore, the file name has not changed. The following two conditions must be satisfied for changing the file name:

- The file name and module names must match in the pre-BIST RTL
- The file must not contain more than one module

Also note, it is the user's responsibility to make sure that appropriate module and file prefixes are supplied so that the module and file names match in the post-BIST RTL.

Q: How to avoid generating any 'file_prefix' for the modified output file with the 'mb_set_prefix_n_suffix' command?

Ans: The Tcl command has no mechanism to prevent file name changes (i.e. not add any file prefix). In other words, the following is not allowed:

```
mb_set_prefix_n_suffix -module_prefix atrenta_${design} \
             -file_prefix “” \    fl Not allowed in Tcl
             -file_suffix sg \
              -wire_prefix atrenta_wire
```

Workaround is to read a SGDC file with the 'rme_config' command to specify the intention:

```
rme_config  [-module_prefix  <string>]  \
            [-file_prefix <string>]     \    <-“” is possible
            [-file_suffix <string>]     \
             [-wire_prefix <string>]
```

Read this sgdc file in the user data file as follows:

```
read_file –type sgdc <sgdc file name>
```

In this case, you will not need to supply the 'mb_set_prefix_n_suffix' command.

(A more robust solution planned for future is to make provision of a keyword 'NO_PREFIX' to be supplied as an argument of this Tcl command (VI-67497))

Q: How to find the SpyGlass version in sg_shell?

Ans:

```
set version [lindex [exec cat $::SPYGLASS_HOME/lib/SpyGlass/
.spg_version] 0]
```

Q: What is the difference between running SpyGlass DFT MBIST product in ‘preview’ and ‘non-preview’ mode?

Ans:

In preview mode, we set up everything, but do not do actual insertion

In non-preview mode, the actual insertion takes place

Preview mode helps ensure that the class definitions and the associated rules (mb_configure_connect_net, etc.) are correct, if the instances are associated with the correct class structures etc. In summary, the preview mode helps confirm that the vendor data (independent of the design) and the user data (dependent on the design) are in compliance. Preview mode also generates intermediate files (for example sgdc files) that are transparent to the user, but used by the tool. .

Q: How to capture the output of ‘sg_shell’ into a log file?

Ans: Use the ‘-shell_log_file <file name>’ option. For example:

```
sg_shell -shell_log_file abc.log -tcl test.tcl
```

or

```
sg_shell -shell_log_file abc.log < test.tcl
```

Q: I have a generate statement in my RTLand it creates instance names with ‘escaped’ characters. I am having trouble in specifying related names as arguments to ‘mb_connect_net’:

```
mb_connect_net \
-from "HC.cpu.dcu.\arrays/dataArrays/
dataSramC0_BIST1RACT09.SYSDONE "  \
-to SYSDONE_2  \
-parallel  \
-add_port
```

This causes a fatal message to be reported:

```
MBist_SGDC_03    MBist_SGDC_03    Fatal       Non-existing
instance 'HC.cpu.dcu.\arrays/dataArrays/
dataSramC0_BIST1RACT09.SYSDONE specified.
```

Ans: The from field of mb_connect_net command expects a list to be supplied. The escaped name requires that the value must be enclosed

within braces (since there is one space character). To unambiguously specify this command, you need to enclose this value within double braces:

```
mb_connect_net \
-from {{HC.cpu.dcu.\arrays/dataArrays/
dataSramC0_BIST1RACT09.SYSDONE }}  \
-to SYSDONE_2  \
-parallel  \
-add_port
```

The following are valid examples of Tcl command usage:

```
mb_connect_net -broadcast  \
   -from inst_bist.BSTART \
   -to [list inst1.BEN inst2.BEN {{inst_ram.\L1[2].inst
.BEN}} \
      {{inst_ram.\L1[1].inst .BEN}} {{inst_ram.\L1[0].inst
.BEN}}  ]
```

```
mb_connect_net -broadcast  \
       -from inst_bist.BSTART \
       -to {inst1.BEN inst2.BEN {inst_ram.\L1[2].inst .BEN} \
           {inst_ram.\L1[1].inst .BEN} {inst_ram.\L1[0].inst
.BEN}}
```

Q: I am trying to implement the following using the SpyGlass design query commands:

```
mb_connect_net -from top.tst_mbist_mode \
               -to controller.mode_sel -parallel
```

Where, both mode_sel and tst_mbist_mode are input busses of width [3:0]. I am using the design query commands in my script as follows:

```
puts $fileId2 "\n mb_connect_net -from [get_attribute
[get_ports -regexp {{.*\.tst_mbist_mode}}] full_name] \\
-to {$main_controller_inst.mode_sel} -parallel"
```

This gives the output as

```
mb_connect_net -from {top.tst_mbist_mode[0]}
```

```
{top.tst_mbist_mode[1]} {top.tst_mbist_mode[2]}
{top.tst_mbist_mode[3]} \
-to {top.controller.mode_sel} -parallel
```

While running SpyGlass it reports a fatal error in the <top>.sgdc file generated during the run.

Ans: You can use ‘list’command of Tcl to specify the multiple entries to an option

```
puts $fileId2 "\n mb_connect_net -from [list [get_attribute
[get_ports -regexp {{.*\.tst_mbist_mode}}] full_name]] -to
{$main_controller_inst.mode_sel}  -parallel"
```

This will result in

```
mb_connect_net -from {{top.tst_mbist_mode[0]}
{top.tst_mbist_mode[1]} {top.tst_mbist_mode[2]}
{top.tst_mbist_mode[3]}}  -to {top.controller.mode_sel}  -
parallel
```

Appendix A: Guidelines of Using Braces in Tcl Files Specified to the SpyGlass DFT MBIST Product

SpyGlass DFT MBIST product users have found that there are various ways to use braces in the User data (Tcl) file for driving the MBIST insertion and that Tcl is rather ‘ambiguous’ in interpreting a single value vs. comprehensive lists. This puts unnecessary overhead and complication to the automated scripts often used to generate the User data files.

This document is an effort towards standardizing the usage of braces and describes the guidelines for use with SpyGlass DFT MBIST product. Adhering to these guidelines will ensure that the users’ intention will be unambiguously interpreted by SpyGlass DFT MBIST product.

The following discussion focuses on specifying hierarchical instances, nets, and pin-names in SpyGlass TCL commands.

Key Points of Consideration

User needs to differentiate between an option that can take a single value or multiple values as input.

Guideline for Options Accepting a Single Value

```
-option I1.I2
-option {I1.\I2.I3 .I4}    => Note we have put an extra brace
due to escape name
```

Guideline

For ease of scripting, user can always generate with a brace:

```
-option {value}
```

Guideline for Options Accepting Multiple Values

```
-option {I1.I2}
-option {I1.I2 I3.I4}
-option {{I1.I2} {I3.I4}}
-option {{I1.\I2.I3 .I4}}   => Note we have put an extra
 brace due to escape name
-option {{I1.\I2.I3 .I4} I5.I6 I7.I8}
-option {{I1.\I2.I3 .I4} {I5.I6} {I7.I8}}
-option {{I1.I2}}
```

Guideline

For ease of scripting, user can always generate with a brace:

```
-option {{value1} {value2} {value3}}
```

Guideline for Options Accepting regexp and a Single Value

```
-option {a*b*cd}
```

```
-option abcd        => note no braces required as no wildcard
 character
```

Guideline

For ease of scripting, user can always generate with a brace:

```
-option {value}
```

Guideline for Options Accepting regexp and Multiple Values

```
-option {{a*B*c}}
-option {{p*q*r} {*m*o}}
```

Guideline

For ease of scripting, user can always generate with a brace:

```
-option {{value1} {value2} {value3}}
```

Tcl Commands in the SpyGlass DFT MBIST Product

- **mb_insert_instance**

  Syntax:

```
mb_insert_instance
        -instance       <single_value>
        -cell           <single_value>
        -parameter_map  <multiple_value>
```

  Example:

```
mb_insert_instance -instance {inst_name} -cell ABCD
mb_insert_instance -instance {inst_name} -cell ABCD \
          -parameter_map {{param value}} => Note that
{param value} is a single object
mb_insert_instance -instance inst_name   -cell PQR  \
           -parameter_map {{param1 value1} {param2 value2}}
```

- **mb_replace_instance**

  Syntax:

```
mb_replace_instance
        -instance       <single_value>
        -cell           <single_value>
```

  Example:

```
mb_replace_instance -instance {inst_name} -cell ABCD
```

- **mb_define_instance**

  Syntax:

```
mb_define_instance
        -instance       <single_value>
        -children       <multiple_value>
        -view           <single_value>
        -class          <single_value>
```

Example:

```
mb_define_instance -instance {\L2[0].a0 } -class ABS_BIST
-view v1
mb_define_instance -instance I1.I2 -children {I3.I4 I5.I6}
mb_define_instance -instance {I1.I2} -children {I3.I4}
mb_define_instance -instance {I2.I5} -children {{\I7.I8 }
{I9.I2}}
mb_define_instance -instance {\I6.I7 .I8} -children {{I3
-view V1}}         => Note that {I3 -view V1} is a single
object
mb_define_instance -instance {\I7.I8 } -children {{I3 -
view V1} {I4 -view V2}} -view V3
mb_define_instance -instance I1.I2 -children {{{\I7.I8
.I9} -view V1}} => Note the extra braces for escaped name
mb_define_instance -instance {I1.I2} -children {{{\I7.I8
.I9} -view V1} {{I2.I5} -view V3}}
mb_define_instance -instance b1_1 -class BIST -children
{{{\L2[0].a0 } -view v1} {{\L2[0].a0 } -view v2}
{{\L2[0].a0 } -view v3}}
mb_define_instance -instance b1_3 -class BIST -children
{{w12_3}}
mb_define_instance -top -class ABS_BCTL -children {{b1_1}
{b1_2} {b1_3}} -view v1
mb_define_instance -instance b_1 -class BIST -children
{{\MYRLMB[0].MYRLMB_inst .W_0.INTWRAP}}
mb_define_instance -instance b1_1 -class BIST -children
{{{a0.\L1[0].inst } -view v1}}
```

Note that there may be extra space characters. They are in general acceptable:

```
mb_define_instance -instance a1 -class B -children { {A} }
mb_define_instance -top -class ABS_B -children { {A} } -
view v1
mb_define_instance -instance b1_1 -class BIST \
                -children { { {a0.\L1[0].inst } -view v1} }
```

```
mb_define_instance -instance {b1_2} -class BIST \
                -children {{{a0.\L1[1].inst } -view v2}
{{a0.\L1[2].inst } -view v3} w12_2}
mb_define_instance -instance {b1_2} -class BIST \
               -children { { {a0.\L1[1].inst } -view v2} {
{a0.\L1[2].inst } -view v3} w12_2}
```

- **mb_remove_instance**

  Syntax:

```
mb_remove_instance
        -instance       <multiple_value>
        -cell           <multiple_value>
```

  Example:

```
mb_remove_instance -instance {{*term*} {*I}} -cell {TERM}
mb_remove_instance -instance {{*term*}}  -cell {{*TERM*}}
```

- **mb_connect_net**

  Syntax:

```
mb_connect_net
        -from            <multiple_value>
        -to              <multiple_value>
        -through_in_pin  <single_value>
        -through_out_pin <single_value>
        -select_enable   <single_value>
```

  Example:

```
mb_connect_net -from {a.pin b.pin} -to {c.pin d.pin}
mb_connect_net -from {{\a.c .b.pin[2]} {I1.pin}} \
            -to_pin {{d.pin[1]} {d.pin[0]}}
mb_connect_net -from {a.pin} -through_in_pin{\b.c .d.pin}\
             -through_out_pin {d.pin[1]}
```

Appendix B: An Example of Using Waivers

SpyGlass does not produce error messages for ‘stopped’ modules that a user explicitly wants to blackbox and for which no definition is supplied. The following methodology illustrates how the relevant messages can be ‘waived’.

Consider the following design hierarchy and the files containing their descriptions:

```
Test                (test.v)
|
--WRAP_1            (WRAP_1.v)
    |
    --WRAP_2        (no definition is specified)
```

You supply the following ‘stop’ options:

```
set_option stop WRAP_2
set_option stop XYZ
set_option stop ABC
set_option stop PQR
```

In addition, you have not supplied any definition of these ‘stop’ modules. You will see one error message:

```
ErrorAnalyzeBBox    ErrorAnalyzeBBox    Error       WRAP_1.v
 5       10    Design Unit 'WRAP_2' has no definition; black-
```

```
box behavior assumed and module interface inferred
```

This message can be waived by following these steps:

1. Create a ‘waiver’ file, for example “tt.swl” with the following content:

   ```
   waive -rule "ErrorAnalyzeBBox" -msg "Design
   Unit.*'WR.*'.*has no definition; black-box behavior
   assumed and module interface inferred" –regexp
   ```

2. Add the following line in the Tcl file:

   ```
   read_file –type waiver tt.swl
   ```

The following will appear in stdout:

```
Total Number of Generated Messages :  8 (1 error, 1 warning,
6 Infos)
  Number of Waived Messages        :  1 (1 error, 0 warning,
0 Info)
  Number of Reported Messages      :  7 (0 error, 1 warning,
6 Infos)
```

Appendix C: Working with Generated Names

Naming Conventions Used for the Generate Blocks

Memories are often instantiated inside a ‘generate’ statement following a Verilog-2001 or SystemVerilog language. Consider the following RTL:

```
genvar j;
generate
  for (j = 0; j < 8; j = j + 1) begin : mylabel
    LV1P4096W16B16C inst (.CK(ck), .CE(_ce[j]), .WE(we),
                          .A(a[12-1:0]), .I(i),
                          .O(_o[16*(j+1)-1:16*j]),
                          .BWE(bwe));
  end
endgenerate
```

In this example, the name of the memory cell is `LV1P4096W16B16C`. The names of the created instances are:

```
"\mylabel[7].inst "
"\mylabel[6].inst "
```

```
"\mylabel[5].inst "
"\mylabel[4].inst "
"\mylabel[3].inst "
"\mylabel[2].inst "
"\mylabel[1].inst "
"\mylabel[0].inst "
```

The Tcl command ‘mb_report_instances’ can be used to extract these names.

The instance names are created with Verilog ‘escape’ character ‘\’. This requires a trailing space character in the part of the name in order to qualify as a legal Verilog name.

The following are some examples of the recommended usage of commands involving escaped characters:

```
mb_define_instance -instance {top.\mylabel[0].inst }
    -class 1PMEM
mb_define_instance -instance {top.\mylabel[1].inst }
    -class 1PMEM
mb_define_instance -instance collar_0 -class COLLAR_2P \
    -children { \
          {top.\mylabel[0].inst } \
          {top.\mylabel[1].inst } \
    }


mb_define_instance -instance {b1_2} -class BIST \
     -children { \
           {{a0.\L1[1].inst } -view v2} \
            {{a0.\L1[2].inst } -view v3} \
    }
```

Multi-level generate statements can be used to instantiate memories:

```
module ram(ck, ce, we, a, i, o, bwe);

  :

  genvar j;
```

```
  generate
    for (j = 0; j < 2; j = j + 1) begin : label1
           LVRAM inst_lvram (.CK(ck), .CE(_ce[j]), .WE(we),
                             .A(a[12-1:0]), .I(i),
                             .O(_o[16*(j+1)-1:16*j]),
                             .BWE(bwe));
    end
  endgenerate
endmodule

module LVRAM (CK, CE, WE, A, I, O, BWE);

 :

  genvar j;
  generate
    for (j = 0; j < 3; j = j + 1) begin : label2

           LV1P4096B16W16 inst (.CK(CK), .CE(CE), .WE(WE),
                             .A(A), .I(I),
                             .O(_o[16*(j+1)-1:16*j]),
                             .BWE(BWE));
    end
  endgenerate
endmodule
```

The following are the instances of the cell ‘LV1P4096B16W16’:

```
"\label1[1].inst_lvram .\label2[2].inst "
"\label1[1].inst_lvram .\label2[1].inst "
"\label1[1].inst_lvram .\label2[0].inst "
"\label1[0].inst_lvram .\label2[2].inst "
"\label1[0].inst_lvram .\label2[1].inst "
"\label1[0].inst_lvram .\label2[0].inst "
```

A Complete Example

```
// File: top.v
module top (func);
  input func;

// ------------------------------------------------------------
// Three memories are instantiated inside this 'ram' block.
// They produce names with escape characters
// ------------------------------------------------------------
ram inst_ram(.func(func));

// ------------------------------------------------------------
// Two more memories are instantiated at the top level.
// They produce 'simple' names
// ------------------------------------------------------------
MEM inst1 (.func(func), .BEN(1'b0));
MEM inst2 (.func(func), .BEN(1'b0));
endmodule
// File: ram.v
module ram(func);
  input func;
// ------------------------------------------------------------
// Three (3) memories are instantiated through this
'generate' block.
// Instance names are generated with 'escape' ('\') character
// ------------------------------------------------------------
  genvar j;
  generate
    for (j = 0; j < 3; j = j + 1) begin : L1
      MEM inst (.func(func), .BEN(1'b0));
    end
  endgenerate

endmodule

// File: lib.v
// Contains the description of memories and the bist IPs
```

```
module MEM (func, BEN);
input func;
input BEN;


endmodule

module bist(BSTART);
output BSTART;
endmodule

# File: vendor.data
mb_define_class -class WRAP
mb_define_class -class BIST -children WRAP

mb_configure_connect_net -from_class BIST -from_pin BSTART \
                      -to_class WRAP -to_pin BEN –broadcast

# File: top.tcl
# This is the user data to drive MBIST insertion in the
design
set design top
new_project prj_$design -force

read_file -type hdl $design.v
read_file -type hdl ram.v
set_option v lib.v

set_option top $design
current_methodology $::SPYGLASS_HOME/Methodology/MBIST_DFT
current_goal mbist_dft -alltop
set_parameter mbist_top_level_name $design
set_parameter mbist_net_prefix w_

mb_reset
source vendor.data

# ------------------------------------------------------------
```

```
# The following three instances refer to the memory instances
# instantiated through a 'generate' block
# ------------------------------------------------------------
mb_define_instance -instance {inst_ram.\L1[2].inst } -class
WRAP
mb_define_instance -instance {inst_ram.\L1[1].inst } -class
WRAP
mb_define_instance -instance {inst_ram.\L1[0].inst } -class
WRAP
# ------------------------------------------------------------
# The following two instances are instantiated outside of the
# 'generate' block. Hence they have 'simple' names.
# ------------------------------------------------------------
mb_define_instance -instance {inst1} -class WRAP
mb_define_instance -instance {inst2} -class WRAP

mb_insert_instance -instance inst_bist -cell bist
mb_define_instance -instance inst_bist -class BIST \
                -children { \
                            {inst1} {inst2} \
                            {inst_ram.\L1[2].inst } \
                            {inst_ram.\L1[1].inst } \
                            {inst_ram.\L1[0].inst } \
                          }

mb_insert -out $design.sgdc
save_project
close_project
exit
```

Run SpyGlass DFT MBIST product as follows:

```
sg_shell –tcl top.tcl
```

Appendix D: An Example of Shell View Created during Bottom-up Flow

Design:

```
// File: rtl/A.v
module A(func);
input func;
memwrap inst_M1(.func(func), .bstart(1'b0));
memwrap inst_M2(.func(func), .bstart(1'b0));
P inst_P (.func(func));
endmodule

// File: rtl/P.v
module P(func);
input func;
endmodule

// File: rtl/top.v
module top(func);
input func;
A inst_A(.func(func));
memwrap inst_M(.func(func), .bstart(1'b0));
endmodule
```

Vendor data:

```
mb_define_class -class WRAP
mb_define_class -class BIST -children WRAP
mb_configure_connect_net -from_class BIST -from_pin bstart \
        -to_class WRAP -to_pin bstart -bitwise_broadcast
```

User data:

```
# File: A.tcl
set design A
new_project prj_$design -force

set_option y {./rtl}
set_option libext { .v }
read_file -type hdl rtl/$design.v lib/lib.v

current_methodology $::SPYGLASS_HOME/Methodology/MBIST_DFT
current_goal mbist_dft -alltop
set_parameter mbist_top_level_name $design
set_parameter mbist_net_prefix w_

mb_reset
source lib/vendor.file

mb_define_instance -instance inst_M1 -class WRAP
mb_define_instance -instance inst_M2 -class WRAP

mb_insert_instance -instance inst_bist -cell bist
mb_define_instance -instance inst_bist -class BIST -children
{inst_M1 inst_M2}

mb_insert -out $design.sgdc
save_project
close_project
exit

// File: top.tcl
set design top
```

```
new_project prj_$design -force

set_option y {./rtl}
set_option libext {.v}
set_option stop { A }
read_file -type hdl rtl/$design.v lib/lib.v
# ------------------------------------------------
# Read the ‘shell’ view for the lower block (‘A’)
# ------------------------------------------------
read_file -type sourcelist prj_A/mbist_dft/spyglass_reports/
rme/mbist-dft/atrenta_modified_rtl_files/
atrenta_generated_abstract_block.f

current_methodology $::SPYGLASS_HOME/Methodology/MBIST_DFT
current_goal mbist_dft -alltop
set_parameter mbist_top_level_name $design
set_parameter mbist_net_prefix w_

mb_reset
source lib/vendor.file

mb_define_instance -instance inst_M -class WRAP

mb_insert_instance -instance inst_bist -cell bist
mb_define_instance -instance inst_bist -class BIST -children
{inst_M }

mb_insert -out $design.sgdc

save_project
close_project
exit
```

Run the SpyGlass DFT MBIST product as follows:

Step 1:

```
sg_shell –tcl A.tcl
```

Step 2:

```
sg_shell –tcl top.tcl
```