Tessent™ MemoryBIST User's Manual
For Use With Tessent Shell

SIEMENS

About Siemens Digital Industries Software

Siemens Digital Industries Software is a leading global provider of product life cycle management (PLM) software and services with 7 million licensed seats and 71,000 customers worldwide. Headquartered in Plano, Texas, Siemens Digital Industries Software works collaboratively with companies to deliver open solutions that help them turn more ideas into successful products. For more information on Siemens Digital Industries Software products and services, visit www.siemens.com/plm.

Support Center: support.sw.siemens.com
Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

Revision History ISO-26262

Revision	Changes	Status/ Date
27	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Dec 2022
26	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Sep 2022
25	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Jun 2022
24	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Mar 2022

Author: In-house procedures and working practices require multiple authors for documents. All associated authors for each topic within this document are tracked within the Siemens documentation source. For specific topic authors, contact the Siemens Digital Industries Software documentation department.

Revision History: Released documents include a revision history of up to four revisions. For earlier revision history, refer to earlier releases of documentation on Support Center.

Table of Contents

List of Figures

List of Tables

Chapter 1
Tessent MemoryBIST Overview

This manual introduces Tessent MemoryBIST and explains how to use Tessent MemoryBIST in the Tessent Shell based integration flow. This manual also provides complete reference information for all Tessent MemoryBIST-specific syntax used within the Tessent Shell environment.

Why Consider Tessent MemoryBIST for Your Chip?

Tessent MemoryBIST is a highly advanced product that addresses several limitations of existing memory BIST solutions:

- Tessent MemoryBIST combines the power of memory BIST with tester programmability. When you use Tessent MemoryBIST, you include an embedded tester on the chip that enables you to test and diagnose the embedded memories with algorithms that were not considered while designing the chip—the post-silicon programmability.

- Tessent MemoryBIST also provides the ability to use different types of repairable memories in one controller. You can use repair analysis to implement a self-repair solution for repairable memories to significantly improve the chip yield without sacrificing a lot of silicon area overhead.

- Tessent MemoryBIST also provides significant flexibility and automation in designing an optimal memory BIST configuration to meet your chip power and test-time budget.

Tessent MemoryBIST Capabilities

Using Tessent MemoryBIST, you can perform the following:

- Test multiple memories using one memory BIST controller that has one or more BIST steps, where BIST steps are run in sequence.

- Test memories in parallel in one BIST step or in sequence in several BIST steps.

- Choose memory test algorithms from the Siemens EDA library of algorithms to be hard-coded into the memory BIST controller.

- Define one or more custom test memory algorithms to be hard-coded into the memory BIST controller.

- Run the memory BIST controller for all steps with the specific algorithms assigned at generation time (default configuration).

- Run the memory BIST controller in diagnostic mode where you can freeze on a specific BIST step, specific memory test port, or specific error count.

- Select a hard-coded memory BIST algorithm to be applied to a specific memory BIST step at the tester.

- Select an algorithm from a library of algorithms to be applied to a specific memory BIST step.

- Define a custom algorithm at tester time to be applied to a specific memory BIST step.

- Perform repair analysis on memories implementing different redundancy schemes such as row only, column only, or row and column.

Tessent MemoryBIST Library Algorithms

You can select algorithms for testing your memories from a library of built-in algorithms.

Below is a list of available library algorithms:

- SMarch

- SMarchCHKB

- SMarchCHKBci

- SMarchCHKBcil

- SMarchCHKBvcd

- ReadOnly

- LVMarchX

- LVMarchY

- LVMarchCMinus

- LVMarchLA

- LVRowBar

- LVColumnBar

- LVGalPat

- LVGalColumn

- LVGalRow

- LVCheckerboard1X1

- LVCheckerboard4X4

- LVWalkingPat

- LVBitSurroundDisturb

The descriptions for the library algorithms listed above are available in the *lib/tools/etv/ membistipg_algos* folder of the tool tree. For details on each algorithm, refer to the "MemoryBIST Algorithms" section. Most of the LV* library algorithms have updated versions that have been optimized to eliminate redundant operations. Their descriptions are stored in the Tessent installation directory, along with a large variety of algorithms from the literature, such as the hammer read and write tests. These algorithms are available in the *lib/technology/ memory_bist/algo* folder of the tool tree, and can be encoded into the BIST engine as custom algorithms by supplying the descriptions to the tool.

User-Defined Algorithms

Often it is required to design specific test algorithms to target specific memory defects that are difficult to detect with the existing algorithms. To effectively test the memory, you might need to apply multiple algorithms to the same memory in a specific sequence. In some cases, you might choose to apply several algorithms to diagnose memory defects that are otherwise too difficult to identify.

For details on implementing user-defined algorithms, refer to the "Using Tessent User-Defined Algorithms" section.

Tessent MemoryBIST Library Operation Sets

You can select from a library of operation sets for generating waveforms to drive the memory. The list below shows the standard operation sets available for each type of memory BIST controller. The names of all operation sets are reserved names.

- Async — This operation set applies to clockless RAMs. The operation set consists of three operations: Write, Read and ReadModifyWrite. Each operation is performed in four cycles (or ticks) of the reference clock used to control the memory BIST controller. Basic shadow reads used to detect some multi-port specific faults are added to the Read and ReadModifyWrite operations. This operation set is rarely used for modern memories.

- AsyncWR — This operation set is a variation of Async. The same operations (Write, Read and ReadModifyWrite) are performed in six cycles instead of four. Shadow reads are added to the Write operation.

- ROM — This operation set applies to ROMs and is typically used in combination with the ReadOnly library algorithm. It only includes two operations, Read and CompareMISR, each run in two cycles. The operation set can be used for ROMs with or without a clock input.

- Sync — This operation set applies to synchronous RAMs, that is memories that have a port with function clock. The three operations, Write, Read and ReadModifyWrite, are performed in two cycles each. Basic shadow reads, which are used to detect some multi-port specific faults, are added to the Read and ReadModifyWrite operations. This operation set is compatible with most library algorithms.

- SyncWR — This operation set is a variation of Sync. The only difference is that the Write operation is modified to include shadow reads to the Write operation.

- SyncWRvcd — This operation set is an extension of SyncWR and is typically used in combination with the SMarchCHKBvcd library algorithm. The operation set includes 12 operations of two to three cycles that are listed in Table C-10 of the SMarchCHKBvcd Algorithm description. The operations allow for more complete testing of memory control signals, such as bit/byte write enables, read enable, and select. It also includes operations performing concurrent reads and writes that improve the coverage of multi-port specific faults.

- TessentSyncRamOps — This operation set is an extension of the SyncWRvcd operation set. It includes 25 operations, of two to three cycles each, that allow implementation of all library algorithms as well as a large variety of algorithms from the literature available in the *lib/technology/memory_bist/algo* directory.

    Additionally, several operations are available to optimize algorithms by reducing test time and the number of instructions required to implement them. This is important when using soft programmable controllers. These operations support fast address changes, switching of the address register in the middle of an operation, and other useful features. Refer to the "Optimizing Custom Algorithms and Operation Sets" section for more information on the subject

- TessentSyncRamOpsHR4 — This operation set is a variation of the TessentSyncRamOps operation set. The only difference is the implementation of the Read operation, which consists of four consecutive reads instead of two. It is a simple way of integrating hammer read tests into existing library or custom algorithms. Note

that even if the memory cell is read multiple times, the read result is only strobed (compared) once on the first read. Using the notation describing the MemoryBIST algorithms, the normal sequence describing the Read operation, (R0Rx), becomes (R0RxRxRx). Similarly, (R1Rx), becomes (R1RxRxRx). This means that a fault caused by the additional reads are detected in a subsequent phase of the algorithm. More information can be found in Knowledge Base Article MG604200 "*Implementing Hammer tests for planar and FinFET memories with Tessent MemoryBIST*", available from Support Center.

- TessentSyncRamOpsHR6 — This operation set is similar to TessentSyncRamOpsHR4. The only difference is that the Read operation has six consecutive reads instead of four. The sequence for reading a 0 or a 1 becomes (R0RxRxRxRxRx) or (R1RxRxRxRxRx) respectively.

The library operation sets, especially TessentSyncRamOps, constitute a good starting point to create your own operation sets. The library operation sets are generic and not optimized for one type of memory. Custom operation sets might be required to accommodate specific timing requirements or modes of operation that can be controlled with UserN signals, where N varies from 0 to 23. The source of all library operation sets can be found in the *lib/technology/ memorybist/opset* folder of the tool tree.

Custom Operation Sets

New memories can have multiple modes that require different waveforms to perform different operations on the memory. Tessent MemoryBIST enables you to define new operation sets at generation time to address these requirements. Combined with the algorithm programming capability, you have a very powerful combination to test and diagnose the memories in different operation modes under different access conditions.

The standard Tessent MemoryBIST library operation sets assume memories with synchronous write and read ports (Sync* OperationSets), or asynchronous write and read ports (Async* OperationSets). Memories with synchronous write and asynchronous read ports need a custom operation set.

Memories with asynchronous read ports output read data in the same cycle as the active read enable. This requires an OperationSet with the StrobeDataOut in the first Tick of a Read operation. The custom OperationSet shown in Figure 1-1 can be used for memories with synchronous write and asynchronous read ports.

Figure 1-1. Synchronous Write, Asynchronous Read Custom OperationSet

```
OperationSet (SyncW_AsyncR) {
  Operation (Write) {
    Tick {
          WriteEnable        : On;
          ShadowReadEnable   : On;
          ShadowReadAddress  : On;
    }
    Tick {
          WriteEnable        : Off;
          ReadEnable         : On;
          ShadowReadAddress  : Off;
    }
  }
  Operation (Read) {
    Tick {
          StrobeDataOut;
          ReadEnable         : On;
          ShadowReadEnable   : On;
    }
    Tick {
          ReadEnable         : On;
          ShadowReadEnable   : On;
    }
  }

  Operation (ReadModifyWrite) {
    Tick {
          StrobeDataOut;
          ReadEnable         : On;
          ShadowReadEnable   : On;
    }
    Tick {
          WriteEnable        : On;
          ReadEnable         : Off;
          ShadowReadAddress  : On;
    }
  }
}
```

For details on implementing a custom OperationSet, refer to the "Using Tessent User-Defined Algorithms" section and the "OperationSet" wrapper description.

Chapter 2
Getting Started

This chapter describes how to start inserting Tessent MemoryBIST within Tessent Shell and includes examples showing the most common scenarios and usages. If you are inserting memory BIST in a design with repairable memories, some of the flow steps have variations that are also described.

For a complete set of wrapper and property descriptions, refer to the "MemoryBist" and "MemoryBisr" sections of the "DftSpecification Configuration Syntax," "PatternsSpecification Configuration Syntax," and "DefaultsSpecification Configuration Syntax" sections in the *Tessent Shell Reference Manual*. The flow and main steps are the same for inserting memory BIST at the sub-block, physical block, or chip level.

DFT Flow Using Tessent Shell

Tessent MemoryBIST in Tessent Shell has a high-level flow sequence.

Figure 2-1 illustrates the high-level sequence of steps required to insert memory BIST into a design. Each step in the figure links to more detailed information about the design-for-test flow, including examples.

Figure 2-1. Design Flow for Tessent Shell MemoryBIST

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.
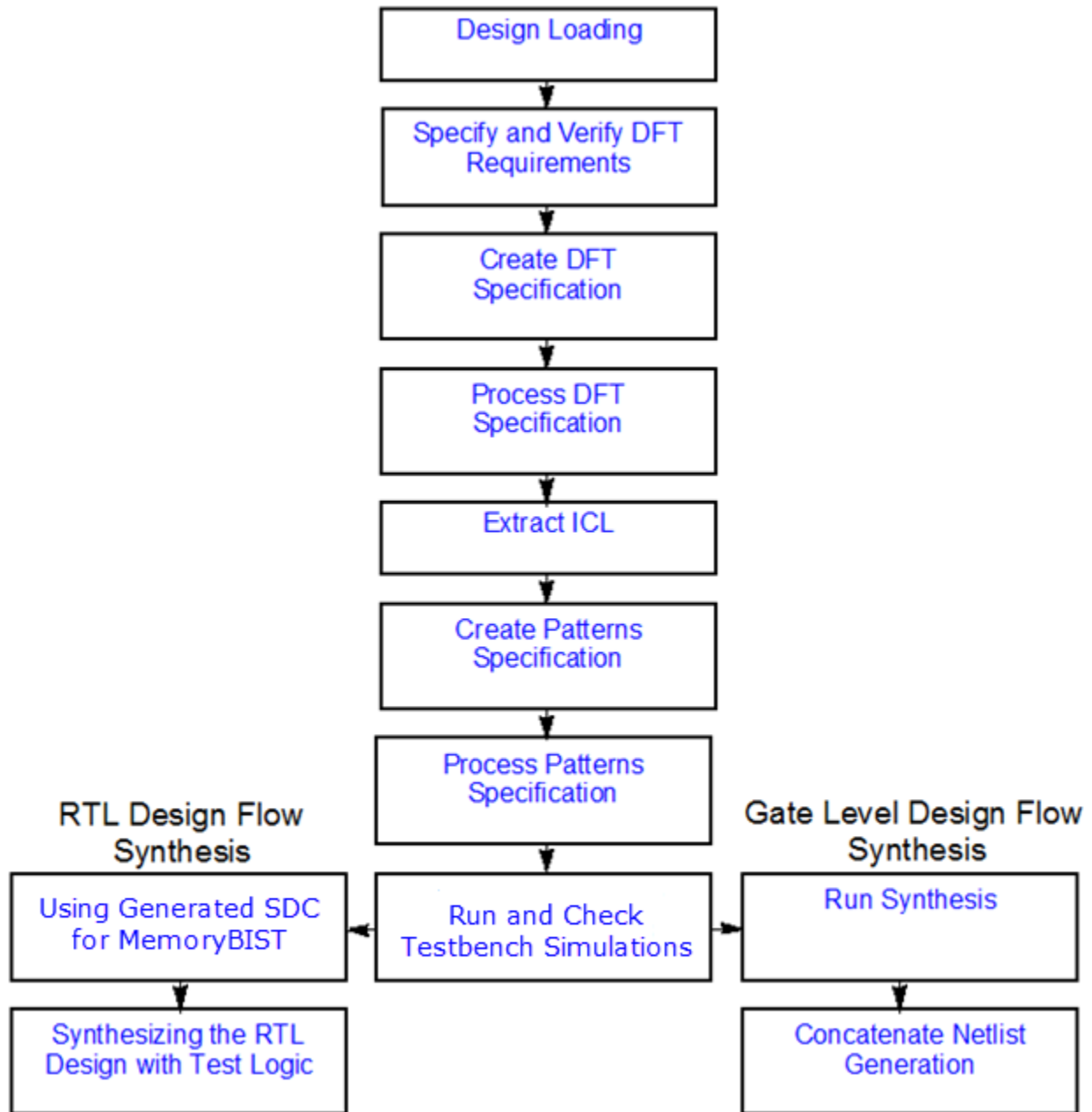
Prerequisites

To insert memory BIST into your design, you must have either an RTL or a gate-level netlist as well as memory BIST libraries.

If the design contains standard cells, the Tessent cell library or ATPG library is also required. For other IP blocks that do not have either RTL or library models, the simulation model can be loaded using the read_verilog or read_vhdl command with the -interface_only option. This instructs the tool to ignore the internals of all modules specified in the filename argument and extract only the module port definitions and parameters.

Design Flow Dofile Example

A sample dofile in this section shows you how to set up a design flow.

The following example dofile sets up the design flow described in Figure 2-1.

Design Loading

```
set_context dft -rtl
read_cell_library ../library/adk.tcelllib
set_design_sources -format verilog -y {../library/mem ../design/rtl} \
   -extension v
set_design_sources -format tcd_memory -y ../library/mem -extension lib
read_verilog ../design/rtl/blockA.v
set_current_design blockA
```

Specify and Verify DFT Requirements

```
set_design_level physical_block
set_dft_specification_requirements -memory_test on
add_clocks CLK -period 12ns -label clka
check_design_rules
```

Create DFT Specification

```
set spec [create_dft_specification]
report_config_data $spec
```

Process DFT Specification

```
process_dft_specification
```

Extract ICL

```
extract_icl
```

Create Patterns Specification

```
create_patterns_specification
```

Process Patterns Specification

process_patterns_specification

Run and Check Testbench Simulations

run_testbench_simulations
check_testbench_simulations

Test Logic Synthesis

run_synthesis

Design Loading

Design Loading is the first step in Tessent MemoryBIST insertion using Tessent Shell. The step consists of setting the correct context, reading libraries, reading the design, elaborating the design, and loading and reporting the design data.

Figure 2-2. Design Loading
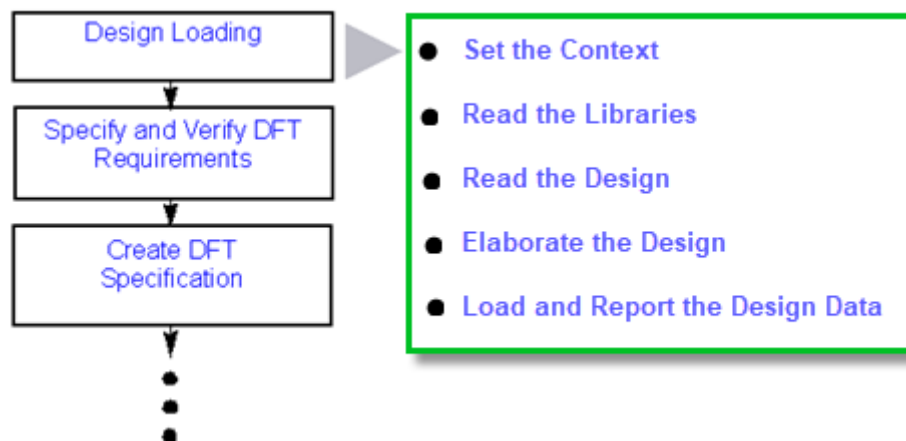
Set the Context

In Tessent Shell, setting the context means two things. First and foremost, you must set the context to dft for memory BIST hardware to be created. Second, you must specify whether the design type to be read in is written in RTL. If so, you must specify -rtl. If the design to be read in is a gate-level Verilog netlist, you should specify -no_rtl. When using the -no_rtl mode, a concatenated netlist is written out at the end of the dft insertion phase. In rtl mode, the file structure of the input design is preserved and only the modified design files are written out at the end of the dft insertion phase along with the newly created test IP. The netlist to be read in can be Verilog, VHDL, or mixed language.

If memory BIST has been inserted into a block that is now being integrated at a higher hierarchy level of the design, you must open the Tessent Shell Data Base (TSDB) of the child block (the memory BIST inserted sub_block or physical_block) using the open_tsdb command. If you are using the same TSDB for both child and parent, you can reuse the TSDB (the default is tsdb_outdir), and you do not need to explicitly open the default TSDB because the existing content of the TSDB output directory is automatically visible to the tool. See the

set_tsdb_output_directory command description for how to control the name and location of the TSDB output directory.

Example 1

The following example sets the context to dft and specifies that the design to be read in is written in RTL.

```
set_context dft -rtl
```

Example 2

The following example sets the context to dft and specifies that a gate-level netlist is read in.

```
set_context dft -no_rtl
```

Example 3

The following example opens a child's TSDB directory and therefore, exposes it at the parent level.

```
open_tsdb ../core_tsdb_outdir
```

Read the Libraries

You can use the read_cell_library command to read in the library file for library cells that are instantiated in the design. The MemoryBIST library suffices for RTL-only designs. However, typical designs also contain library cells instantiated in the RTL for pad cells and clock control, requiring additional libraries. When the Tessent cell libraries do not include the pad information, the LV pad library is natively supported by the read_cell_library command and can be used to augment the Tessent cell libraries with the pad information.

Memory BIST libraries are considered as Tessent core descriptions (TCDs) in Tessent Shell and are loaded and referred to as TCDs. The memory BIST models for the LV flow are compatible with Tessent Shell without modification and are loaded and referred to in the flow as TCDs.

Examples

Example 1

The following example shows how to read in the Tessent cell library file for the pad IO macros.

```
read_cell_library ../library/adk_complete.tcelllib
```

Example 2

The following example shows how to explicitly read in memory BIST models.

```
read_core_descriptions ../library/128x64_RAM.memlib
```

Example 3

The following example shows how to use set_design_sources to reference memory BIST libraries. Using the tcd_memory type for the -format option indicates to the tool to expect a memory library file to be read in.

```
set_design_sources -format tcd_memory -y ../library/memlibs -extensions \
    memlib
```

Flow Variation for Repairable Memories

The memory library files used as input to Tessent MemoryBIST must contain specific wrappers to enable the generation and insertion of repair logic.

The RedundancyAnalysis wrapper is necessary to enable the generation of the BIRA logic that calculates the repair solution for a memory. The PinMap sub-wrapper inside the RedundancyAnalysis wrapper is necessary to enable the generation of the BISR logic that is used to transfer the repair solution to the memory.

Typically, memory BIST library files are provided and certified by a memory IP provider and can be used without further modification.

Read the Design

In Tessent Shell, after setting the context and loading the required libraries, you can use the read_verilog command to read in the design.

Examples

Example 1

The following example shows how to read in one netlist that can be either RTL or gate level.

```
read_verilog ../netlist/cpu_top.v
```

Example 2

The following example shows how to specify a file and a directory library in which to search for Verilog modules.

```
set_design_sources -format verilog -v ../design/top.v \
    -y ../design -extensions v gv
```

> **Note**
> set_design_sources should only be set once for each format type. If multiple set_design_sources are used for a given format, the last set_design_sources for the given file type is used. The -v and -y options accept a list of paths, if needed.

Elaborate the Design

The next step in design loading is to elaborate the design using the set_current_design command.

The set_current_design command specifies the root of the design that all subsequent operations are acted on. If any module descriptions are missing, design elaboration identifies them. For memory BIST insertion, modules are allowed to have no definition if they do not contain memories or do not contain clock tree elements in the fan-in of the memories. You can specify them with the add_black_box -module command.

Example

The following example shows how to use the set_current_design command.

```
set_current_design blockA
```

Load and Report the Design Data

You can load memory placement and power domain data, and use several commands to check the loaded design and associated libraries.

You can load memory placement (DEF, or Design Exchange Format files) and power domain data (UPF, or Unified Power Format and CPF, or Common Power Format files) in this step. The data from these files affects the memory BIST planning in terms of memory partition and memory BIST controller allocation.

The following section provides examples of the more commonly used reporting commands.

Examples

Example 1

The following example runs set_design_sources and then reports the designs sources that are referenced.

```
> set_design_sources -format verilog -y {../data/design/mem \
    ../data/design/rtl} -extension v
> report_design_sources
//  ICL search_design_load_path: activated
//  BoundaryScan search_design_load_path: activated
//  Scan search_design_load_path: activated
//  ----------------------------------------------------
//  format   type  path                   file extensions
//  -------  ----  --------------------   ---------------
//  Verilog  dir   '../data/design/mem'   'v v.gz'
//  Verilog  dir   '../data/design/rtl'   'v v.gz'
```

Example 2

The following example reports the memory models as well as other test core descriptions currently loaded in Tessent Shell. Using the -levels 1 option displays only the core name and type. When this option is not set, the entire model content displays.

```
> report_config_data -partition tcd -levels 1
Core(SYNC_1R1W_16x8) {
  Memory {
    // Not shown
  }
}
Core(SYNC_1RW_32x16) {
  Memory {
    // Not shown
  }
}
Core(SYNC_2R2W_12x8) {
  Memory {
    // Not shown
  }
}
```

_____ **Note** _____

report_config_data displays all core descriptions currently loaded, in addition to memory models.

Example 3

The following example reports the memory instances found during design elaboration.

```
> report_memory_instances
//
// Memory Instance: blockA_l1_i1/blockA_l2_i1/mem1
// ----------------------------------------------
// bist_data_in_pipelining              : off
// physical_cluster_override            :
// power_domain_island                  :
// test_clock_override                  :
// use_in_memory_bist_dft_specification : auto
// use_in_memory_bisr_dft_specification : auto
//
// Memory Instance: blockA_l1_i1/blockA_l2_i1/mem2
// ----------------------------------------------
// bist_data_in_pipelining              : off
// physical_cluster_override            :
// power_domain_island                  :
// test_clock_override                  :
// use_in_memory_bist_dft_specification : auto
// use_in_memory_bisr_dft_specification : auto
```

> **Note**
> This report command provides additional property information as well the memory instance path. These properties can be set indirectly by loading additional design data such as a DEF file to designate a physical cluster. Most properties can be changed explicitly by using the set_memory_instance_options command.

Example 4

This example shows how to read in a DEF file containing the placement information of the memories and a UPF file describing the power domain associated with the memories.

```
read_upf ../data/design/power/blockA.upf
read_def ../data/design/layout/blockA.def
```

Specify and Verify DFT Requirements

The next steps to insert MemoryBIST in Tessent Shell are specifying the DFT requirements, adding constraints, and verifying whether the DFT requirements specified are correct by running Design Rule Checking (DRC).

Figure 2-3. Specify and Verify DFT Requirements
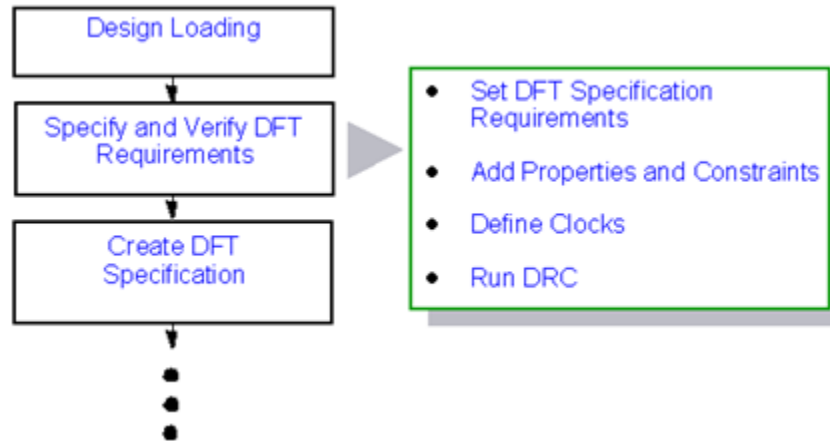
Set DFT Specification Requirements

To insert MemoryBIST, you must specify the DFT specification requirements with the set_dft_specification_requirements command. This enables the DRC specific to memory BIST and instructs the create_dft_specification command to include the MemoryBist wrapper.

Example

The following example shows how the memory BIST DFT specification requirements are specified and how the design level is defined as the chip level.

```
set_dft_specification_requirements -memory_test on
set_design_level chip
```

Flow Variation for Repairable Memories

When the design contains repairable memories, you must consider additional options when using the set_dft_specification_requirements command.

When using

```
set_dft_specification_requirements -memory_test on
```

the following additional options associated with repairable memories are set by default:

```
-memory_bisr_chains auto
-memory_bisr_controller auto
```

In most cases, you do not need to change these default values.

When -memory_bisr_chains is set to auto, if the current design instantiates blocks containing repairable memories that are not already connected to a memory BISR register, BISR registers are added. When -memory_bisr_controller is set to auto and set_design_level is set to chip, a BISR controller is added to the design.

If -memory_bisr_chains and -memory_bisr_controller are set to off, BISR chains and a BISR controller are not inserted into the design, effectively disabling BISR for the entire design.

To disable generation of BISR registers for specific memory instances, use the set_memory_instance_options command with the -use_in_memory_bisr_dft_specification option set to off. The following example disables the generation of the BISR register for memory instance mem4 even if the memory has repair resources.

```
SETUP> set_memory_instance_options blockA_clka_i1/mem4 \
   -use_in_memory_bisr_dft_specification off
```

Add Properties and Constraints

You can add or change properties and constraints that affect DFT specification creation, as shown in the following examples.

Examples

Example 1

To insert memory BIST at the chip level, four TAP pins (TDI, TCK, TMS, and TDO) must be available at the chip level and connected to pad IO macros. TRST, which is optional, can be an output pin of a power-up detector. If the TAP pins are already named tck, tdi, tms tdo, and trst, the function is automatically set.

The following example shows how to specify the five TAP pins (tck_p, tdi_p, tms_p, trst_p, tdo_p).

```
set_attribute_value tck_p    -name function -value tck
set_attribute_value tdi_p    -name function -value tdi
set_attribute_value tms_p    -name function -value tms
set_attribute_value trst_p   -name function -value trst
set_attribute_value tdo_p    -name function -value tdo
```

Example 2

The following example adds additional attributes for connecting essential fuse box connections at the chip level that are not inferred from the fuse box model.

```
set_config_value -in /DftSpecification(top,rtl)MemoryBisr/Controller/\
  repair_clock_connection clkb
set_config_value -in /DftSpecification(top,rtl)MemoryBisr/Controller/\
  programming_voltage_source vddq
set_config_value -in /DftSpecification(top,rtl)MemoryBisr/Controller/\
  repair_trigger_connection bisr_rstn
```

Example 3

You can use the DefaultsSpecification wrapper to change many of the built-in default values to match your preferences and requirements. The defaults setting mechanism also comes with a hierarchy of DefaultsSpecification wrappers that enable you to have company, group, and user-level defaults.

The following example shows how you can create a company memory BIST DefaultsSpecification.

Generate a file with all of the default options for memory BIST:

```
report_config_syntax DefaultsSpecification/DftSpecification/MemoryBist \
> company.tessent_defaults
```

The following is a snippet from the company.tessent_defaults file:

```
DefaultsSpecification(<policy>) { // legal: company group user
  DftSpecification {
    MemoryBist {
      clock_partitioning                 : <string>;// legal :
                                                    // (per_clock_domain)
                                                    // per_sync_clock_group

      max_steps_per_controller      : <int>;   // default: unlimited
      max_memories_per_step         : <int>;   // default: unlimited
      max_test_time_per_controller  : <time>;  // default: 500ms {
                                                // symbols: unlimited }
      max_power_per_step                 : <milli_watts>; // default: 500
                                                    // { symbols:
                                                    // unlimited }
      single_memory_dimension_per_step : <boolean>;     // default: off
      ControllerOptions {
      ...
```

You can use this file to create a company memory BIST DefaultsSpecification as follows:

```
DefaultsSpecification(Company) {
  DftSpecification {
    MemoryBist {
      clock_partitioning              : per_clock_domain;
      max_steps_per_controller        : 5;
      max_memories_per_step           : 10;
      max_test_time_per_controller    : 1000ms;
      max_power_per_step              : 250;
      single_memory_dimension_per_step : Off;
      ControllerOptions {
      ...
```

The created DefaultsSpecification can be read automatically by referencing it in the .tessent_startup file or explicitly using the read_config_data command.

Define Clocks

The next step in Specify and Verify DFT Requirements is to define the clocks with the add_clocks command.

When elaborating the design, Tessent Shell automatically traces clocks from the clock pins of a memory to a clock source.

For a detailed explanation of defining clocks and the associated clock DRC, please refer to the add_clocks command in the *Tessent Shell Reference Manual*.

Examples

Example 1

The following example shows that with a simple clock path (in this case, a path from the primary design clock pin to memories that consists of wires and buffers), you need to only define the clock period and reference the clock source pin. The reason is that the clock path is automatically traced from the memory being tested to the clock source pin. The -label option enables you to specify a symbolic name. If you do not set this option, a label name is generated automatically.

```
add_clocks CLK -period 12ns -label blockA_ram_clk
```

Example 2

The following example defines an internal clock source from a PLL that drives the memories.

```
add_clocks clk_ref -period 10ns -label clk_ref
add_clocks U_PLL/VCO -label clk_100mhz -reference U_PLL/REF \
  -freq_multiplier 4
```

The first add_clocks command defines the clock source that is the reference for the PLL. The second add_clocks command defines the internal clock source that drives the memories with reference to an input pin of the same instance. The DRC traces this reference pin to its source, which is the clk_ref port in this case.

Run DRC

The next step in Specify and Verify DFT Requirements is to run Design Rule Checking (DRC) to make sure all the constraints are correct. Once DRC is clean, Tessent Shell moves from the SETUP to the ANALYSIS prompt.

```
check_design_rules
```

If your design includes clock gating cells in the memory clock path, see the add_dft_clock_enables command description. If your design includes multiplexers that must be controlled, see the add_dft_control_points command description. And finally, to insert multiplexers in the memory clock paths, see the add_dft_clock_mux command description.

If the design setups have any issues such as incorrectly defined clocks, clock DRC rule violations occur and you must address them before you can proceed to the next flow step.

Flow Variation for Repairable Memories

BISR chains are connected according to the contents of a file named *<design_name>.bisr_segment_order* that is specified with the DftSpecification/MemoryBisr/ bisr_segment_order_file property. The file is automatically generated in the current working directory when running check_design_rules and contains a list of memory instances defining the BISR chain order. By default, the list is ordered by power domain islands (if a UPF or CPF file has been loaded) and memory instance names. The order is also affected by the physical memory placement if a DEF file has been loaded.

If the default BISR chain order is not satisfactory, you can affect BISR chain ordering in one of two ways. The first consists of manually modifying the order of the memory instance names in the *<design_name>.bisr_segment_order* file before executing the process_dft_specification command. If you need to re-run the check_design_rules analysis later either during the same Tessent Shell invocation or a subsequent invocation to reuse the modified *<design_name>.bisr_segment_order* file, you must specify the set_dft_specification_requirements -bisr_segment_order_file command option when still in setup mode.

For example, the following command indicates that the file *modA.bisr_segment_order* should be preserved and used to determine the BISR chain order when the next check_design_rules command is run.

```
SETUP> set_dft_specification_requirements -bisr_segment_order_file \
  modA.bisr_segment_order
```

The second way of affecting the BISR chain order consists of reading a DEF file corresponding to the design using the read_def command. The file should be loaded in Tessent Shell during setup mode after the set_current_design command is performed. For example, the following command reads the DEF file corresponding to design modA. The BISR chain ordering is determined based on the memory placement information found in the DEF file.

```
SETUP> read_def modA.def
```

Note
The file name of the bisr_segment_order_file is arbitrary and does not need to be the default name.

Create DFT Specification

The next step is to create a DFT specification.

Use the create_dft_specification command to create a default DFT specification based on the DFT requirements specified in the previous step. You can use the report_config_data command to report the created DFT specification. Several different ways are available to edit or configure the DFT specification to meet the custom requirements.

Figure 2-4. Create DFT Specification
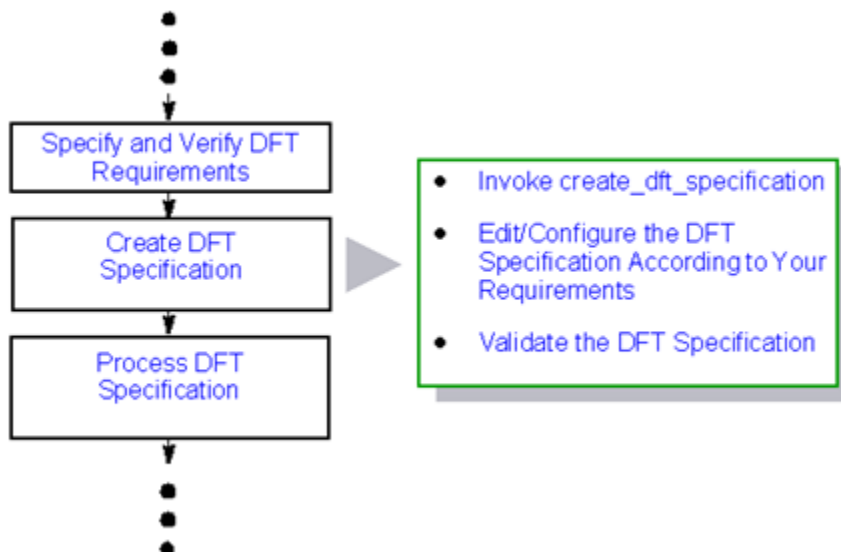
Invoke create_dft_specification

Based on the DFT specification requirements described in the previous step, a DFT specification is automatically created using the create_dft_specification command. This DFT specification is stored in memory.

To report the DFT specification in memory, use the report_config_data command. The DFT specification includes the IjtagNetwork wrapper to specify the access circuitry needed to program the BIST controllers and the MemoryBist wrapper to specify the memory BIST configuration. The IJTAG network is fully compliant with the 1687 IEEE standard.

Example

In the following example, the DFT specification generated with the create_dft_specification is stored in a variable called dft_spec so that the variable can be used to report the DFT specification.

```
set dft_spec [create_dft_specification]
report_config_data $dft_spec
```

Edit/Configure the DFT Specification According to Your Requirements

Use one of the following methods to edit or configure the created DFT specification according to your requirements. You do not need to edit the DFT specification if want to use the default configuration the way it is.

Method 1: Change design constraints and regenerate the DFT specification

In this method, after generating a DFT specification, you return to setup mode and change one of the memory instance options and regenerate the DFT specification. In this case, the physical_cluster_size_ratio is changed from the default of 20% to 40%.

```
set_system_mode setup
set_memory_instance_options -physical_cluster_size_ratio 40
check_design_rules
create_dft_specification -replace
```

Method 2: Use the GUI to edit the DFT specification

In this method, you use the GUI to edit the DFT specification that has been created.

First, open the GUI with the display_specification command. The GUI displays the DFT specification based on the DFT requirements specified in the Specify and Verify DFT Requirements step. The edits you make and apply with the GUI update the DFT specification in memory. If you want the flow to work next time without the GUI edits and be more script-based, you can use the report_config_data command to display the edits you made, and then you can add them to the DFT specification using the read_config_data command.

Figure 2-5. GUI to Edit the DFT Specification
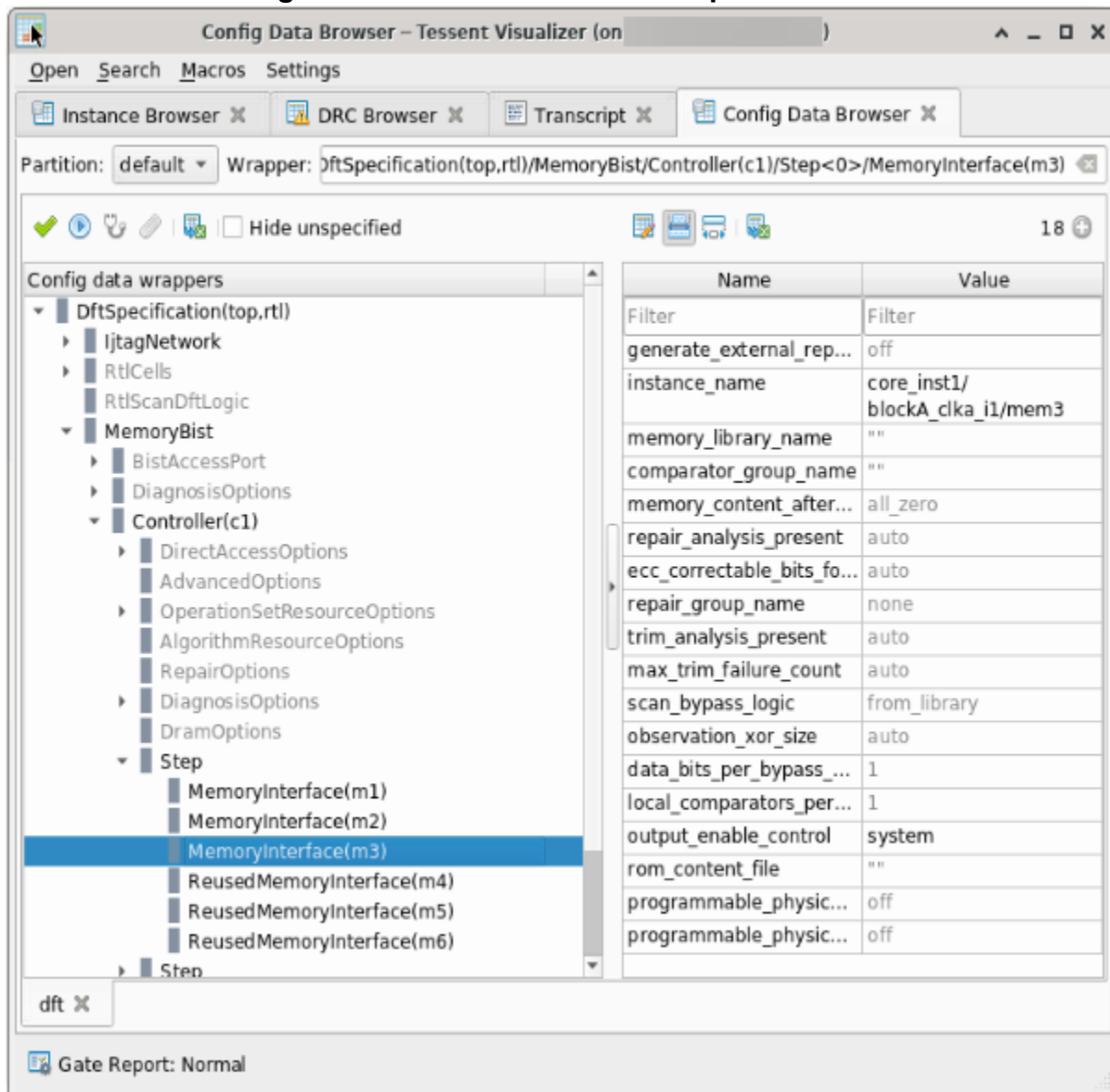
Method 3: Modify the DFT specification in memory

An alternative method is to modify the DFT specification with the add_config_element and set_config_value commands. This way, every time you run the same flow, the edits are called, therefore making your Tcl file or dofile repeatable through iterations, if necessary.

Validate the DFT Specification

In this optional but recommended step, you can validate the edits you made to the DFT specification to make sure no errors exist before you proceed to the next step.

Example

The following example shows how to validate your DFT specification to check for issues or mistakes before processing it.

```
process_dft_specification -validate_only
```

Process DFT Specification

The next step is to process the DFT specification that was created, edited, and validated in the previous step. This step creates and inserts the hardware for all the components that are in the DFT specification.

Figure 2-6. Process DFT Specification
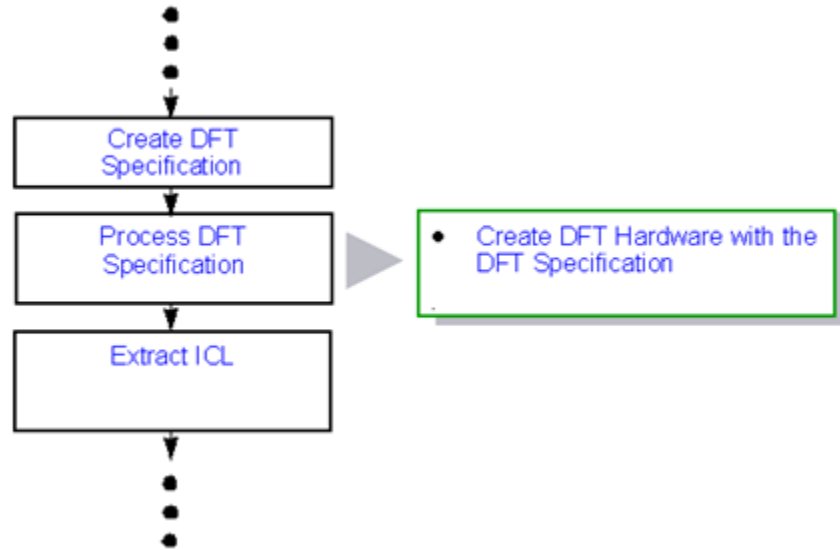
Create DFT Hardware with the DFT Specification

Use the process_dft_specification command to generate and insert into the design all DFT hardware requested with the DFT specification. For Tessent MemoryBIST, when inserting at the chip level, the TAP controller is inserted. When inserting memory BIST at the physical and sub-block levels, an IJTAG host scan interface is inserted.

Example

The following example shows how to generate and insert into the design the hardware requested with the DFT specification.

```
process_dft_specification
```

Extract ICL

The Extract ICL step verifies the proper connectivity of the ICL modules that were inserted with the process_dft_specification command and, when no DRC violations are detected, extracts the top-level ICL description.

The extract_icl command also creates an SDC file that can be used for synthesis. Please refer to the RTL Design Flow Synthesis section for more information.

For a more in-depth explanation of the IJTAG network and associated data such as the ICL used to control the inserted memory BIST, refer to the *Tessent IJTAG User's Manual*.

Tools downstream use this for creating patterns. You can use the open_visualizer command to debug ICL extraction DRC violations.

Figure 2-7. Extract ICL
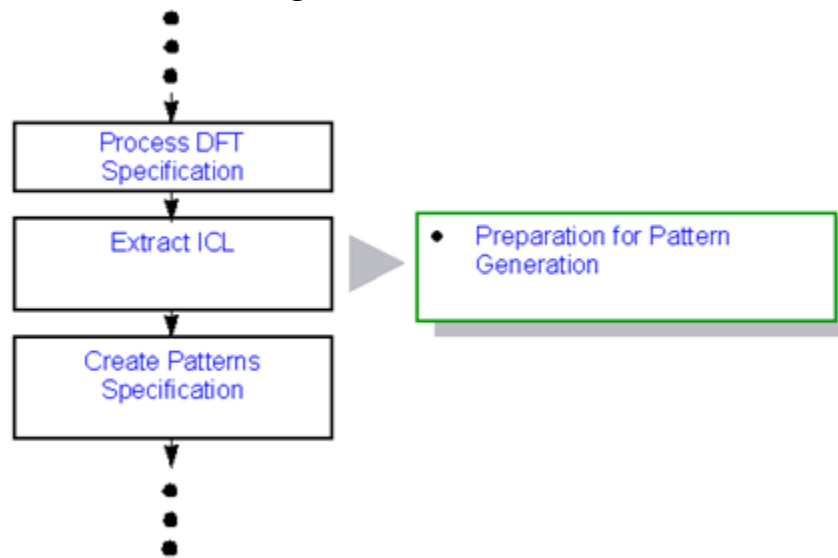
Preparation for Pattern Generation

The extract_icl command prepares the current design for pattern generation by finding all modules (both Tessent instruments and non-Siemens EDA instruments) with their associated ICL modules and, if no DRC violations are detected, creates the ICL for the current design.

The root of the design was specified with the set_current_design command during design elaboration in the Design Loading step. The Create Patterns Specification and Process Patterns Specification steps use the ICL that was created for the root of the design. You can use the open_visualizer command to debug ICL extraction DRC violations. Refer to the "Debugging DRC Violations with Tessent Visualizer" section in the Tessent IJTAG User's Manual.

Example

The following shows how to extract all ICL modules to the root of the design. The example shows the transcript when performing ICL extraction at the current_design level.

```
INSERTION> extract_icl
//  Note: Updating the hierarchical data model to reflect RTL design
changes.
//  Module 'blockA' synthesized (Time = 1.30 sec).
//  Module 'blockB' synthesized (Time = 1.02 sec).
//  Module 'core' synthesized (Time = 1.00 sec).
//  Warning: Rule FN1 violation occurs 534406 times
//  Warning: Rule FP13 violation occurs 108 times
//  Flattening process completed,  design_cells=369  leaf_cells=242
//  library_primitives=115  netlist_primitive=6443  sim_gates=2165  PIs=9
POs=2
//  CPU time=1.07 sec.
//  ----------------------------------------------------------------------
------
//  Begin circuit learning analyses.
//  -------------------------------
//  Learning completed, CPU time=0.08 sec.
//  ----------------------------------------------------------------------
------
//  Begin ICL extraction.
//  --------------------
//  ICL extraction completed, ICL instances=27, CPU time=0.42 sec.
//  ----------------------------------------------------------------------
------
//  Writing ICL file :
//  ./tsdb_outdir/dft_inserted_designs/core_rtl.dft_inserted_design/
core.icl
```

Create Patterns Specification

The Create Patterns Specification step creates the default patterns specification. The patterns specification is a configuration file that tells you what tests are created using process_patterns_specification. You can edit or configure the default patterns specification to generate the patterns specification you want.

Figure 2-8. Create Patterns Specification
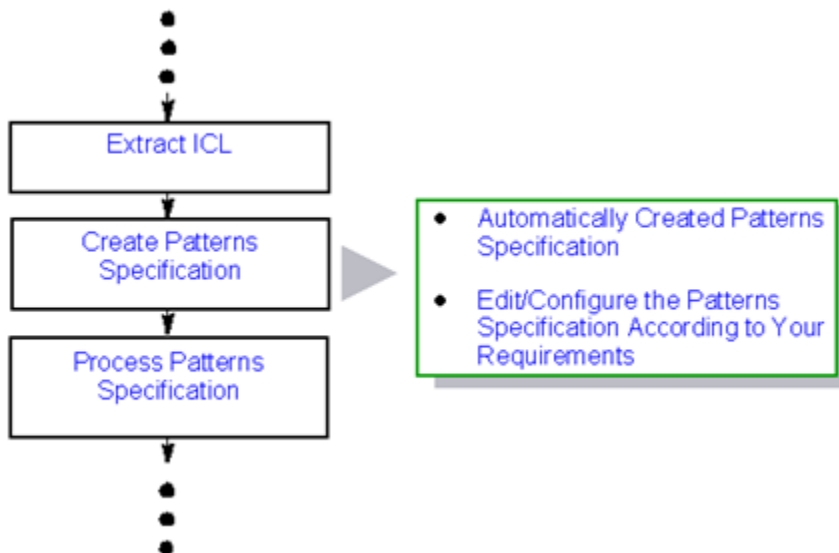
Automatically Created Patterns Specification

The default patterns specification, which is created with the create_patterns_specification command, is only stored in memory.

A copy is written into the patterns directory of the TSDB output directory when running the process_patterns_specification command and the validation generated no errors. To see the specification, use the report_config_data command.

Example

The following example creates the default signoff patterns specification and stores the specification in a variable called pat_spec and then uses this variable to report the patterns specification in memory.

```
set pat_spec [create_patterns_specification]
report_config_data $pat_spec
```

Edit/Configure the Patterns Specification According to Your Requirements

Use one of the following methods to edit or create a patterns specification according to your requirements. Typically, you do not need to edit the default signoff patterns specification. The reason is that it exercises all of the inserted test logic, which is usually the IJTAG network and associated instruments, as well as all of the memory BIST controllers. Only the manufacturing patterns specification may need editing based on your requirements.

Method 1: Edit the patterns specification in memory

This is the preferred method because as you edit the patterns specification in memory, the commands that are used are specified in the Tcl or dofile. This approach enables the edits to be easily repeated for the next iteration by using only scripts.

Example

The following example shows that the default value for tester_period is 100ns and can be changed by editing the patterns specification in memory. For a comprehensive set of configuration values that are changeable, refer to the PatternsSpecification section in the *Tessent Shell Reference Manual*.

```
set pat_spec [create_patterns_specification]
set_config_value -in $pat_spec/Patterns(MemoryBist_P1)/tester_period 50ns
```

Method 2: Write out the patterns specification, edit the file, and read the file back in

Although this method is easier, it is not recommended because every time the primary Tcl script or dofile runs, the patterns specification that is written out overwrites your edits. To make sure your edits are reusable and repeatable using scripts, make a copy of the patterns specification and then edit the specification before reading it back in.

Example

The following example writes the patterns specification into a file called initial_mbist_cfg.pat_spec. After making the edits you want in a copy of this file, this file is read back in. Note that the file that is written out is different from the edited file that is read back in.

```
create_patterns_specification
report_config_data
write_config_data initial_mbist_cfg.pat_spec -wrappers \
  PatternsSpecification(blockA,signoff)
read_config_data bonding1_config_edited.pat_spec
report_config_data
```

Method 3: Use the GUI to edit the Patterns Specification

The third option is to use the add_config_tab PatternsSpecification() command and edit the patterns specification through the GUI.

Example

The following example invokes the patterns specification for blockA.

```
add_config_tab PatternsSpecification(blockA,rtl,signoff)
```

In this example, you use the GUI to edit the patterns specification that has been created. The edits you make and apply with the GUI update the patterns specification in memory. If you want the flow to work next time without the GUI edits and be more script-based, you can use the report_config_data command to display the edits you made, and then you can add them to the DFT specification using the read_config_data command.

Figure 2-9. GUI to Edit the Patterns Specification
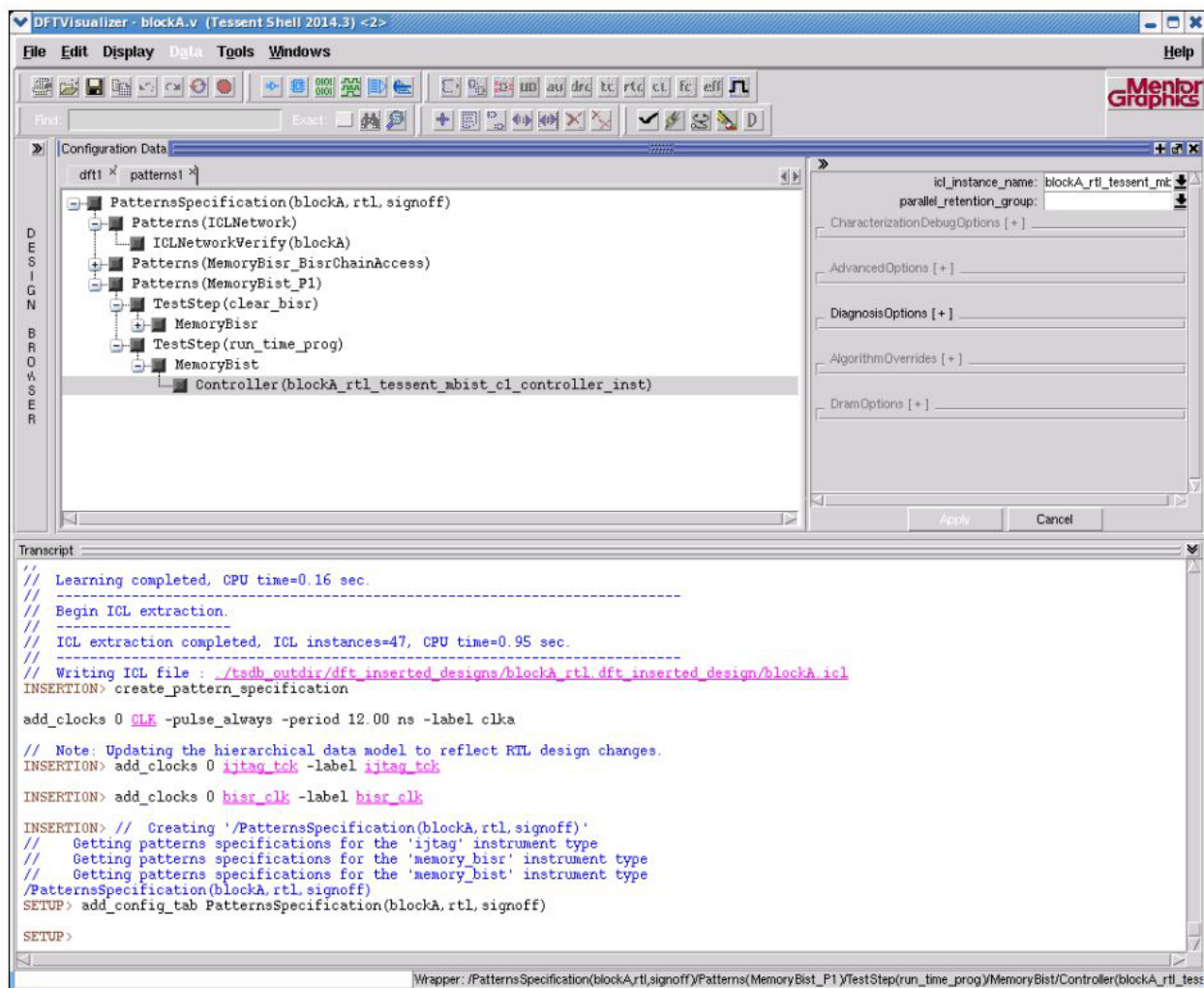
Process Patterns Specification

The Process Patterns Specification step creates the patterns and testbenches.

Figure 2-10. Process Patterns Specification
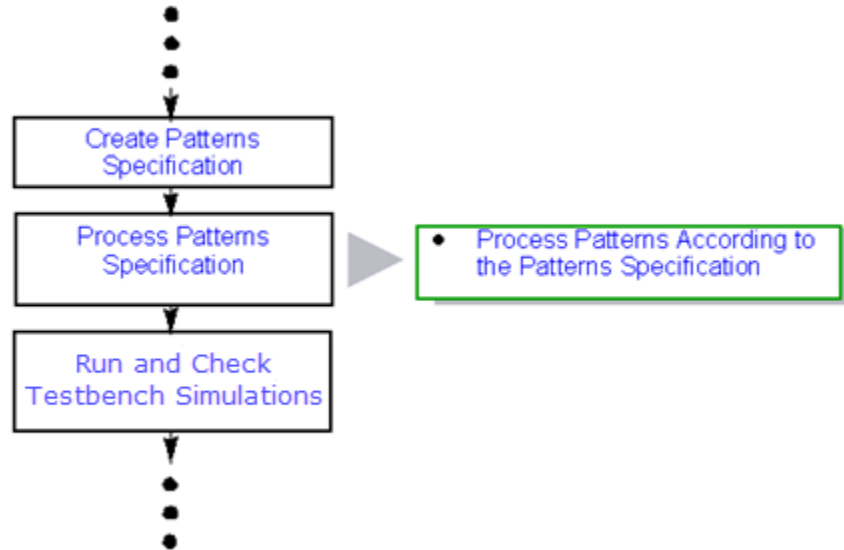
Process Patterns According to the Patterns Specification

In this step, you create the patterns or testbenches according to either the default patterns specification or to the edited patterns specification that you created in the previous step.

Example

The following example generates the testbenches.

```
process_patterns_specification
```

Run and Check Testbench Simulations

The Run and Check Testbench Simulations step runs simulations of the memory BIST verification and then checks the results.

Figure 2-11. Run and Check Testbench Simulations
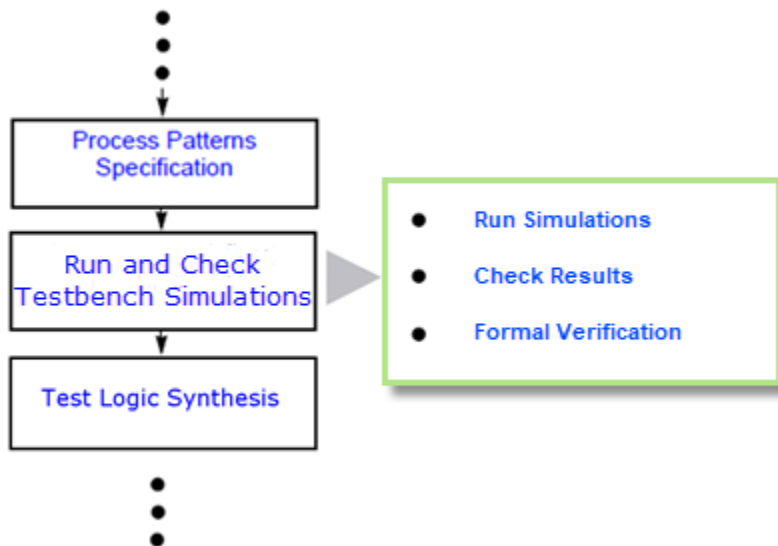
Run Simulations

Use the run_testbench_simulations command to invoke a simulation manager to run a set of simulation testbenches.

The run_testbench_simulations command compiles and simulates the testbenches generated from the process_patterns_specification command that are located in <tsdb_outdir>/patterns/ <design>.patterns_signoff.

For a detailed description of the run_testbench_simulations command and its usage, see the *Tessent Shell Reference Manual*.

Example

The following example runs simulations of all patterns defined in the PatternsSpecification.

```
run_testbench_simulations
```

Check Results

Use the check_testbench_simulations command to check the status of the simulations that were previously launched by the run_testbench_simulations command.

For a detailed description of the check_testbench_simulations command and its usage, see the *Tessent Shell Reference Manual*.

Example

The following example checks the simulation results for errors.

```
check_testbench_simulations
```

Formal Verification

Tessent Shell based products currently do not generate scripts for use with Synopsys Formality.

You can however, set constraints in your design and manually create a script that is used with Formality. For guidance on how this is accomplished, refer to the "Formal Verification" Appendix in the *Tessent Shell User's Manual*.

Test Logic Synthesis

The test logic synthesis process is differs between RTL or gate-level designs. The following sections outline the different options.

_____ **Note** _____

During the synthesis of MemoryBIST logic, two types of warnings related to the optimization of registers may be issued by synthesis tools. These warnings may be ignored as synthesis tools are very reliable. Additionally, formal verification can be used to confirm the functionality is not affected. The warning types are:

- Removal of registers with no fanout, along with the combinational logic driving the inputs.

- Registers with inputs and outputs that are always identical are merged.

Figure 2-12. Test Logic Synthesis
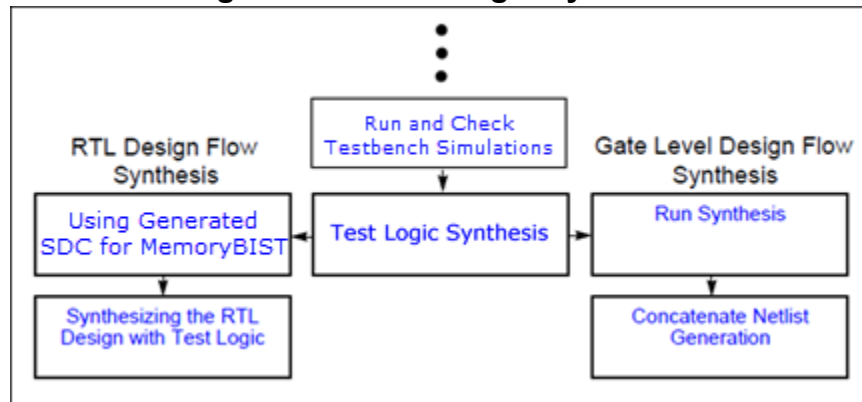
RTL Design Flow Synthesis

The RTL synthesis flow that integrates the MBIST RTL and associated test logic with the design RTL is an automated flow. The following sections outline the process.

Using Generated SDC for MemoryBIST

The extract_icl and extract_sdc commands both create a Synopsys Design Constraints (SDC) file that can be used for synthesis, layout and static timing analysis (STA). Because the extract_sdc command requires ICL, it needs to be run after extract_icl.

When extract_icl is run on a physical block containing sub-blocks, the SDC constraints are generated for the physical module as well as the sub-blocks.

The created SDC is composed of several procedures that can be integrated into a design synthesis script. For more information and examples on how to use the generated SDC procs, refer to the "Timing Constraints SDC" chapter in the *Tessent Shell User's Manual*. Additional information is also provided specific to MemoryBIST Instrument proc usage.

Synthesizing the RTL Design with Test Logic

This process is automated by a script that can be created and then processed by a synthesis tool to synthesize an RTL design that has been DFT inserted.

The write_design_import_script command can be used to generate a script that can be processed by a synthesis tool to load the RTL design that has been DFT inserted. The script file written can be combined with the SDC generated during extract_icl to synthesize a physical block or chip design unit.

For additional synthesis information, refer to the "Synthesis Guidelines for RTL Designs with Tessent Inserted DFT" and "Tessent Shell Workflows" sections in the *Tessent Shell User's Manual*.

Gate Level Design Flow Synthesis

The gate level design flow synthesis is a fully automated flow and only requires the run_synthesis command to synthesize the test logic and integrate into the design,

Run Synthesis

The run_synthesis command only synthesizes test logic RTL contained within the TSDB.

When creating and inserting memory BIST, boundary scan or IJTAG logic, the generated RTL is automatically written to the TSDB during process_dft_specification.

The run_synthesis command invokes a synthesis manager to perform synthesis of the test logic RTL.

For a detailed description of the run_synthesis command and its usage, refer to the *Tessent Shell Reference Manual*.

Example

The following example performs synthesis for a design and can be run at the physical_block or top design level.

```
run_synthesis
```

Concatenate Netlist Generation

When run_synthesis completes successfully, a concatenated netlist of the design that contains the synthesized test logic and modified design modules is automatically created and placed in the dft_inserted_designs directory of the TSDB.

Run and Check Testbench Simulations with a Gate Level Netlist

In order to run MemoryBIST gate level verification on a design with MemoryBIST previously inserted in the RTL, regeneration of the signoff patterns is required. The main steps to do this are outlined below.

Prerequisites

- A gate level design with MemoryBIST previously inserted in the RTL

Procedure

1. Set context to IJTAG patterns

2. Point to the TSDB that was used to do MemoryBIST RTL insertion

3. Read cell and memory libraries

4. Read the gate level netlist

5. Read the design data from the TSDB, excluding the RTL

6. Set current design and check design rules

7. Create and process pattern specification

8. Run and check testbenches

Examples

The following example runs the steps outlined above for a design "cpu_top":

```
set_context patterns -ijtag
set_tsdb_output_directory tsdb_outdir
read_cell_library adk.tcelllib
read_cell_library memory.lib
read_verilog cpu_top.vg
read_design cpu_top -design_id gate -no_hdl
set_current_design cpu_top
check_design_rules
create_patterns_specification
process_patterns_specification
set_simulation_library_sources -v adk.v -v ram.v
run_testbench_simulations
check_testbench_simulations
check_testbench_simulations -report_status
```

Chapter 3
Planning and Inserting MemoryBIST

The topics in this chapter cover the planning, creation and subsequent insertion of the memory BIST logic. The chapter explains the planning step and how you can influence the Tessent MemoryBIST tool to give you a DFT plan, that is, a DftSpecification that matches your intended design. It shows how you can review and further edit this DftSpecification through a variety of tool features, including a GUI. Finally this chapter explains the validation and subsequent execution of the DftSpecification to have Tessent MemoryBIST generate the RTL and insert the logic into your design.

Design and Library Requirements

You must have loaded the design that the memory BIST and repair solution shall be inserted into. You must also load the memory models.

> **Note**
>
> Legacy models you may have from the LogicVision design flow of memory BIST can be read in using the read_core_descriptions command. These are translated internally to the new description described in the Tessent Core Description (TCD)/Memory section. Note that Tessent Shell requires the LogicVision MemoryTemplate name to match the specified CellName as shown below:
>
> ```
> MemoryTemplate(mydram) {
>         MemoryType     : SRAM;
>         CellName       : mydram;
>          .
>          .
>          .
> }
> ```

A design flow requiring memory BIST and repair logic insertion into a gate-level design needs to use the Tessent Cell library. For top-level insertion the library must include pad cells, or a pad cell library must be loaded. For an RTL level flow, only the testbench simulation step at the end of the flow requires a library, as described in "Run and Check Testbench Simulations".

Finally, you must have set the current top level design using the set_current_design command as well as the design hierarchy level of operation using set_design_level.

Specifying and Verifying MemoryBIST Requirements

In this section you learn how to first enable the MemoryBIST product in Tessent Shell, define clocks, set defaults that globally influence the DftSpecification, and how you can review and modify key memory options on a per-instance basis. Additional data you can give to the Tessent MemoryBIST tool are layout location information through a DEF file and power information through an UPF or CPF file.

The section concludes with running design rulechecks that enable you to proceed with creation of the memory BIST DftSpecification in the next step of the design flow. Tessent MemoryBIST then utilizes this DftSpecification to implement the memory BIST solution you want in your design.

Enabling Memory BIST and Repair

After design loading and setting the current level, you first must tell Tessent Shell that you want to add Tessent MemoryBIST, and optionally the memory repair functionality.

You do this through the command set_dft_specification_requirements:

```
SETUP> set_dft_specification_requirements -memory_test on
```

This command has a few parameters that Tessent MemoryBIST sets automatically. For example, Tessent MemoryBIST automatically enables its repair features, if the design has repairable memories loaded.

> **Note**
> If you have repairable memories in the current design, you must insert both the memory BIST and the memory repair logic at the same time for all such memories.

The current memory BIST flow does not allow the addition of memory repair logic independently of the BIST logic. This is a flow restriction and applies to memories in the current design view that previously had BIST inserted, as well as to memories that have not. Tessent MemoryBIST issues a warning or an error, depending on your particular flow case.

> **Tip**
> ⓘ You may choose to disable the repair functionality for all memory instances using the
> set_dft_specification_requirements command, or for a subset of memory instances using the
> set_memory_instance_options command.

You can report all loaded memories, visible in the current design:

```
SETUP> report_memory_instances
```

This reporting command lists the current Tessent MemoryBIST design information at the moment of command invocation. Please observe how the reported information changes, and becomes more and more detailed as you progress through the flow.

Loading Layout Placement Information

Tessent MemoryBIST takes the physical location of memories into consideration when it assigns memories to different controllers.

The physical location is only one of several factors that make up the partitioning of memories among controllers. A few of many other factors include the power consumption, power domains, and test time estimates. You provide the physical layout information through a standard DEF (Design Exchange Format) file. You also have the option to change the physical cluster size, which is a factor expressed as a percentage of the die diagonal, that determines if Tessent MemoryBIST assigns the current memory to a new controller or to an existing one.

After design rulechecks and an initial DftSpecification is created, you have the chance to override the chosen cluster assignment as shown in the example below. This example loads the DEF file for your design, including the physical location information for your memories, and changes the size of the physical cluster to 10%.

```
SETUP> read_def ../data/design/rtl/blockA.def
SETUP> set_memory_instance_options -physical_cluster_size_ratio 10
```

Loading Power Domain Information

Tessent MemoryBIST takes the power domain information of memories into consideration when it assigns memories to different controllers.

The power domain is only one of several factors that make up the partitioning of memories among controllers. Other factors include the power consumption, physical location in the layout, test time estimates, and others as outlined in "MemoryBIST Partitioning Rules". You provide the power domain information through a standard UPF or CPF file, as shown in the example below.

```
SETUP> read_upf ../data/design/rtl/blockA.upf
//  Reading UPF file ../data/design/rtl/blockA.upf ...
//  Finished reading UPF file ../data/design/rtl/blockA.upf.
```

MemoryBIST controllers can only test memories within the same power domain, or power-domain island. A power-domain island is created when a specific power domain lies within a different power domain. MemoryBIST will partition the memory instances and assign controllers within each power-domain island separately. This is done to avoid driving BIST signals across power domain boundaries. To illustrate, one controller can test mem3 and mem4 in the figure below. However, memories mem0 and mem1 are on different power-domain islands causing MemoryBIST to generate a separate controller for each memory.

Figure 3-1. Power-Domain Islands
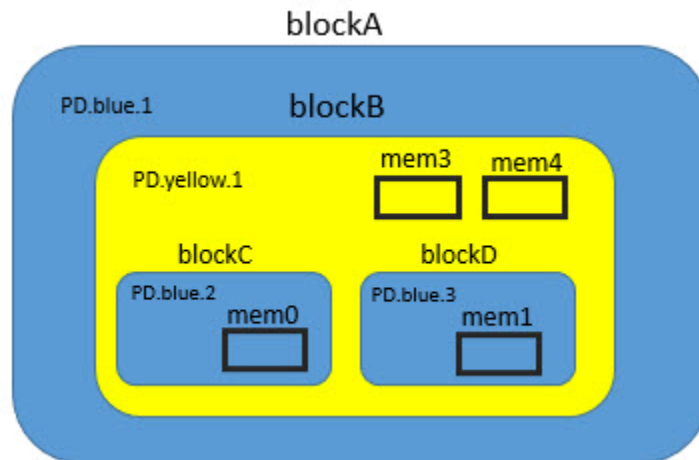
After design rulechecks and during the creation of the initial DftSpecification, you have the option to observe the power domain assignment Tessent MemoryBIST derived from the UPF or CPF files. You can run the report_memory_instances command and reference the power_domain_island information for each memory instance, as shown below.

```
ANALYSIS> report_memory_instances
//
// Memory Instance: 'mem0'
// ---------------------------------------------
// bist_data_in_pipelining              : auto
// physical_cluster_override            :
// power_domain_island                  : pd.blue.2
// test_clock_override                  :
// use_in_memory_bist_dft_specification : on
// use_in_memory_bisr_dft_specification : off
// parent_cluster_module                :
// partitioning_group                   :
// repair_sharing                       : off
//
// Memory Instance: 'mem1'
// ---------------------------------------------
// bist_data_in_pipelining              : auto
// physical_cluster_override            :
// power_domain_island                  : pd.blue.3
// test_clock_override                  :
// use_in_memory_bist_dft_specification : on
// use_in_memory_bisr_dft_specification : off
// parent_cluster_module                :
// partitioning_group                   :
// repair_sharing                       : off
//
// Memory Instance: 'mem3'
// ---------------------------------------------
// bist_data_in_pipelining              : auto
// physical_cluster_override            :
// power_domain_island                  : pd.yellow.1
// test_clock_override                  :
// use_in_memory_bist_dft_specification : on
// use_in_memory_bisr_dft_specification : on
// parent_cluster_module                :
// partitioning_group                   :
// repair_sharing                       : off
//
// Memory Instance: 'mem4'
// ---------------------------------------------
// bist_data_in_pipelining              : auto
// physical_cluster_override            :
// power_domain_island                  : pd.yellow.1
// test_clock_override                  :
// use_in_memory_bist_dft_specification : on
// use_in_memory_bisr_dft_specification : on
// parent_cluster_module                :
// partitioning_group                   :
// repair_sharing                       : off
```

Adding Clocks

Using the add_clocks command, you tell the Tessent MemoryBIST tool the clocks to use and their speed. This information is then used during design rule checking and for partitioning of memories among controllers.

You must be in the setup mode to add clocks and their properties. You can issue the add_clocks command before or after the set_dft_specification_requirements command.

The following example defines the clock named 'CLK' to be a free-running clock with a period of 12ns. The example command also defines a label for this clock. If you do not define a label, Tessent MemoryBIST creates one for you. This label is found later on in the DftSpecification, in the MemoryBIST /Controller wrappers.

```
SETUP> add_clocks CLK -period 12ns -label clka
```

To learn more about adding clocks of different properties and types, like differential clocks, refer to the *Tessent Shell Reference Manual* where there are numerous examples showing how to use the add_clocks command.

Changing Default Settings

You use the DefaultsSpecification to change the default behavior and values of Tessent MemoryBIST. Changes in this specification apply to all controllers and all memories. The next section shows how you can make changes to an individual memory instance or a collection of instances.

You have two ways of changing the entries in the DefaultsSpecification. The first is to edit the file itself and load it into the tool. The second option is to use set_defaults_value command or the more general set_config_value command. The set_defaults_value command is a shortcut to the DftSpecification and PatternsSpecification wrappers in the DefaultsSpecification, whereas with the set_config_value command, you have to explicitly reference the DefaultsSpecification wrapper and user mode.

Examples

Example 1: Loading the DefaultsSpecification File

```
SETUP> read_config_data MyDesign.defaults_specification
```

In this example, the file named MyDesign.defaults_specification is read in. Any specification properties set in this file are used for the subsequent steps of the flow, replacing the respective tool default. Other default settings are not touched. Default settings have an impact in particular for the hardware of the controllers, other DFT elements, like pipelines, and memory partitioning.

Example 2: Changing a Controller Option

This example shows how to set the shared comparator as the new default for the design, instead of the Tessent MemoryBIST default of 'per interface'. You only need to write the part of the DefaultsSpecification you want to change; everything else remains at the tool default. After you created the file, all you have read it as shown in Example 1

```
DefaultsSpecification(user) {
   DftSpecification {
    MemoryBist {
      ControllerOptions {
        comparator_location : shared_in_controller;
      }
    }
   }
}
```

> **Note**
> After having read the DefaultsSpecification file, Tessent MemoryBIST places the information in memory, where you can make additional changes as shown in Example 3 below. Any subsequent changes to the file are considered until it is read in again. Remember, Tessent MemoryBIST operates on data in memory, not on disk. There are very few exceptions to this, one of which is the BISR Chain Order file described in "Planning and Inserting MemoryBIST".

Example 3: Using a Specific Cell Selection instead of the RTL Cells Tessent MemoryBIST is creating

This example has two parts. First you must provide a cell selection library, then you tell Tessent MemoryBIST to use these instead of the built-in ones. This example also shows how to change a default setting through dofile commands, without a DefaultsSpecification file.

In the *Tessent Cell Library Manual* you find several examples of how to declare cells to be used for specific purposes. One important purpose is for clock gating. With the cell selection you instruct Tessent MemoryBIST to use very specific cells for example to implement clock gating or to implement data muxing, instead of the built-in RTL cells that Tessent MemoryBIST would create and use otherwise.

After having read in this library with the read_cell_library command, you need to change the default to activate the cell usage. This example uses the set_defaults_value command in setup mode for this. You do not need a DefaultsSpecification file for this to work. Tessent MemoryBIST is creating the respective DefaultsSpecification wrappers for you in memory.

```
SETUP> set_context dft -no_rtl
SETUP> get_defaults_value DftSpecification/use_rtl_cells
auto
SETUP> set_defaults_value DftSpecification/use_rtl_cells off
off
SETUP> get_defaults_value DftSpecification/use_rtl_cells
off
SETUP> report_config_data DefaultsSpecification(user)

DefaultsSpecification(user) {
  DftSpecification {
    use_rtl_cells : off;
  }
}
```

Adding Constraints Before Design Rule Checking

The previous section shows how to make global changes that apply to all controllers and all memories. Next, how to make changes that impact only one or a collection of memories is discussed. The set_memory_instance_options command is used for all these actions.

Most changes using the set_memory_instance_options command become visible only after design rule checking, since only then are physical placement information and power information evaluated and a first partitioning planning is done by Tessent MemoryBIST. Also done during design rule checking is the evaluating of the memory properties, like repair status.

Nonetheless, you may want to already add certain constraints in setup mode, before design rule checking. One important constraint is removing memories from consideration for BIST.

_____ **Note** _____

If you have repairable memories in the current design, you must insert both the memory BIST and the memory repair logic at the same time for all such memories.

The current memory BIST flow does not allow the addition of memory repair logic independently of the BIST logic. This is a flow restriction and applies to memories in the current design view that previously had BIST inserted, as well as to memories that have not. Tessent MemoryBIST issues a warning or an error, depending on your particular flow case.

_____ **Tip** _____

You may choose to disable the repair functionality for all memory instances using the set_dft_specification_requirements command, or for a subset of memory instances using the set_memory_instance_options command.

Example for Removing a Memory from Consideration for BIST

You can do this with the -use_in_memory_bist_dft_specification option of the set_memory_instance_options command as shown in this example for one selected memory. Of course, you can also specify a list or a collection of memory instances.

```
SETUP> report_memory_instances -limit 2
//
// Memory Instance: 'blockA_l1_i1/blockA_l2_i1/mem1'
// ------------------------------------------------
// bist_data_in_pipelining              : off
// physical_cluster_override            :
// power_domain_island                  :
// test_clock_override                  :
// use_in_memory_bist_dft_specification : auto
// use_in_memory_bisr_dft_specification : auto
// parent_cluster_module                :
//
// Memory Instance: 'blockA_l1_i1/blockA_l2_i1/mem2'
// ------------------------------------------------
// bist_data_in_pipelining              : off
// physical_cluster_override            :
// power_domain_island                  :
// test_clock_override                  :
// use_in_memory_bist_dft_specification : auto
// use_in_memory_bisr_dft_specification : auto
// parent_cluster_module                :
//
// Reached limit of 2, skipping remaining 16 instances.
SETUP> set_memory_instance_options blockA_l1_i1/blockA_l2_i1/mem2 \
-use_in_memory_bist_dft_specification off
SETUP> report_memory_instances -limit 2
//
// Memory Instance: 'blockA_l1_i1/blockA_l2_i1/mem1'
// ------------------------------------------------
// bist_data_in_pipelining              : off
// physical_cluster_override            :
// power_domain_island                  :
// test_clock_override                  :
// use_in_memory_bist_dft_specification : auto
// use_in_memory_bisr_dft_specification : auto
// parent_cluster_module                :
//
// Memory Instance: 'blockA_l1_i1/blockA_l2_i1/mem2'
// ------------------------------------------------
// bist_data_in_pipelining              : off
// physical_cluster_override            :
// power_domain_island                  :
// test_clock_override                  :
// use_in_memory_bist_dft_specification : off
// use_in_memory_bisr_dft_specification : auto
// parent_cluster_module                :
//
// Reached limit of 2, skipping remaining 16 instances.
SETUP> check_design_rules
[...]
ANALYSIS> report_memory_instances -limit 2
//
// Memory Instance: 'blockA_l1_i1/blockA_l2_i1/mem1'
// ------------------------------------------------
// bist_data_in_pipelining              : off
// physical_cluster_override            :
// power_domain_island                  :
// test_clock_override                  :
```

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

```
// use_in_memory_bist_dft_specification : on
// use_in_memory_bisr_dft_specification : off
// parent_cluster_module                :
//
// Memory Instance: 'blockA_l1_i1/blockA_l2_i1/mem2'
// ----------------------------------------------
// bist_data_in_pipelining              : off
// physical_cluster_override            :
// power_domain_island                  :
// test_clock_override                  :
// use_in_memory_bist_dft_specification : off
// use_in_memory_bisr_dft_specification : off
// parent_cluster_module                :
//
// Reached limit of 2, skipping remaining 16 instances.
ANALYSIS>
```

Creating, Modifying, and Validating a MemoryBIST DFT Specification

Once a successful design rule check is completed, you can begin creating the DftSpecification.

During the design rule checking all loaded setup data, including the optional physical location and power domain information, is processed. Tessent MemoryBIST has already completed initial analysis and you can report or introspect information to analyze the results and make needed modifications before you create the initial DftSpecification. You also have the option of modifying some of the analysis results before you create your initial DftSpecification.

At this point of the memory BIST insertion flow you want to iterate through the process of creating DftSpecifications, reporting or introspecting values and settings and modifying these values, settings, and constraints to reach a final DftSpecification. Tessent MemoryBIST then utilizes this DftSpecification to implement the memory BIST solution you want in your design.

Commands that become important in this part of the flow are report_memory_instances, get_memory_instance_option, create_dft_specification, and the DftSpecification introspection and editing commands, like add_config_element, get_config_value, or set_config_value. You can visualize the DftSpecification using display_specification.

The section concludes with validating the edited MemoryBIST DftSpecification, enabling you to then create the hardware and insert the logic into the design as described in the next section of the design flow.

Adding Constraints Before Creating an Initial DftSpecification

Some constraints have a broad impact on the DftSpecification you are going to generate a few steps later in the flow.

You can use the set_memory_instance_options command to change those constraints. As an example, this section shows the steps you can take to make changes with the memory partitioning.

The check_design_rules command analyzes the design and attaches the power domain and placement information to each memory instance that you may have loaded earlier in the design flow with the UPF or CPF file and DEF file. Tessent MemoryBIST considers this information when assigning memories to different controllers. The UPF or CPF file determines the power domain information and you cannot change or diverge from the information contained in the file. However, the -physical_cluster_size_ratio and -physical_cluster_override arguments of the set_memory_instance_options command can configure the placement-based grouping or physical clustering of memories for controller assignment. MemoryBIST performs the grouping in conjunction with other controller partitioning rules, as described in "MemoryBIST Partitioning Rules".

Tessent MemoryBIST creates a rectangle to fully enclose the current design and extracts the length of its diagonal, as shown in the figure below. The -physical_cluster_size_ratio argument specifies a ratio that is the percentage of a physical cluster's diagonal with respect to the total diagonal distance. MemoryBIST adds a memory instance to a given cluster if it does not cause the diagonal of the cluster to exceed the threshold ratio, otherwise it creates a new cluster. The default ratio is 20 percent, but you can change this percentage value by specifying an integer between 0 and 100. A value of 0 causes each memory instance to be in its own cluster, and a value of 100 results in all memory instances being in the same (single) cluster. The example at the end of this section shows the effect of specifying a ratio of 100, forcing all memory instances into a single cluster. This only occurs if no other MemoryBIST partitioning rules need enforcement.

Figure 3-2. MemoryBIST Physical Cluster Grouping
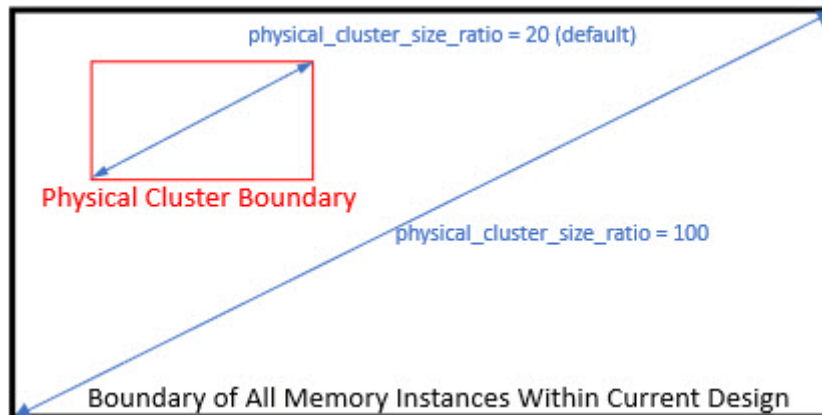
The example in "Re-Creating the DftSpecification" shows the usage of the -physical_cluster_override *string* switch for the set_memory_instance_options command, where *string* identifies a cluster ID name. This switch gives you the ability to exclude memories from physical clusters that would normally be assigned based on placement information and place them into a cluster specified by the cluster ID name.

Note

If you have repairable memories in the current design, you must insert both the memory BIST and the memory repair logic at the same time for all such memories.

The current memory BIST flow does not support the addition of memory repair logic independently of the BIST logic. This flow restriction applies to memories in the current design view that previously had BIST inserted, as well as to memories that have not. Tessent MemoryBIST issues a warning or an error, depending on your particular flow case.

Tip

You may choose to disable the repair functionality for all memory instances using the set_dft_specification_requirements command, or for a subset of memory instances using the set_memory_instance_options command.

Example 1

This example shows how to use the -physical_cluster_size_ratio argument of the set_memory_instance_options command to change the memory partitioning. This example changes the partitioning to have one controller for all memories in the current design. All memories are on the same power domain and clock domain, and no other partitioning rules are in effect.

After you load the design and elaborate it, you load the power domain (UPF file) and placement (DEF file) information, then specify the DFT requirements and clocks.

```
SETUP>set_current_design blockA
SETUP>read_upf ../data/design/rtl/blockA.upf
SETUP>read_def ../data/design/rtl/blockA.def
SETUP>set_dft_specification_requirements -memory_test on
SETUP>set_design_level sub_block
SETUP>add_clock CLK -period 12ns -label clka
```

Design rule checks are run and the initial DFT specification is created with the default physical cluster size ratio setting of 20.

```
SETUP>check_design_rules
ANALYSIS>create_dft_specification
```

You then report the DFT specification to show the resulting controller and memory partitioning. The tool creates two controllers with the memory and placement for this example.

```
ANALYSIS>report_config_data
  MemoryBist {
    ijtag_host_interface : Sib(mbist);
    Controller(c1) {
      clock_domain_label : clka;
      Step {
        MemoryInterface(m1) {
          instance_name : mem1;
          output_enable_control : system;
        }
        MemoryInterface(m2) {
          instance_name : mem2;
          output_enable_control : system;
        }
      }
      Step {
        MemoryInterface(m3) {
          instance_name : mem3;
        }
      }
    }
    Controller(c2) {
      clock_domain_label : clka;
      Step {
        MemoryInterface(m1) {
          instance_name : mem4;
        }
        MemoryInterface(m2) {
          instance_name : mem5;
        }
        MemoryInterface(m3) {
          instance_name : mem6;
        }
      }
    }
  }
```

The requirement for this design block is a single controller for all memory instances. Therefore you specify 100 for the -physical_cluster_size_ratio switch of the set_memory_instance_options command, which puts all memory instances in the same cluster.

```
ANALYSIS>set_system_mode setup
SETUP>set_memory_instance_options -physical_cluster_size_ratio 100
```

You then create the DFT specification, replacing the original specification.

```
SETUP>check_design_rules
ANALYSIS>create_dft_specification -replace
```

You report the DFT specification and observe the partitioning is a single physical cluster with a single MemoryBIST controller.

```
ANALYSIS>report_config_data
MemoryBist {
    ijtag_host_interface : Sib(mbist);
    Controller(c1) {
      clock_domain_label : clka;
      Step {
        MemoryInterface(m1) {
           instance_name : mem1;
           output_enable_control : system;
        }
        MemoryInterface(m2) {
           instance_name : mem2;
           output_enable_control : system;
        }
      }
      Step {
        MemoryInterface(m3) {
           instance_name : mem3;
        }
        MemoryInterface(m4) {
           instance_name : mem4;
        }
        MemoryInterface(m5) {
           instance_name : mem5;
        }
        MemoryInterface(m6) {
           instance_name : mem6;
        }
      }
    }
  }
```

Parameter Selection Impacts on Performance and Area

Some parameters specified in the DftSpecification and DefaultsSpecification have more of an impact on design performance and area than others. While it is difficult to quantify an exact impact, understanding whether certain parameter selections tend to improve or degrade performance and area is important to consider when inserting a memory BIST solution in your design.

Table 3-1 below shows the memory BIST parameters within various wrappers in the DftSpecification and DefaultsSpecification that can have a positive or negative impact on performance and area depending on which options are selected.

- Parameters that can improve performance or reduce area are marked with a **"+"** entry, indicating a **positive** impact.

- Parameters that typically increase area, and may also decrease performance in some cases, are marked with a **"-"** entry, indicating a **negative** impact.

- The Table Notes specified below Table 3-1 provide additional details.

Table 3-1. Parameter Impact on Performance and Area

Wrapper	Parameter	Perf	Area	Notes
DftSpecification Parameters				
AdvancedOptions	pipeline_controller_outputs	+	-	1
	observation_xor_size		-	2,3
	shared_comparators_per_go_id		+	2,3
	use_multicycle_paths	+		16
AlgorithmResourceOptions	data_register_bits		-	4,5
	counter_a_bits		-	4,3
	delay_counter_bits		-	4,3
	max_data_inversion_address_bit_index	-	-	4,3
	a_equals_b_command_allowed	-	-	6
	address_segment_x0_y0_allowed	-	-	
	max_x0_segment_bits	-	-	7
	max_y0_segment_bits	-	-	7
	soft_instruction_count	-	-	17

Table 3-1. Parameter Impact on Performance and Area (cont.)

Wrapper	Parameter	Perf	Area	Notes
MemoryCluster	pipeline_cluster_inputs	+	-	8
	pipeline_cluster_outputs	+	-	8
	repair_analysis_present	-	-	9
RepairOptions	row_bira_location		-	10
DiagnosisOptions	comparator_selection_mux		-	11
	StopOnErrorOptions/failure_limit		-	4
Step	comparator_location		-	12
Step/MemoryInterface	repair_analysis_present		-	9
	scan_bypass_logic	-	-	13
	local_comparators_per_go_id		+	2,3
	data_bits_per_bypass_signal		-	2,3,15
DefaultsSpecification Parameters				
RepairOptions	max_repair_group_size	-	+	3,14

Notes for Table 3-1:

1. Parameter option should be set to "on" for high speed controllers, especially when a_equals_b_command_allowed is set to "on", or if a large value is specified for max_data_inversion_bit_index.

2. Area is inversely proportional to the integer value specified for this option. The maximum area is realized when a value of "1" is specified.

3. Performance might be affected when large values are specified.

4. Area is proportional to the value specified for this parameter.

5. Increases data path width from the controller to all memories that potentially creates routing congestion.

6. If this parameter is set to "on", it is also recommended to set AdvancedOptions/ pipeline_controller_outputs to "on" for high speed controllers.

7. It is recommended to set the value of this parameter to "0" for minimum area, or to "1" whenever Operation/Cycle/AdvancedSignals/row_address_count_enable or column_address_count_enable properties are present in the operation sets embedded in the controller. Values larger than "1" increase area and degrade the performance of the address counters.

8. Use this parameter if the memory cluster does not already have registered inputs or outputs.

9. If using IO/column repair, limit the number of IO's in the same column segment to 64 to reduce the performance impact.

10. Area is minimum when the parameter is set to "controller", but more wires are added between the controller and memory to connect the repair bus, which potentially creates routing congestion.

11. Area is minimum when the parameter is set to "off", however bit-level resolution for diagnostics is no longer possible when local_comparators_per_go_id or shared_comparators_per_go_id are set to a value greater than "1".

12. Area is minimum when the parameter is set to "shared_in_controller", but routing congestion might increase around the controller. Set the parameter to "per_interface" to reduce routing congestion because of a large number of memories or wide data paths.

13. Area is proportional to the number of bits in the data path if the value specified for this parameter is different than "none". Specifying the value of "sync_mux" provides best performance and testability, but has maximum area impact. Specifying the value of "async_mux" reduces area, but typically has a very negative impact on performance and could introduce untestable faults.

14. Area is inversely proportional to the group size. However, a very large group size might affect yield.

15. Specifying a value other than "1" may have an impact on timing closure and ATPG scan coverage. Refer to the "Reducing Bypass and Observation Logic Within the Memory Interface" topic for more information.

16. Parameter option should be set to "on" for high-speed controllers.

17. Specifying a value for soft_instruction_count is a trade-off between area and the complexity of the soft algorithm you can download at runtime. Typically, a soft_instruction_count of 4 to 8 is sufficient for most test algorithms. Specifying a large number of instructions results in a high hardware cost to implement the microprogram memory (typically 50 registers per instruction). However, if the number of instructions is small, it may limit you in the types of algorithms that you can implement and may require that you segment the test algorithm into several shorter phases and apply them in different TestSteps. For this case, you must use the preserve_bist_inputs property if a phase relies on the memory state left by a previous phase.

Additional topics that address parameter impacts on performance and area are listed below:

Reducing Bypass and Observation Logic Within the Memory Interface

To achieve high logic test coverage, Tessent MemoryBIST implements memory bypass and scan observation registers in the memory interface. The bypass registers provide observability of the fan-in cone of the memory data inputs, and provide controllability to the fanout cone of the memory data outputs. The observation registers provide observation points for the fan-in logic reaching the memory address and control inputs.

The MemoryBIST area can be minimized by adjusting the number of bypass and observation registers implemented in the memory interface. The following hardware adjustments are introduced:

- **Reduce the number of synchronous bypass registers** — Each data output port requires one set of bypass registers. Prior to the 2020.1 Tessent release, the number of registers had to equal the word size. Now, the reduction feature enables the register count to range from the word size down to 1 for each data output port.

- **Reduce the number of control/address observation registers** — The memory address and control signals are combined into observation registers. Prior to the 2020.1 Tessent release, the signal-to-register ratio was limited from 2 to 8. Now, the area reduction feature extends the range of the valid ratios. The register count can range from the total of all addresses and control signals down to 1.

Overview of Bypass and Observation Logic

Tessent MemoryBIST interacts with the memory under test through the memory interface. The control circuitry implemented in the memory interface is illustrated in the figure below.

The multiplexers that select between the functional and MBIST signals are shown on the left. The common set of observation registers, labeled "O", observe the memory address and control inputs. The synchronous bypass registers, labeled "B" at the bottom of the figure, implement the bypass path from the memory data input to the data output. The fanout of the memory data output to the functional logic and the MBIST comparator circuit is shown on the right. The XOR logic, which compresses multiple signals, are highlighted in red.

Figure 3-3. Memory Observation and Synchronous Bypass Registers
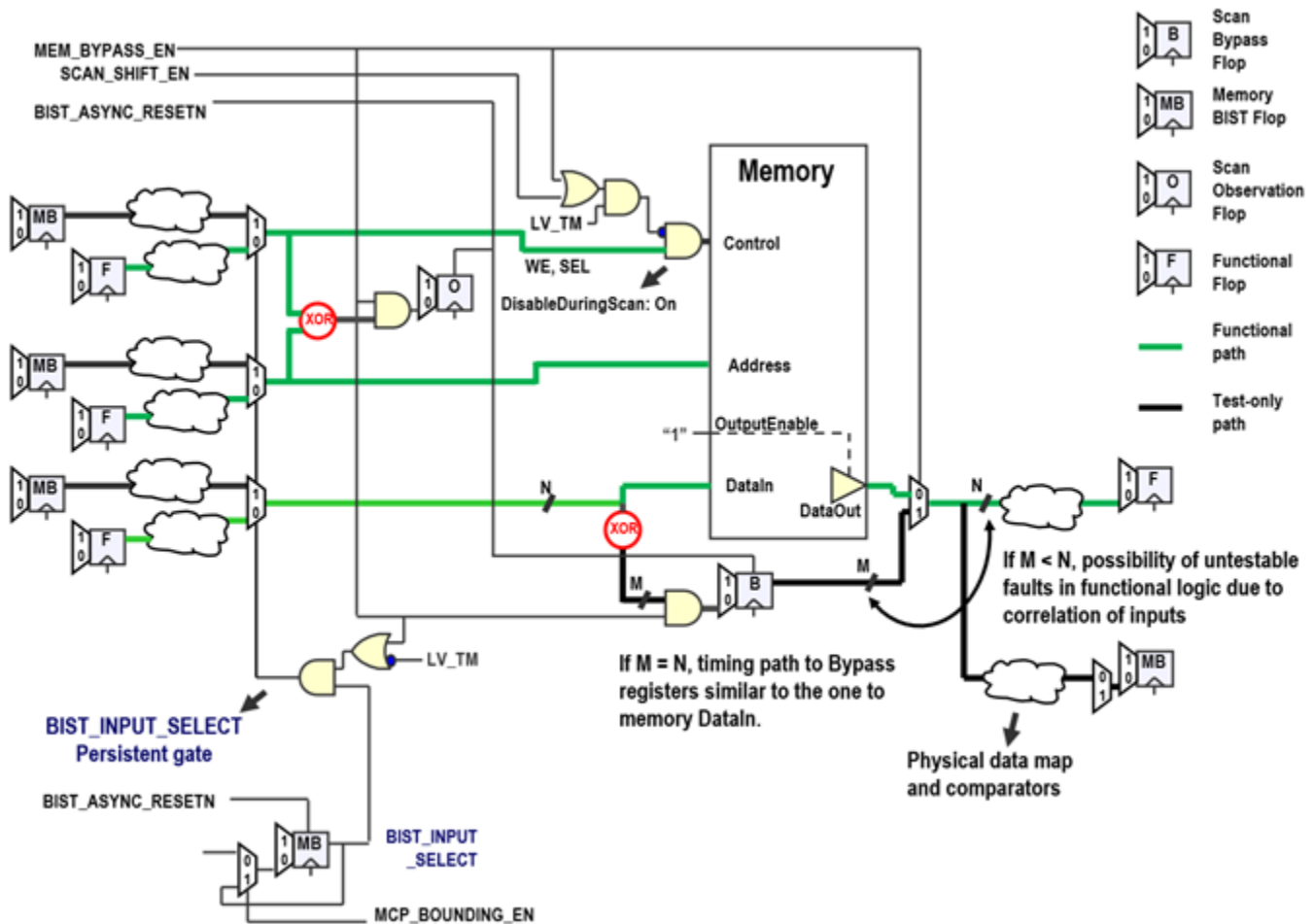
The MemoryBIST area can be minimized, especially for a controller implementing local comparators, by adjusting the number of bypass and observation registers implemented in the memory interface. The area savings achieved must be weighed against potential impacts on ATPG coverage, as well as timing impacts to the observation and bypass registers because of the depth of XOR trees. Refer to "Bypass and Observation Logic Reduction Requirements, Assumptions and Limitations" for more information.

Logic Reduction Implementation

The bypass and observation logic reduction level implemented is defined, generated, and inserted in the current DFT insertion pass, together with the MemoryBIST instruments.

The DftSpecification properties shown below are used to configure the number of bypass and observation registers inserted into the memory interface:

```
DefaultsSpecification(policy) {
  DftSpecification {
    MemoryBist {
      MemoryInterfaceOptions {
        observation_xor_size :  off | 1..MaxPosInt | all; // default: 3
        data_bits_per_bypass_signal : 1..MaxPosInt | all; // default: 1
        scan_bypass_logic : async_mux | none | sync_mux | from_library ;
      }
    }
  }
}

DftSpecification(module,id) {
  MemoryBist {
    Controller(id) {
      AdvancedOptions {
        observation_xor_size : off | 1..MaxPosInt | all; // *DefSpec
      }
    }
    Step {
      MemoryInterface(id) {
        observation_xor_size : auto | off ;
        data_bits_per_bypass_signal : 1..MaxPosInt | all ; //*DefSpec
        scan_bypass_logic : async_mux | none | sync_mux |
                            from_library ; //*DefSpec
      }
    }
  }
}
```

By default for each memory interface, observation logic reduction ratio of address and control signals is 3 and no reduction is done on the bypass logic, which is equal to the memory data output port word size.

The sections that follow provide information on how to use these properties to achieve the wanted level of observation and bypass logic reduction. Refer to "Logic Reduction Examples" to see implementation examples illustrating varying degrees of logic reduction.

Adjusting the Number of Scan Observation Registers

You can disable the insertion of observation logic for a controller by setting AdvancedOptions observation_xor_size to off. Disabling insertion for specific controller memory interfaces is done by specifying Step/MemoryInterface/observation_xor_size to off.

The DftSpecification AdvancedOptions/observation_xor_size property is used to specify the reduction ratio of address and control signals to observation registers through an XOR tree. The tool provides the flexibility of assigning one address/control signal per observation register, up to combining all signals into one observation register. One consideration for limiting the

reduction ratio of address and control signals is to avoid making the at-speed timing paths to the observation registers become more critical than the paths to the memory.

The reduction ratio of address and control signals to observation registers can be set for all memory interface modules by specifying the observation_xor_size property of the DefaultsSpecification.

Adjusting the Number of Memory Bypass Registers

The insertion of memory bypass registers is determined from the Memory TCD TransparentMode property by default. The DftSpecification Step/MemoryInterface/ scan_bypass_logic property can be used to override the TCD setting for the specified memory interface. Either of these property settings must be different than none to implement insertion of memory bypass registers. The synchronous bypass is the default and most common scheme. For the asynchronous bypass scheme, the XOR reduction is implemented without the bypass register.

The DftSpecification Step/MemoryInterface/data_bits_per_bypass_signal property is used to specify the number of functional data input signals combined in an XOR tree to a memory bypass register. The setting is meaningful when building bypass logic around a RAM. The tool provides the flexibility of assigning one data input per bypass register, up to combining all data input signals from a logical port into a 1-bit register. As with observation registers, limiting the bypass reduction ratio should be considered to avoid making the at-speed timing paths to the bypass registers more critical than the paths to the memory.

The reduction ratio of data signals to bypass registers can be set for all memory interface modules by specifying the data_bits_per_bypass_signal property of the DefaultsSpecification.

Bypass and Observation Logic Reduction Requirements, Assumptions and Limitations

The following are the requirements, assumptions, and limitations for implementing and utilizing bypass and observation logic reduction for a memory interface in Tessent Shell:

1. This feature is only available for the Tessent Shell design flow.

2. The features are fully backward compatible. The default value of 1 for the data_bits_per_bypass_signal property matches prior implementations. For the observation_xor_size property, all prior values are available and the default ratio of 3 remains unchanged.

3. When data_bits_per_bypass_signal is greater than 1, ATPG coverage may be impacted because of the correlation of data applied to the functional circuit in the memory fanout. The results are variable from one circuit to another, and it is very difficult to find the root cause when the coverage is low. Therefore, debugging poor test coverage may be difficult.

4. Differences in timing between the memory inputs and the bypass and observation registers can cause timing violations during at-speed scan tests. The paths to the bypass and observation registers can become more critical than the paths to the memory, because of the depth of the XOR tree.

5. When the Memory TCD DataOutStage property is specified as StrobingFlop, the memory bypass registers are reused to pipeline the memory data output to the MBIST circuit. In this configuration, data_bits_per_bypass_signal property must be specified as 1. The tool issues an error during DftSpecification validation if this condition is not met.

Logic Reduction Examples

The following examples show the DftSpecification implementations for varying degrees of observation and bypass logic insertions.

Full Observation Registers and Reduced Bypass Registers

In this example, one observation register is assigned for each address and control signal, for all memory interfaces of controller c1. For memory interface m2, the memory word width is 12 bits. The bypass register count is reduced from 12 to 2. Eight data input signals are combined into the first register and four data input signals are combined into the second register.

```
DftSpecification(blka,rtl) {
  MemoryBist {
    Controller(c1) {
      AdvancedOptions {
        observation_xor_size : 1;
      }
      Step {
        MemoryInterface(m1) {
          instance_name : unita_i1/SYNC_1R1W_512x12_i1;
        }
        MemoryInterface(m2) {
          instance_name : unita_i1/SYNC_1R1W_512x12_i2;
          data_bits_per_bypass_signal : 8;
          scan_bypass_logic : sync_mux;
        }
      }
    } // Controller
  }
}
```

Minimum Observation and Bypass Logic

In this example, one observation register inserted for all memory interfaces of controller c1. For memory interface m1, the memory has one data output port 12 bits wide, and the bypass register count is reduced from 12 to 1. For memory interface m2, the memory has two data output ports, each 16 bits wide. The bypass register count for this interface is reduced from 32 to 2.

```
DftSpecification(blka,rtl) {
  MemoryBist {
    Controller(c1) {
      AdvancedOptions {
        observation_xor_size : all;
      }
      Step {
        MemoryInterface(m1) {
          instance_name : unita_i1/SYNC_1R1W_512x12_i1;
          data_bits_per_bypass_signal : all;
        }
        MemoryInterface(m2) {
          instance_name : unita_i1/SYNC_2RW_512x16_i1;
          data_bits_per_bypass_signal : all;
        }
      }
    } // Controller
  }
}
```

Disabling Observation and Bypass Logic for One Memory

In this example, memory interface m1 uses the default ratios. The TransparentMode property in
its Memory TCD determines the bypass logic setting. No observation or bypass logic is inserted
for memory interface m2.

```
DftSpecification(blka,rtl) {
  MemoryBist {
    Controller(c1) {
      AdvancedOptions {
        observation_xor_size : 3;
      }
      Step {
        MemoryInterface(m1) {
          instance_name : unita_i1/SYNC_1R1W_512x12_i1;
          observation_xor_size : auto;
          scan_bypass_logic : from_library;
        }
        MemoryInterface(m2) {
          instance_name : unita_i1/SYNC_1R1W_512x12_i2;
          observation_xor_size : off;
          scan_bypass_logic : none;
        }
      }
    } // Controller
  }
}
```

Disabling Observation Logic for all Memories

In this example, no observation logic is inserted, and the default bypass register ratio is 3 for all
memory interfaces of controller c1.

```
DftSpecification(blka,rtl) {
  MemoryBist {
    Controller(c1) {
      AdvancedOptions {
        observation_xor_size : off;
      }
      Step {
        MemoryInterface(m1) {
          instance_name : unita_i1/SYNC_1R1W_512x12_i1;
        }
        MemoryInterface(m2) {
          instance_name : unita_i1/SYNC_1R1W_512x12_i2;
        }
      }
    } // Controller
  }
}
```

Pipeline Stage Options and Locations

MemoryBIST incorporates multi-cycle path (MCP) declarations in the generated SDC and STA scripts to facilitate timing closure for high-speed controllers. The tool may declare MCPs for timing paths internal to the controller that operate at half speed, IO repair logic, and for diagnosis logic. However, MemoryBIST performs memory operations at the full memory clock rate over paths declared as single-cycle timing paths. You can pipeline the paths to and from the memories to achieve timing closure. This section describes where the pipeline stages are located and which DftSpecification parameters define their properties.

In the standard MemoryBIST insertion flow, Figure 3-4 shows where you can place pipeline stages for designs that locate comparators in the MemoryBIST controller (Step/comparator_location : shared_in_controller). Figure 3-5 shows the pipeline stage locations for designs that locate comparators in the memory interface (Step/comparator_location : per_interface).

The following legend identifies the groups of pipeline stage locations in the figures and the DftSpecification properties you can specify to configure them into your design.

- (P) — Pipeline stage locations in the MemoryBIST controller. You include these single-register pipeline stages with the "AdvancedOptions/pipeline_controller_outputs : on" setting in the DftSpecification. This is a global setting for all controller outputs and they are not included by default.

- (P) — Pipeline stage locations in the Memory Interface. You include these pipeline stages by specifying the Step/bist_data_in_pipelining and Step/bist_data_out_pipelining properties in the DftSpecification to a setting other than the "off" default.

  The bist_data_in_pipelining property defines the presence and depth of the pipeline stages you want to add to all BIST inputs in the memory interface and the

bist_data_out_pipelining property specifies the presence of a single-register pipeline stage for the BIST data output.

_____ **Note** _____

MemoryBIST does not create BIST data out pipeline registers for ROM memory types.

_____

- ⦿ — Pipeline stages that MemoryBIST automatically infers, based on your selection of previously described pipeline options.

Figure 3-4. Standard Flow Pipeline Stage Locations (Comparators in MemoryBIST Controller)

Figure 3-5. Standard Flow Pipeline Stage Locations (Local Comparators)
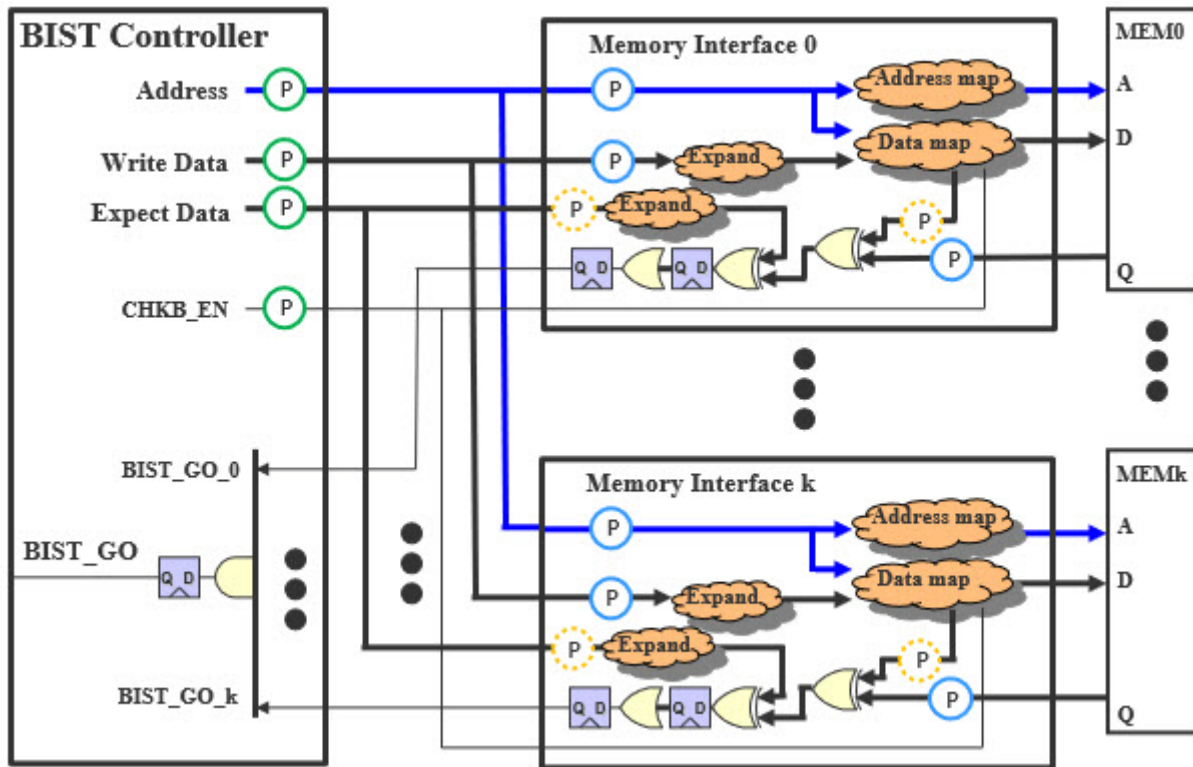
In a Shared Bus MemoryBIST insertion flow, Figure 3-6 shows where you can place pipeline stages. The following legend identifies the groups of pipeline stage locations in the figure and the DftSpecification properties you can specify to configure them into your design.

- (P) — MemoryBIST controller pipeline stage locations. You include these single-register pipeline stages with the "AdvancedOptions/pipeline_controller_outputs : on" setting in the DftSpecification. This is a global setting for all controller outputs and they are not included by default.

  ___ **Tip** ___

  ⓘ Setting the DftSpecification AdvancedOptions/pipeline_controller_outputs property to "on" is essential for high-speed controllers.

- (P) — Cluster pipeline stage locations. You can include these single-register pipeline stages by specifying the MemoryCluster/pipeline_cluster_inputs and MemoryCluster/pipeline_cluster_output properties in the DftSpecification to "on". They are not included by default.

Figure 3-6. Shared Bus Flow Pipeline Stage Locations
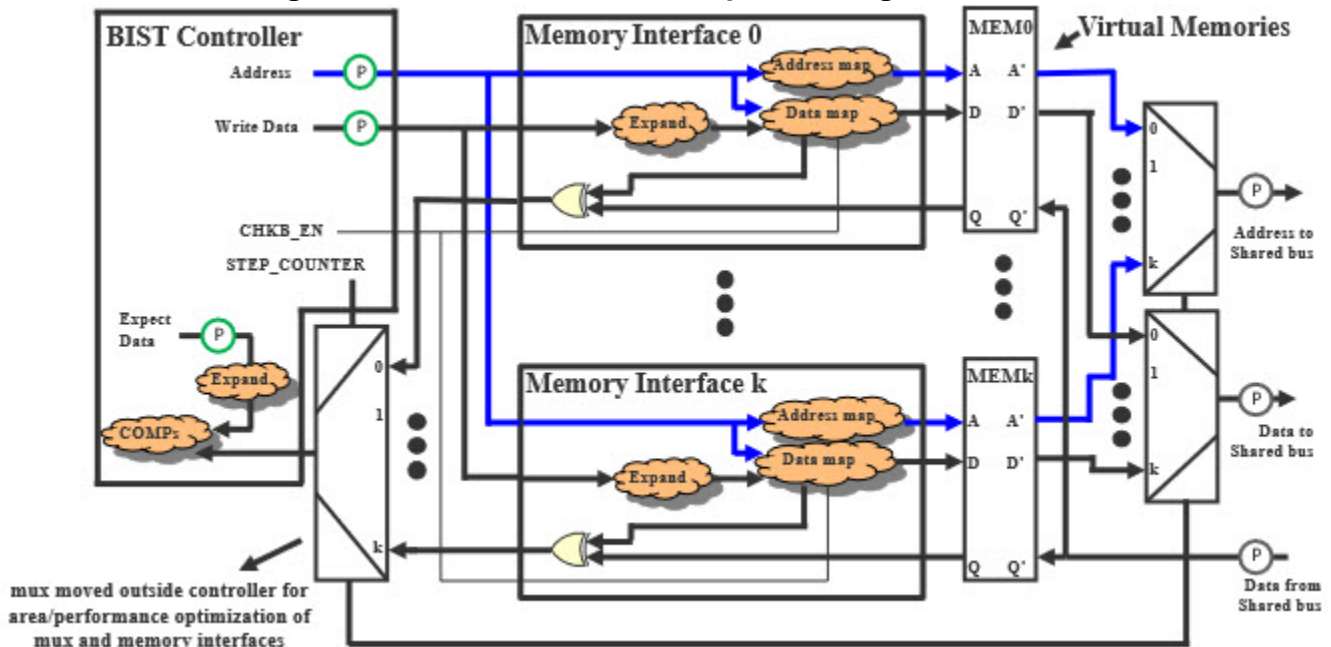
MemoryBIST Partitioning Rules

Tessent MemoryBIST follows certain partitioning rules to separate and group memories. The rules are outlined in this section so you are aware of how this is done and can make any needed modifications as you iterate through the DftSpecification creation process. Controller Compatibility Rules are denoted as "CCRx" and Step Compatibility Rules are denoted as "SCRx" in the following section.

The memory BIST controller compatibility partitioning in the DftSpecification is performed by the following CCR rules:

- **CCR1:** Memories of the different types (RAMs/ROMs/DRAMs) are assigned to separate controllers. Each controller is testing memories of the same type exclusively.

- **CCR2:** All DRAMs on the same controller must have identical row, column and bank dimensions.

- **CCR3:** MemoryBIST separates memories into groups such that the physical region, clock domain, power domain, and memory cluster are identical for all memories in a common group. When multiple clock domains drive a multi-port memory, the memory is associated to the clock domain with the fastest frequency. Refer to "Shared Bus Interface MemoryBIST Implementation Flow" for information on creating a memory cluster on a Shared Bus interface.

- **CCR4:** Memories designated to different partitioning group labels are assigned to different controllers. Refer to the set_memory_instance_options -partitioning_group

group_label command and option for information on assigning group labels to memory instances.

The memories are further separated into compatibility groups such that all memories in a group can be tested in parallel in a single memory controller step. Memories of different sizes are forced into separate groups if testing them in parallel takes longer than testing them in series. This happens when one memory has many row addresses and few column addresses, and the other one has many columns addresses and few row addresses.

The memory BIST step compatibility partitioning in the DftSpecification is performed by the following SCR rules:

- **SCR1:** All memories must use the same algorithm. Refer to the Algorithm property in the Core/Memory wrapper of the memory library for further information.

- **SCR2:** All memories must use the same operation set. Refer to the OperationSet property of the Core/Memory wrapper in the memory library for further information.

- **SCR3:** All memories must be of the same type of SRAM, ROM, or DRAM.

- **SCR4:** All DRAMs must have the same number of row, column, and bank address bits. For information about the address register segment sizes, refer to the LogicalAddressMap wrapper description in the Memory/AddressCounter/Function wrapper.

- **SCR5:** The column segments for all memories must have the same low value for CountRange. Likewise, the row segments for all memories must have the same low value for CountRange. The high value may be different. For information about the CountRange property, refer to the CountRange parameter description in the Memory/AddressCounter/Function wrapper.

- **SCR7:** The bit groupings of all memories must be either all even or all odd, except for memories that have a bit slice of "1". For a description of bit grouping, refer to BitGrouping parameter in the Core/Memory wrapper.

- **SCR8:** All memories must have identical bist_data_out_pipelining option settings. For a chained-memory configuration, this only applies to the last memory in the chain. Refer to the bist_data_out_pipelining parameter found in the in the Step wrapper of the DftSpecification/MemoryBist/Controller wrapper for further information.

- **SCR9:** All memories must have identical DataOutStage option settings. Refer to the DataOutStage parameter in the in the Core/Memory wrapper for further information.

- **SCR10:** The memory groups are separated such that max_power_per_step and max_memories_per_step are not exceeded.

  These parameters are set in the DefaultsSpecification/DftSpecification/MemoryBist wrapper. The power calculation is based only on the estimated memory power according to the specified MilliWattsPerMegaHertz property in the Core/Memory wrapper multiplied by the frequency of the clock domain. It does not include the power

consumed by the controller itself or the rest of the surrounding functional logic. The latter can be significant for a large clock domain, and it might be necessary to disable the functional logic during memory BIST.

- **SCR11:** The resulting memory groups are then assigned to controller steps such that the constraints, specified by the max_test_time_per_controller and max_steps_per_controller properties, are not exceeded.

  These parameters are set in the DefaultsSpecification/DftSpecification/MemoryBist wrapper. Memories that are not part of the same physical region, are not on the same clock domain, or do not share a common memory cluster are never tested by the same controller. Likewise, non-identical DRAMs are never tested by the same controller.

Creating the DftSpecification

The next step in the flow is to create your initial DftSpecification.

In most cases, there is no need to edit this specification and you can continue with the creation and insertion of the Memory BIST hardware. The following sections show you how you can report and introspect the data in the DftSpecification. However, at first you must create one. You use the create_dft_specification command for this:

```
ANALYSIS> set spec [ create_dft_specification ]
//  sub-command: create_dft_specification
//
//  Begin creation of DftSpecification(blockA,rtl)
//    Creation of RtlCells wrapper
//    Creation of IjtagNetwork wrapper
//    Creation of MemoryBist wrapper
//    Creation of MemoryBisr wrapper
//
//  Done  creation of DftSpecification(blockA,rtl)
//
/DftSpecification(blockA,rtl)
```

At the end of the command execution, Tessent MemoryBIST reports the generated name of the current DftSpecification. In this case, the name is "/DftSpecification(blockA,rtl)". Note that your transcript may look different depending on your settings and design requirements.

Tip

ℹ Use the Tcl command 'set' to capture the returned reference to the DftSpecification generated by the 'create_dft_specification' command into a Tcl variable. This way, you can use the variable instead of writing the entire name of the DftSpecification in subsequent commands.

The create_dft_specification command creates a specification in memory for you to report, introspect, and edit. No Memory BIST RTL has been created and your design has not yet been modified. This happens later through the process_dft_specification command.

Review and Basic Edits of the DftSpecification

Now that you have a DftSpecification, you want to see what the outcome is of Tessent MemoryBIST's analysis of your setup information and constraints. Under some circumstances, you may want to edit the DftSpecification. This should be rare and not necessary for most memory BIST implementations. The examples below show how to complete various tasks, ranging from viewing the DftSpecification to making changes.

Examples

Example 1: Reporting the Entire DftSpecification

```
ANALYSIS> set spec [create_dft_specific ]
//  sub-command: create_dft_specification
//
//  Begin creation of DftSpecification(blockA,rtl)
//    Creation of RtlCells wrapper
//    Creation of IjtagNetwork wrapper
//    Creation of MemoryBist wrapper
//    Creation of MemoryBisr wrapper
//
//  Done  creation of DftSpecification(blockA,rtl)
//
/DftSpecification(blockA,rtl)
ANALYSIS> report_config_data $spec

DftSpecification(blockA,rtl) {
  IjtagNetwork {
    HostScanInterface(ijtag) {
      Sib(sti) {
        Attributes {
          tessent_dft_function : scan_tested_instrument_host;
        }
        Sib(mbist) {
        }
      }
    }
  }
  MemoryBist {
    ijtag_host_interface : Sib(mbist);
    Controller(c1) {
      clock_domain_label : clka;
      Step {
        MemoryInterface(m1) {
          instance_name : blockA_l1_i1/blockA_l2_i1/mem1;
        }
        MemoryInterface(m2) {
          instance_name : blockA_l1_i1/blockA_l2_i1/mem2;
        }
        MemoryInterface(m3) {
          instance_name : blockA_l1_i1/blockA_l2_i1/mem3;
        }
      }
    }
[...]
```

Example 2: Reporting Parts of the DftSpecification

Usually, reporting the entire DftSpecification is too large. This example shows how to report only parts of it. You use the hierarchy in the DftSpecification, or in any other specification for that matter, similar to a hierarchical path name in the design. The example below reports the contents of memory BIST Controller(c1) of Example 1 above.

```
ANALYSIS> report_config_data $spec/MemoryBist/Controller(c1)

Controller(c1) {
  clock_domain_label : clka;
  Step {
    MemoryInterface(m1) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem1;
    }
    MemoryInterface(m2) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem2;
    }
    MemoryInterface(m3) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem3;
    }
  }
  Step {
    MemoryInterface(m4) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem4;
    }
    MemoryInterface(m5) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem5;
    }
  }
  Step {
    MemoryInterface(m6) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem6;
    }
  }
}
```

Example 3: Reporting all Default Values

By default, Tessent MemoryBIST only reports entries in the DftSpecification that differ from the defaults setting. This is done so you can easily see only what has changed, what is different. If you want to see the entire set of wrappers, options, and values for your DftSpecification, you use report_config_data -show_unspecified.

```
ANALYSIS> report_config_data $spec/MemoryBist/Controller(c1) -
show_unspecified

Controller(c1) {
  parent_instance : "";
  leaf_instance_name : "";
  clock_domain_label : clka;
  clock_period : "";
  AdvancedOptions {
    algorithm : from_library;
    operation_set : from_library;
    extra_algorithms : "";
    extra_operation_sets : "";
    incremental_test_mode : off;
[...]
```

Example 4: Introspecting and Changing DftSpecification Values

This example shows you the steps to edit a specific entry in the DftSpecification. Because you edit the information in-memory, a subsequent execution of the modified DftSpecification with the process_dft_specification command considers these changes.

> ⚠️ **Caution**
>
> Editing the DftSpecification on this level is usually not necessary and can result in unintended side effects, up to and including an invalid DftSpecification or BIST solution.

In this example you reduce the StopOnError failure limit of controller c1 from 4096 to 2048.

```
ANALYSIS> report_config_data $spec/MemoryBist/Controller(c1)/
DiagnosisOptions -show_unspecified

DiagnosisOptions {
  comparator_selection_mux : on;
  StopOnErrorOptions {
    failure_limit : 4096;
  }
}
ANALYSIS> get_config_value $spec/MemoryBist/Controller(c1)/
DiagnosisOptions/StopOnErrorOptions/failure_limit
4096
ANALYSIS> set_config_value $spec/MemoryBist/Controller(c1)/
DiagnosisOptions/StopOnErrorOptions/failure_limit 2048
ANALYSIS> get_config_value $spec/MemoryBist/Controller(c1)/
DiagnosisOptions/StopOnErrorOptions/failure_limit
2048
ANALYSIS> report_config_data $spec/MemoryBist/Controller(c1)/
DiagnosisOptions -show_unspecified

DiagnosisOptions {
  comparator_selection_mux : on;
  StopOnErrorOptions {
    failure_limit : 2048;
  }
}
```

Re-Creating the DftSpecification

Some information is only available for report or introspection in the DftSpecification. Other information only influences the DFT solution, like memory partitioning, and only its result can be seen in the DftSpecification. Changing that information requires significant editing of the DftSpecification. The easier way is to change the incoming data and just re-create the DftSpecification.

In this example, you want to move one memory instance, blockA_l1_i1/blockA_l2_i1/mem1, to a separate controller. The easiest way is to place this instance into its own physical cluster.

Example: Changing Physical Clustering Information and Re-Creating the DftSpecification

The example has several steps. At first you report parts of the DftSpecification, as well as report of the memory instance options. Then you override the physical cluster information for one instance. The next reporting shows that the DftSpecification remains unchanged, because it has not been re-created. Your change of the physical cluster information becomes only represented in the DftSpecification after the create_dft_specification -replace command. The subsequent reporting shows this.

```
ANALYSIS> report_config_data $spec/MemoryBist/Controller(c1)

Controller(c1) {
  clock_domain_label : clka;
  Step {
    MemoryInterface(m1) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem1;
    }
    MemoryInterface(m2) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem2;
    }
    MemoryInterface(m3) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem3;
    }
  }
  Step {
    MemoryInterface(m4) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem4;
    }
    MemoryInterface(m5) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem5;
    }
  }
  Step {
    MemoryInterface(m6) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem6;
    }
  }
}
ANALYSIS> report_memory_instances -limit 2
//
// Memory Instance: 'blockA_l1_i1/blockA_l2_i1/mem1'
// ----------------------------------------------
// bist_data_in_pipelining              : off
// physical_cluster_override            :
// power_domain_island                  : pd_A_1
// test_clock_override                  :
// use_in_memory_bist_dft_specification : on
// use_in_memory_bisr_dft_specification : off
// parent_cluster_module                :
//
// Memory Instance: 'blockA_l1_i1/blockA_l2_i1/mem2'
// ----------------------------------------------
// bist_data_in_pipelining              : off
// physical_cluster_override            :
// power_domain_island                  : pd_A_1
// test_clock_override                  :
// use_in_memory_bist_dft_specification : on
// use_in_memory_bisr_dft_specification : off
// parent_cluster_module                :
//
// Reached limit of 2, skipping remaining 16 instances.
ANALYSIS> set_memory_instance_options blockA_l1_i1/blockA_l2_i1/mem1
-physical_cluster_override MyCluster
ANALYSIS> report_memory_instances -limit 2
//
// Memory Instance: 'blockA_l1_i1/blockA_l2_i1/mem1'
// ----------------------------------------------
```

```
// bist_data_in_pipelining              : off
// physical_cluster_override            : MyCluster
// power_domain_island                  : pd_A_1
// test_clock_override                  :
// use_in_memory_bist_dft_specification : on
// use_in_memory_bisr_dft_specification : off
// parent_cluster_module                :
//
// Memory Instance: 'blockA_l1_i1/blockA_l2_i1/mem2'
// ----------------------------------------------
// bist_data_in_pipelining              : off
// physical_cluster_override            :
// power_domain_island                  : pd_A_1
// test_clock_override                  :
// use_in_memory_bist_dft_specification : on
// use_in_memory_bisr_dft_specification : off
// parent_cluster_module                :
//
// Reached limit of 2, skipping remaining 16 instances.
ANALYSIS> report_config_data $spec/MemoryBist/Controller(c1)

Controller(c1) {
  clock_domain_label : clka;
  Step {
    MemoryInterface(m1) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem1;
    }
    MemoryInterface(m2) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem2;
    }
    MemoryInterface(m3) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem3;
    }
  }
  Step {
    MemoryInterface(m4) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem4;
    }
    MemoryInterface(m5) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem5;
    }
  }
  Step {
    MemoryInterface(m6) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem6;
    }
  }
}
ANALYSIS> set spec [create_dft_spec -replace]
//   sub-command: create_dft_specification -replace
//
//   Begin creation of DftSpecification(blockA,rtl)
//     Creation of RtlCells wrapper
//     Creation of IjtagNetwork wrapper
//     Creation of MemoryBist wrapper
//     Creation of MemoryBisr wrapper
//
//   Done  creation of DftSpecification(blockA,rtl)
```

```
//
/DftSpecification(blockA,rtl)
ANALYSIS> report_config_data $spec/MemoryBist/Controller(c1)

Controller(c1) {
  clock_domain_label : clka;
  Step {
    MemoryInterface(m1) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem1;
    }
  }
}
ANALYSIS> report_config_data $spec/MemoryBist/Controller(c2)

Controller(c2) {
  clock_domain_label : clka;
  Step {
    MemoryInterface(m1) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem2;
    }
    MemoryInterface(m2) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem3;
    }
  }
  Step {
    MemoryInterface(m3) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem4;
    }
    MemoryInterface(m4) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem5;
    }
  }
  Step {
    MemoryInterface(m5) {
      instance_name : blockA_l1_i1/blockA_l2_i1/mem6;
    }
  }
}
ANALYSIS>
```

Validating the DftSpecification

After editing the DftSpecification, especially when done outside of Tessent MemoryBIST, you may have introduced errors in the DftSpecification. These errors could be as simple as a typo in a memory instance path name, or as severe as placing incompatible memories into the same test step of the same controller. Before proceeding with the memory BIST hardware generation and insertion, you want to validate that the current DftSpecification is error free and consistent.

To do this, you use the -validate_only switch of the process_dft_specification command. Because you only validate the specification, no RTL has been created and your design remains unchanged.

```
ANALYSIS> process_dft_specification -validate_only
//
//  Begin validation of /DftSpecification(blockA,rtl)
//     Validation of IjtagNetwork
//     Validation of MemoryBist
//     Validation of MemoryBisr
//
//  Done  validation of DftSpecification(blockA,rtl)
//
/DftSpecification(blockA,rtl)
```

Additional Editing Options of the DftSpecification

The examples in the prior sections show basic DftSpecification editing, which is usually all that is needed. In this section, you go beyond these basic editing and changing of values in existing wrappers.

The first part in this section introduces the creation, interactive editing, and validation of a DftSpecification using the graphical interface. The second part shortly introduces the rich command-based editing capability of the DftSpecification enabled by Tessent MemoryBIST.

Editing the DftSpecification With the Config Data Browser in Tessent Visualizer

The Config Data Browser in Tessent Visualizer is a convenient way to learn about the DftSpecification. At each step and data entry point it offers to you only valid entries.

Through the following sequence of examples, you learn how to invoke the Config Data Browser, expand the hierarchy of the DftSpecification, add a new specification wrapper (a test step in this case) and use the cut-and-paste feature of the GUI to move a memory testing instance from its original step to the newly created step.

These examples only show you a small sample of the capabilities of the Config Data Browser.

Examples

Example 1: Invoking the Config Data Browser for a DftSpecification

Upon execution of the display_specification command, the Config Data Browser starts with the data from your last session (if any). The entries displayed in the Config Data Browser for your design might differ from the ones shown in Figure 3-7, but the general objects should be there.

In the left pane, you see the (initially) collapsed hierarchy of the DftSpecification. Wrappers that have not been specified are grayed out. In the right pane, you see the properties and values for the selected wrapper node. The fields are automatically populated based on the DftSpecification and its respective default value setting.

Figure 3-7. Config Data Browser Invocation
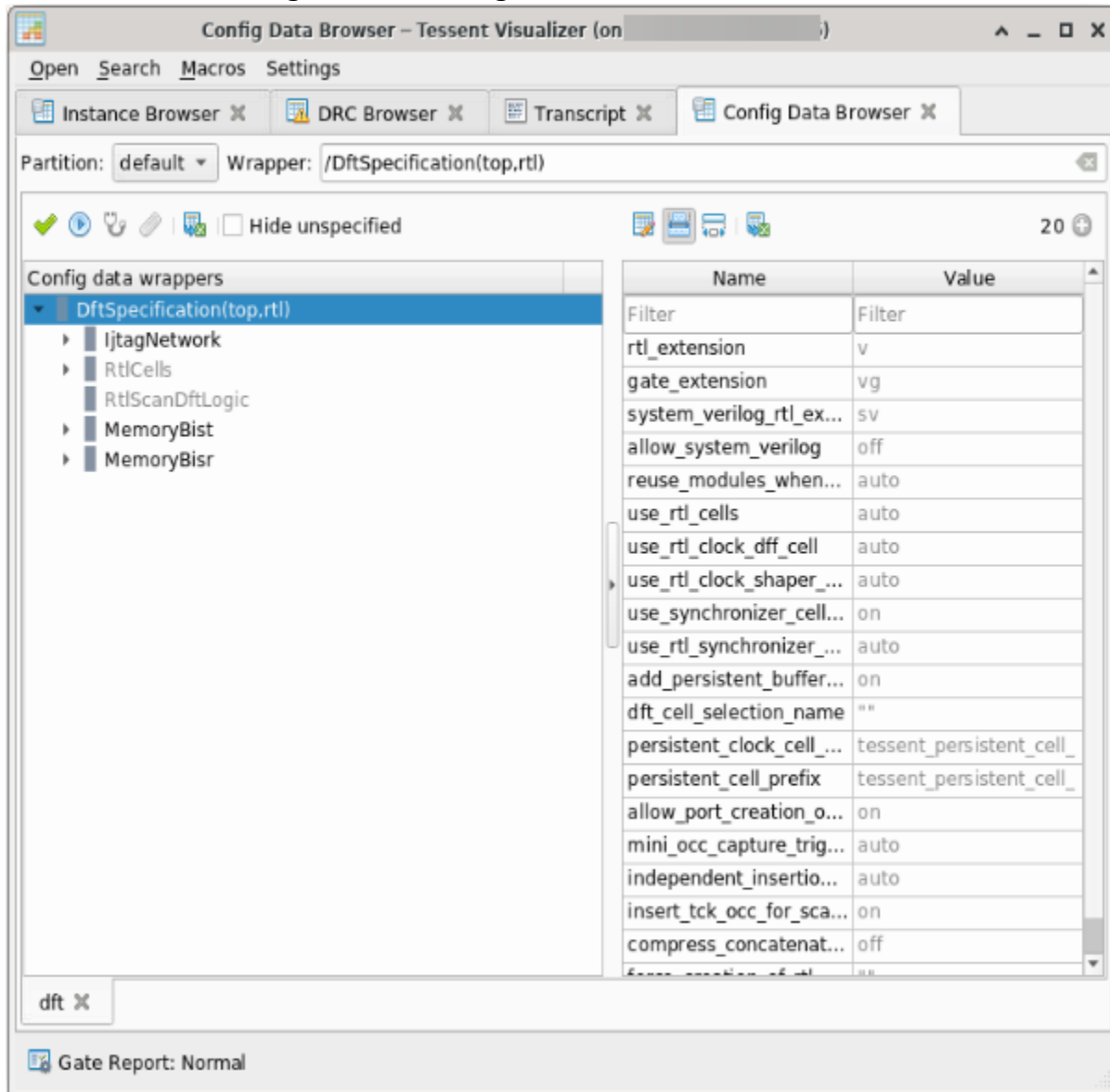
Example 2: Navigating the Hierarchy of the DftSpecification

You navigate through the hierarchy of the DftSpecification by clicking on the small arrow symbols next to the specification wrappers. This expands the wrapper, showing you additional wrappers until you come to a leaf entry. Figure 3-8 shows you the expansion down to the memory interface m3 of step 0 of the memory BIST controller (c1).

Figure 3-8. Config Data Browser Hierarchy and Values
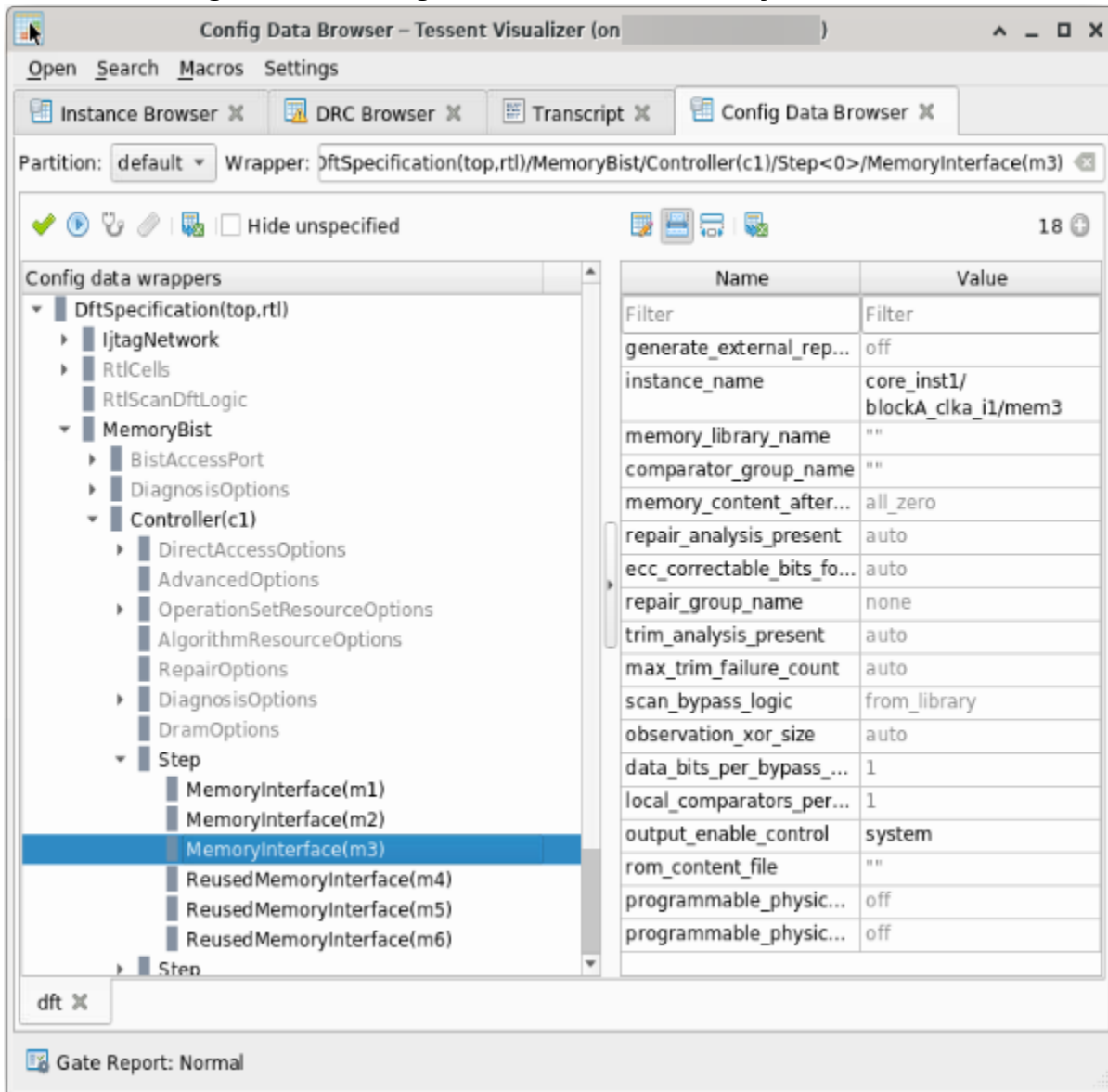
Example 3: Adding a new Specification Wrapper

For controller c1, you want to add another test step. As shown in Figure 3-9, when you right-click the Controller(c1) wrapper, a menu displays. Select "Add wrapper as a child." A second dialog box is displayed as shown in Figure 3-10. Choose "Step" from the dropdown menu and click "Apply.". You now see that a new step has been added to the Controller(c1) wrapper. You can move the new wrapper to its position in the sequence of all steps of the controller, or define other parameters of this step as shown in the right pane of the window.

Figure 3-9. Creating a New Test Step for Controller(c1)
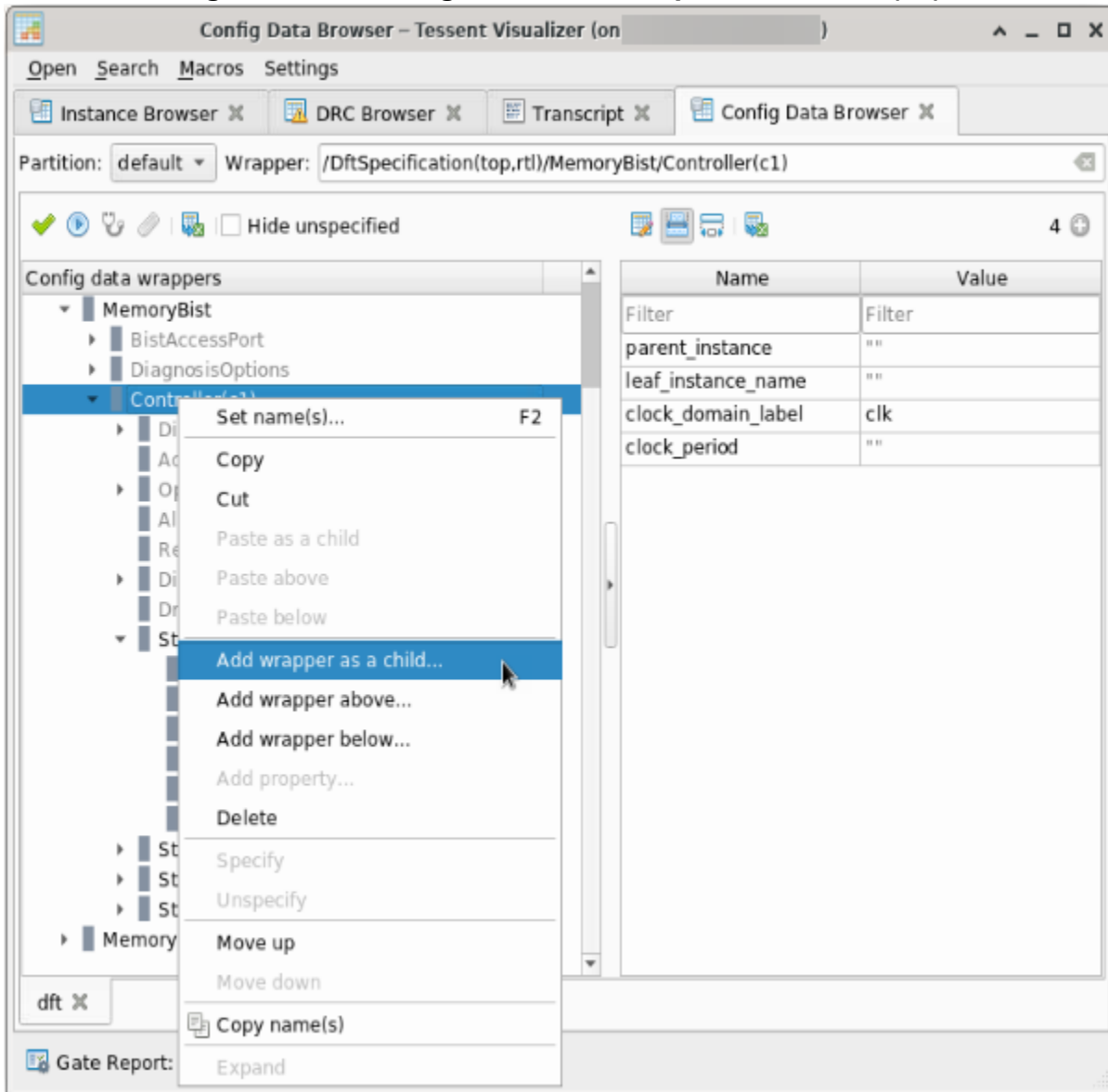
Figure 3-10. Adding a Step
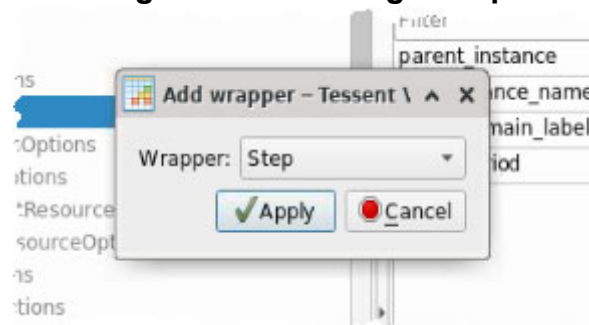
Example 4: Using Cut-and-Paste

Next you need to populate the new step wrapper with a MemoryInterface. In this example, you use the cut-and-paste feature of the Config Data Browser to move the MemoryInterface(3) from the first Step to the new Step. To do this, you right-click MemoryInterface(3), choose "cut," move the mouse cursor over the new Step, right-click again, and choose "paste.". You have just dropped the MemoryInterface(m3) to its new step and position in the memory BIST test execution. Figure 3-11 shows you the final picture of your DftSpecification editing.

Figure 3-11. MemoryInterface(m3) Cut-and-Paste Result
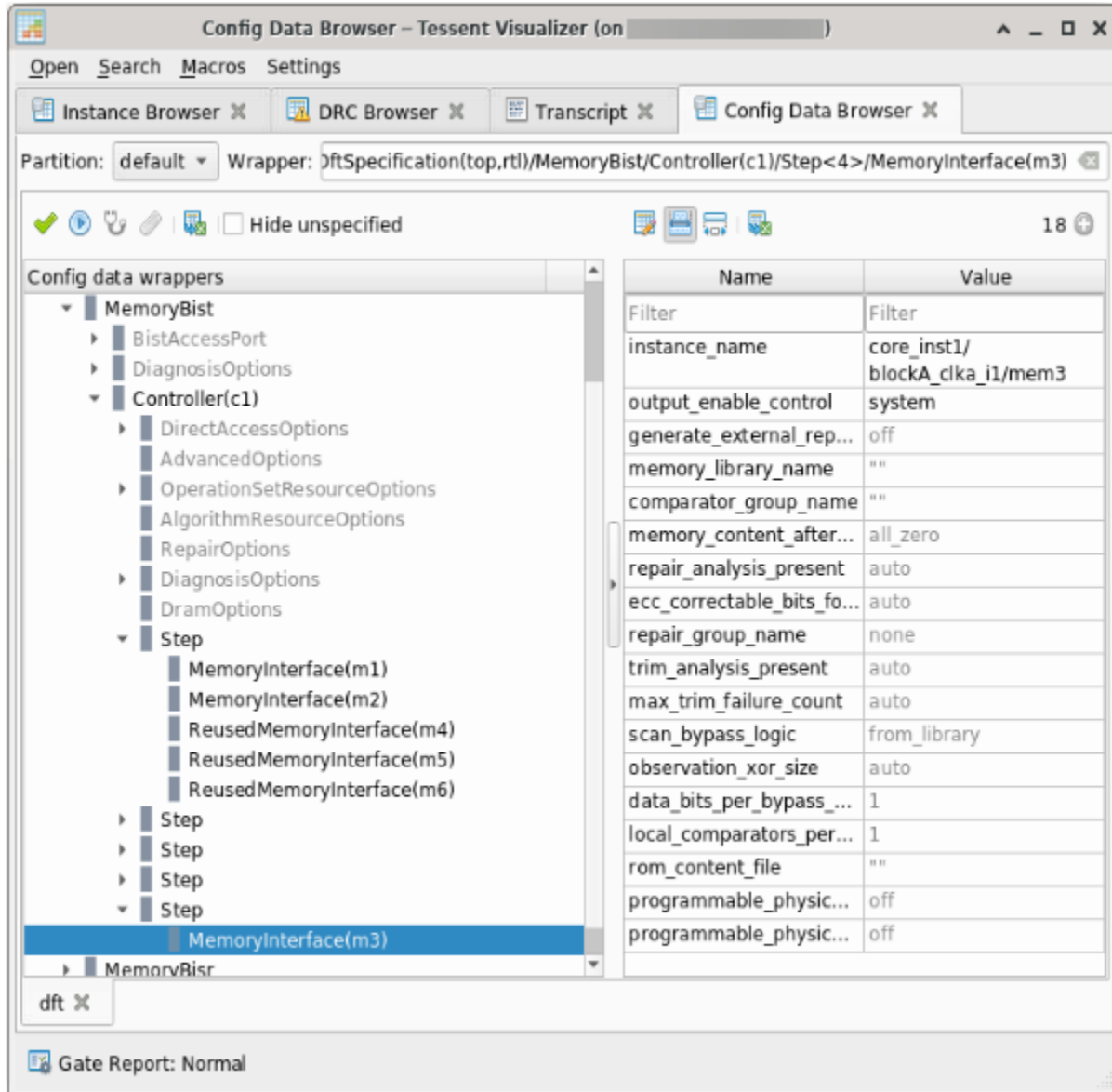
Validating a DftSpecification With the Config Data Browser in Tessent Visualizer

You should perform a validation step for the DftSpecification you edited either in the Config Data Browser, through commands, or manually. The advantage of running the validation step with the Config Data Browser is the color-coding of the DftSpecification hierarchical path down to the point of error. This path is highlighted in orange and red. Together with the error messages, you should be able to identify the problem quickly and implement the required fix. You can select the "Validate" button at any time.

Figure 3-12 shows the color-coding of the path through the DftSpecification to the error point and a portion of the Tessent Shell window showing the results of the validation, as well as the "Validate" button (circled in red). In addition to the transcript display, the error is available on a problematic wrapper tree node or property in the table, and you can read it in a tooltip with a mouse hover.

Figure 3-12. DftSpecification Validation Error in the Config Data Browser
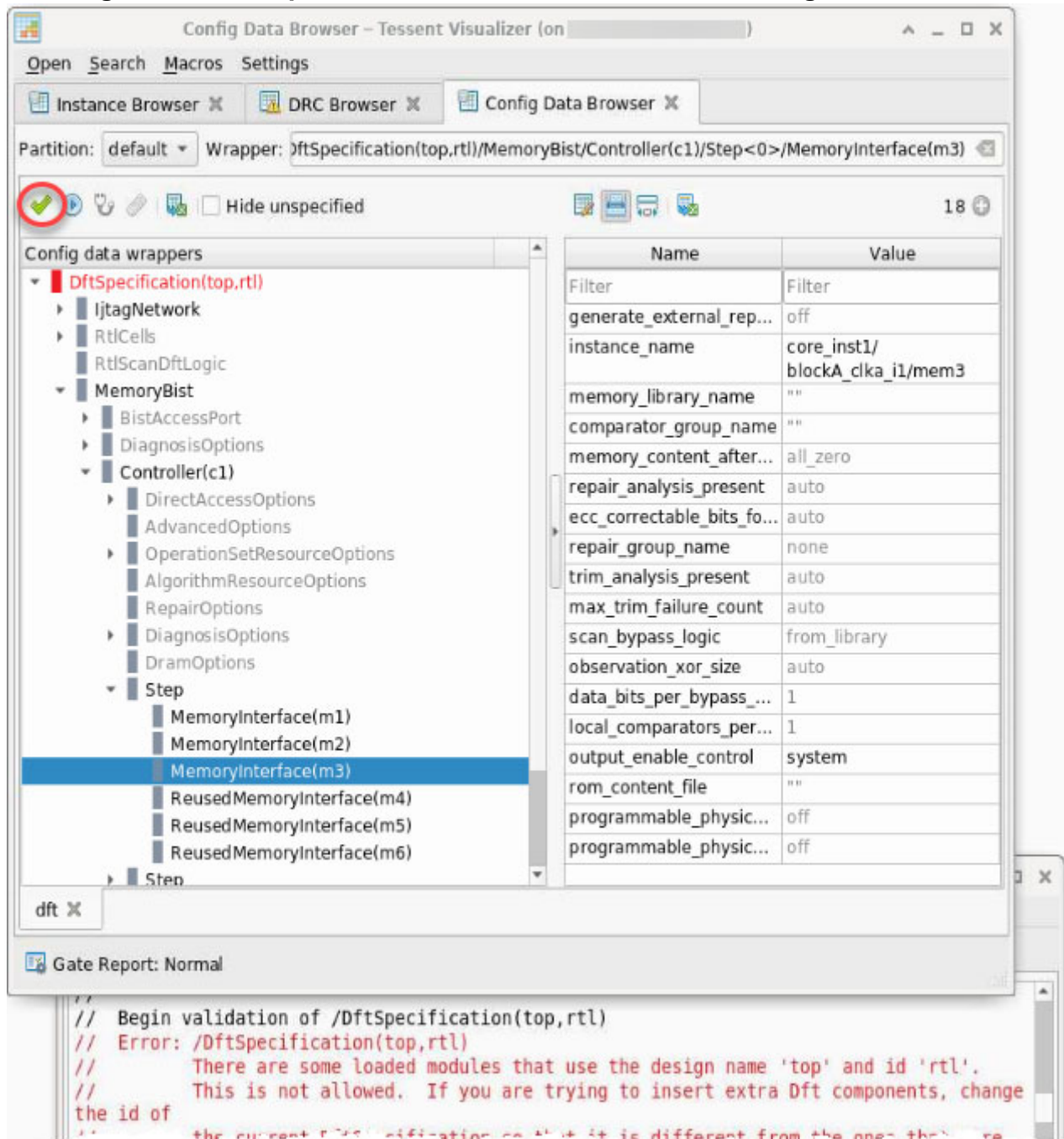
Editing the DftSpecification Using Shell Commands

Tessent MemoryBIST offers powerful and rich commands to create a DftSpecification from scratch, edit an existing one, merge templates into your specification and much more.

There is an entire family of commands documented in the *Tessent Shell Reference Manual*, each having usage examples. You are encouraged to learn about these commands. The related topics listed below provides a good starting point.

Related Topics

add_config_element [Tessent Shell Reference Manual]

delete_config_element [Tessent Shell Reference Manual]

get_config_elements [Tessent Shell Reference Manual]

get_config_value [Tessent Shell Reference Manual]

move_config_element [Tessent Shell Reference Manual]

set_config_value [Tessent Shell Reference Manual]

Processing the DftSpecification

After the successful creation of the DftSpecification, the examples shown in this section describe the steps you take to generate the memory BIST and repair logic, and then insert them into your design.

Examples

Example 1: Validating the DftSpecification

If you have not done so, it is recommended to perform a validation of the DftSpecification prior to executing the RTL generation and insertion. To do this, you use the -validate_only switch of the process_dft_specification command. Because you only validate the specification, no RTL has been created and your design remains unchanged.

```
ANALYSIS> process_dft_specification -validate_only
//
//  Begin validation of /DftSpecification(blockA,rtl)
//    Validation of IjtagNetwork
//    Validation of MemoryBist
//    Validation of MemoryBisr
//
//  Done  validation of DftSpecification(blockA,rtl)
//
/DftSpecification(blockA,rtl)
```

Example 2: Executing the DftSpecification

Once the validation has passed successfully, you should not have any issues executing the current DftSpecification. This step generates all Tessent MemoryBIST RTL, including any BISR and BIRA logic, as well as RTL cells, as needed. Also your design is modified, inserting the BIST logic into your design.

___ **Note** ___

You have the option to process the DftSpecification but not have the BIST logic inserted into your design. You do this by using the -no_insertion option of the process_dft_specification command. Tessent MemoryBIST then generates all of its RTL as usual, but does not modify your design.

The transcript below is significantly shortened. You see in the transcript where Tessent MemoryBIST saves all generated and modified files. All generated RTL, ICL, and PDL goes into the 'instruments' directory of the current TSDB (Tessent Shell Data Base), further subdivided by instrument. The TSDB has the default location and name of './tsdb_outdir'. All modified design files are saved in the 'dft_inserted_design' directory of the TSDB.

```
ANALYSIS> process_dft_specification
//
//   Begin processing of /DftSpecification(blockA,rtl)
//     --- IP generation phase ---
//     Validation of IjtagNetwork
//     Validation of MemoryBist
//     Validation of MemoryBisr
//     Processing of RtlCells
//       Generating Verilog RTL Cells
//         Verilog RTL : ./tsdb_outdir/instruments/
blockA_rtl_cells.instrument/blockA_rtl_tessent_and2.v
[...]

//     Processing of IjtagNetwork
//       Generating design files for IJTAG SIB module
blockA_rtl_tessent_sib_1
//         Verilog RTL : ./tsdb_outdir/instruments/
blockA_rtl_ijtag.instrument/blockA_rtl_tessent_sib_1.v
//         IJTAG ICL   : ./tsdb_outdir/instruments/
blockA_rtl_ijtag.instrument/blockA_rtl_tessent_sib_1.icl
[...]
//     Processing of MemoryBist
//       Generating the IJTAG ICL for the memories.
//       Generating design files for MemoryBist Controller(c1)
//       Generating design files for MemoryBist Controller(c2)
[...]
//     Processing of MemoryBisr
//       Generating design files for BISR module
blockA_rtl_tessent_mbisr_register_SYNC_1RW_32x16_RC_BISR
//         Verilog RTL : ./tsdb_outdir/instruments/
blockA_rtl_mbisr.instrument/
blockA_rtl_tessent_mbisr_register_SYNC_1RW_32x16_RC_BISR.v
//         IJTAG ICL   : ./tsdb_outdir/instruments/
blockA_rtl_mbisr.instrument/
blockA_rtl_tessent_mbisr_register_SYNC_1RW_32x16_RC_BISR.icl
//     --- Instrument insertion phase ---
//     Inserting instruments of type 'ijtag'
//     Inserting instruments of type 'memory_bist'
//     Inserting instruments of type 'memory_bisr'
//
//     Writing out modified source design in ./tsdb_outdir/
dft_inserted_designs/blockA_rtl.dft_inserted_design
//     Writing out specification in ./tsdb_outdir/dft_inserted_designs/
blockA_rtl.dft_spec
//
//   Done  processing of DftSpecification(blockA,rtl)
//
/DftSpecification(blockA,rtl)
INSERTION>
```

Chapter 4
Creating and Verifying Test Patterns

This chapter describes the process to create and verify test patterns for MemoryBIST controllers.

Pattern Generation for TS-MBIST Insertions

Tessent MemoryBIST uses a similar flow for generating patterns at the block, core, or top level.

You typically generate only Verilog testbenches and not manufacturing test patterns at the block or core level, because that portion of the circuitry normally gets instantiated in a larger top-level design.

At the top level, you generate Verilog testbenches before signing-off the design. You then create Automated Test Equipment (ATE) test patterns, to apply as manufacturing tests on silicon devices.

In bottom-up flows, MBIST-inserted lower-level blocks or cores are fully verified in a stand-alone fashion before being integrated into a larger design. This hierarchical DFT method enables completing MBIST insertion in smaller design portions, even though the rest of the design may not be ready yet. Because MBIST only requires valid clocks and a low-speed serial test access to run, verifying a block or core after it has been integrated into the full (top) design is greatly simplified.

In top-down flows, MBIST insertion is done across the entire design at once. This methodology requires fewer steps overall than a hierarchical DFT insertion, however the MBIST insertion has to be verified across the entire design before it can be signed-off. This process consequently takes longer to perform and requires more computing resources than a hierarchical approach. This full-chip verification task typically happens at a very critical time (for example, when the chip is essentially completed and just about to tape out), so unexpected delays may impact the design schedule.

The following figure illustrates the pattern generation for MemoryBIST controllers that are inserted in a core or top design.

Figure 4-1. Pattern Generation
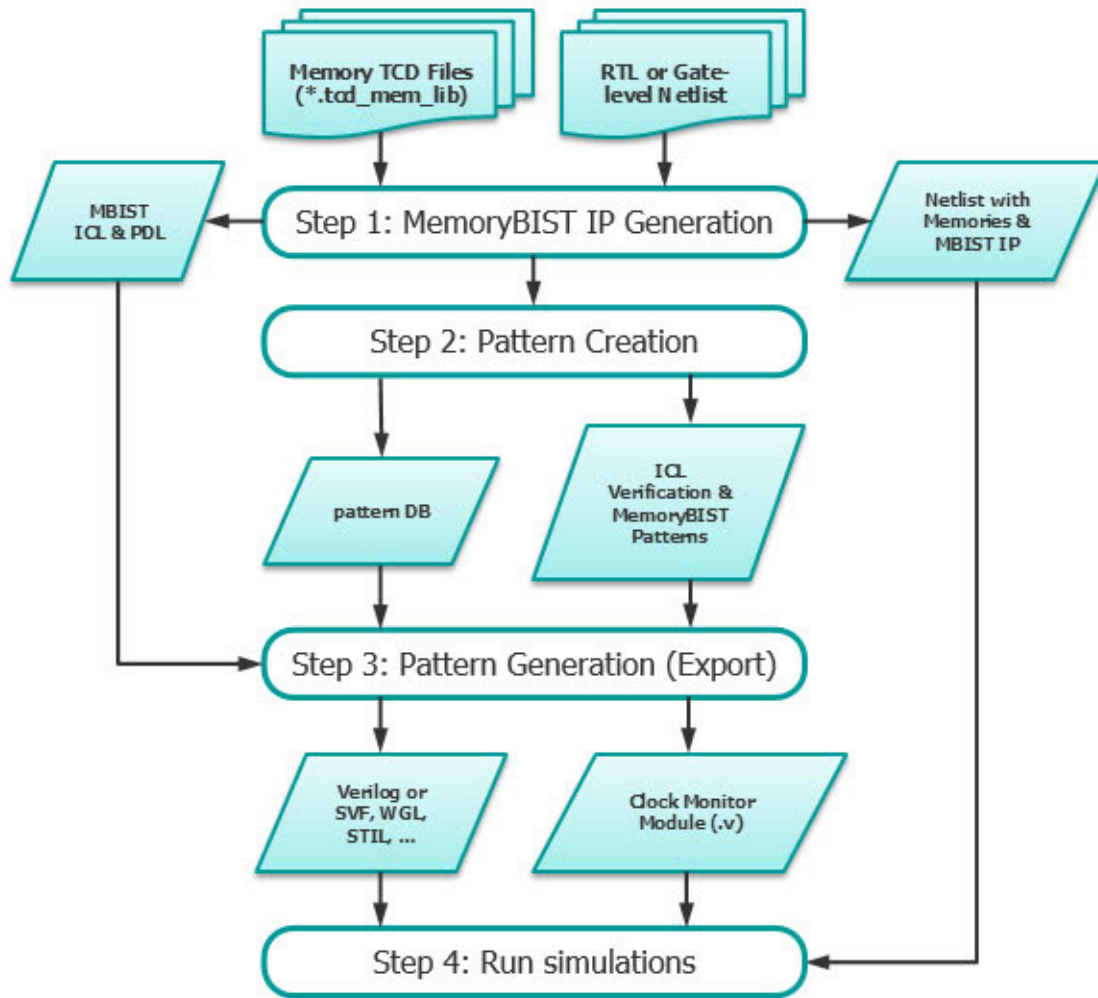
You generate the following:

- For the bottom-up flow, core-level Verilog testbenches

- For the top-down flow, top-level manufacturing test patterns and Verilog testbenches

The following sections describe how to create Verilog testbench simulation patterns and manufacturing test patterns. For manufacturing test patterns, the tool supports all test pattern formats currently supported for ATPG.

Creating Simulation Testbench Patterns

Use this procedure to generate Verilog simulation testbench patterns that can be applied at the block, core, or top level.

Signoff, or simulation testbench patterns, are generated by default. The test patterns only target controllers inserted in the current design. Controllers in lower level physical blocks can be included when the property simulate_instruments_in_lower_physical_instances is set to on in the DefaultsSpecification. All controllers are run in parallel. Additionally, for controllers with RAMs you can limit testing to address corners by setting the patterns specification property reduced_address_count to on.

Prerequisites

The required inputs for this step of the flow that you specify from the tool prompt or within the pattern generating dofile are as follows:

- The TSDB (the tsdb_outdir) of a completed block, core or top-level design.

- Extracted ICL for the current design portion (normally found in the TSDB).

- Exported PDL with procedures for the current design (also found in the TSDB).

- An RTL or gate-level netlist of the current design for sign-off simulations.

The following procedure assumes the current design already went through MBIST insertion in a separate Tessent Shell session. If you just completed MBIST and are still within the same Tessent Shell session, skip step 1 below and go directly to step 2.

Procedure

1. From a shell, invoke Tessent Shell using the following syntax:

   ```
   % tessent -shell
   ```

2. Set the tool context to IJTAG patterns mode using the set_context command as follows:

   ```
   SETUP> set_context patterns -ijtag
   ```

3. Open the TSDB with the open_tsdb command, if it is not already open (which would be the case if you just completed MBIST insertion within the very same Tessent Shell session). For example:

   ```
   SETUP> open_tsdb tsdb_outdir
   ```

4. Unless it is already in memory, read the current design's extracted ICL and TCD with the read_design command. For example:

   ```
   SETUP> read_design blockA -design_identifier rtl -no_hdl
   ```

5. Set the current design with the set_current_design command. For example:

   ```
   SETUP> set_current_design blockA
   ```

The TSDB is analyzed and all inserted DFT logic (including MBIST controllers) are identified.

6. Create a pattern specification using the create_patterns_specification command. The example below sets a variable with the identification for the created specification, which can be used in modifying the specification if needed.

```
SETUP> set pat_spec [create_patterns_specification]
```

Memory BIST patterns are generated at this point.

7. If wanted, follow this step to set reduced_address_count to off for testing the full address range of memories. Otherwise, continue to the next step. The process outlined can be modified to configure other properties within the pattern specification as needed.

   a. Specify the path to the wrapper containing the property. For example:

   ```
   SETUP>set wrap [get_config_elements -in_wrapper ${pat_spec} \
           Patterns(MemoryBist_P1)/TestStep/MemoryBist]
   ```

   The variable $wrap now contains the full path.

   b. Set the reduced_address_count property to the off setting. For example:

   ```
   SETUP>set_config_value reduced_address_count \
           -in_wrapper ${wrap} off
   ```

   The patterns specification is now updated with the wanted property setting.

   c. Optionally, confirm the setting by inspecting the patterns specification:

   ```
   SETUP>report_config_data $pat_spec
   ```

   The patterns specification is displayed for examination.

8. Process the pattern specification:

   ```
   SETUP> process_patterns_specification
   ```

   _____ **Note** _____
   The create_patterns_specification command always includes the Parallel Retention
   Test (PRT) pattern as part of the default signoff and manufacturing patterns
   specifications. It is highly recommended to simulate the PRT pattern during signoff
   verification even if you are not planning to use it during manufacturing test. The
   structure of the PRT pattern provides verification that all memories are accessible. You
   can modify the PRT pattern to operate with a reduced address space and zero retention
   time to reduce simulation time.
   _____

9. Point to the simulation library sources so all design files can be found. For example:

   ```
   SETUP> set_simulation_library_sources -y ./memlibs -extensions { v }
   ```

10. Simulate the memory BIST testbenches with the following command:

```
SETUP> run_testbench_simulations
```

11. The above simulation can be monitored or checked with the following command:

```
SETUP> check_testbench_simulations
```

Results

If you are using the Top-Down Flow, you finished all necessary steps in this flow.

If you are using the Bottom-Up Flow, you are ready to perform the Top-Level ICL Network Integration.

Examples

The following example dofile opens the TSDB for blockA and generates its memory BIST testbenches. The pattern specification gets saved into a variable named $spec and could be reported on-screen if needed, by uncommenting the report_config_data line:

```
set_context patterns -ijtag

open_tsdb tsdb_outdir
read_design blockA -design_identifier rtl -no_hdl
set_current_design blockA

set spec [create_patterns_spec]
#report_config_data $spec
process_patterns_specification

run_testbench_simulations
check_testbench_simulations

exit
```

Related Topics

Patterns [Tessent Shell Reference Manual]

TestStep [Tessent Shell Reference Manual]

MemoryBist [Tessent Shell Reference Manual]

write_config_data [Tessent Shell Reference Manual]

read_config_data [Tessent Shell Reference Manual]

Creating Manufacturing Test Patterns

Use this procedure to generate manufacturing test patterns that can be applied at the top level.

Generated patterns are grouped based on the number of asynchronous ATE clocks that can be active within one pattern. The number is based on the DefaultsSpecification max_async_clock_sources property that defaults to unlimited, placing all memory BIST

controllers inside of a single pattern, operating in parallel and potentially belonging to different asynchronous frequency groups.

Generated patterns can be further split into multiple test step or procedure step pattern files, to enable test program events such as VDD bumping or PMU measurements being inserted in the middle of the patterns. Refer to the AdvancedOptions/split_patterns_file property descriptions in the TestStep and ProcedureStep wrappers for more information about patterns file splitting.

Manufacturing test patterns always perform tests across the full memory address space. The patterns specification property reduced_address_count defaults to off, and unlike signoff patterns, it is not set to on when a controller is testing RAMs only.

Manufacturing pattern files can be generated in WGL, STIL, STIL2005, TITD, FJTDL, MITDL, TSTL2 and SVF formats. The format you want can be specified with the PatternsSpecification manufacturing_patterns_formats property. When unspecified, the STIL format is generated. For a description of these formats, see the description of the format_switch argument of the write_patterns command.

Prerequisites

The required inputs for this step of the flow that you specify from the tool prompt or within the pattern generating dofile are as follows:

- The TSDB (the tsdb_outdir) of a completed top-level design.

- Extracted ICL for the current design (normally found in the TSDB).

- Exported PDL with procedures for the current design (also found in the TSDB).

The following procedure assumes the current design already went through MBIST insertion in a separate Tessent Shell session. If you just completed MBIST and are still within the same Tessent Shell session, skip step 1 below and go directly to step 2.

Procedure

1. From a shell, invoke Tessent Shell using the following syntax:

   ```
   % tessent -shell
   ```

2. Set the tool context to IJTAG patterns mode using the set_context command as follows:

   ```
   SETUP> set_context patterns -ijtag
   ```

3. Open the TSDB with the open_tsdb command, if it is not already open (which would be the case if you just completed MBIST insertion within the very same Tessent Shell session). For example:

   ```
   SETUP> open_tsdb tsdb_outdir
   ```

4. Unless it is already in memory, read the current design's extracted ICL and TCD with the read_design command. For example:

```
SETUP> read_design blockA -design_identifier rtl -no_hdl
```

5. Set the current design with the set_current_design command. For example:

```
SETUP> set_current_design top
```

The TSDB is analyzed and all inserted DFT logic (including MBIST controllers) are identified.

6. Create a pattern specification using the create_patterns_specification command. The example below sets a variable with the identification for the created specification, which can be used in modifying the specification if needed.

```
SETUP> set pat_spec [create_patterns_specification manufacturing]
```

Specifying the manufacturing usage option automatically adds the usage : manufacturing_test property to the PatternsSpecification. Memory BIST patterns are generated at this point.

7. If you want, follow this step to specify the wanted pattern file format with the manufacturing_patterns_formats property in the PatternsSpecification. If the default STIL format is acceptable, continue to the next step. The process outlined can be modified to configure other properties within the pattern specification as needed.

   a. Using the variable set in Step 6, set the manufacturing_patterns_formats property to the wanted setting. For example, for WGL format:

```
SETUP>set_config_value manufacturing_patterns_formats \
        -in_wrapper ${pat_spec} Wgl
```

   The patterns specification is now updated with the new property setting.

   b. Optionally, confirm the setting by inspecting the patterns specification:

```
SETUP>report_config_data $pat_spec
```

   The patterns specification is displayed for examination.

8. Process the patterns specification with the process_patterns_specification command:

```
SETUP> process_patterns_specification
```

Results

The generated manufacturing pattern files are located by default in the tsdb_outdir/patterns/ *<design_name>_<design_id>*.patterns.manufacturing folder. The pattern filenames are *<pattern_name>.<format_ext>*.gz, where *<format_ext>* is determined by the format specified in the manufacturing_patterns_formats property of the PatternsSpecification.

Generated MBIST Verification Patterns

A pattern specification for a MBIST-only design typically consists of:

- One ICL verification pattern

- One or multiple Memory BIST patterns

ICL verification patterns are automatically created from extracted ICL of the current design. These patterns ensure the ICL description is correct and matches the design's implemented hardware. They check that all ICL-described Test Data Registers (TDRs) can be selected and have the expected length.

Memory BIST patterns exercise implemented MBIST controllers by clocking the design with appropriate clocks and instructing every BIST controller to launch a memory test. The generated testbenches (or patterns) thus run MBIST against design memories.

Example

An example block-level pattern specification is provided below:

```
PatternsSpecification(blockA,rtl,signoff) {
  Patterns(ICLNetwork) {
    ICLNetworkVerify(blockA) {
    }
  }
  Patterns(MemoryBist_P1) {
    ClockPeriods {
      CLK : 12.0ns;
    }
    TestStep(run_time_prog) {
      MemoryBist {
        run_mode : run_time_prog;
        reduced_address_count : on;
        Controller(blockA_rtl_tessent_mbist_c1_controller_inst) {
          DiagnosisOptions {
            compare_go : on;
            compare_go_id : on;
          }
          RepairOptions {
            check_repair_status : on;
          }
        }
      }
    }
  }
}
```

```
Patterns(MemoryBist_ParallelRetentionTest_P1) {
  ClockPeriods {
    CLK : 12.0ns;
  }
  TestStep(ParallelRetentionTest) {
    MemoryBist {
      run_mode : hw_default;
      parallel_retention_time : 0;
      reduced_address_count : on;
      Controller(blockA_rtl_tessent_mbist_c1_controller_inst) {
        parallel_retention_group : 1;
        DiagnosisOptions {
          compare_go_id : on;
        }
      }
    }
  }
}
```

The ClockPeriods wrapper indicates clock ports on the design, along with a clock period for each. Verilog testbenches model this clock as a clock generator for simulations). When creating manufacturing patterns for silicon the tool assumes this clock is provided by the tester.

A single memory BIST pattern file contains one or several TestStep wrapper(s). Test steps are run sequentially and perform specific memory BIST and BISR controllers actions.

Memory BIST controllers can run algorithms across the entire address space being tested. This verification step may be considered excessive when inserting MBIST into a small block or core, since memory TCD files were previously certified and validated. In such case a quicker simulation can be performed, using the reduced_address_count property in the MemoryBist wrapper.

Within a single TestStep wrapper, one or several memory BIST Controller wrapper(s) may be listed; they instruct an associated MBIST controller to perform a test. If multiple Controller wrappers are listed, they perform their tests concurrently.

Diagnosis options are controller-specific and can be specified using the DiagnosisOptions wrapper:

- If available, the MBIST controller's GO output can be compared on every clock cycle using the compare_go property. If a comparison mismatch is detected between expected and actual (observed) data, the GO output is asserted low.

- The compare_go_id property can shift out the individual data comparator bits after BIST completes. This information is typically used to identify the faulty data bit(s) that resulted in the expected vs. observed data comparison error.

> ___ **Tip** ___
> 🛈 **The generated pattern specification is expected to be correct by construction and adequate for most cases. Only edit it when needed.**

For example, you can save a pattern specification using the write_config_data command and then immediately read back a modified copy of this file, using the read_config_data command. However, care should be taken to ensure that both files stay relatively similar.

Chapter 5
Implementing and Verifying Memory Repair

This chapter explains the process of implementing and verifying Built-In Self-Repair (BISR) capability in the Tessent Shell MemoryBIST flow.

Overview of Memory Repair Capabilities

Repairable memories are popular in today's designs. However, the failure data collection, redundancy analysis, and memory repair access increases the design complexity. Furthermore, when a chip has multiple repairable memories, a solid infrastructure must be developed to integrate the repair analysis and fuse box programming at the chip level so that the memories can be repaired at chip power-up.

Types of Repair

Memory repair can be used to improve manufacturing yield of integrated circuits. Tessent MemoryBIST supports all types of memory repair: column repair, row repair, or a combination of both. Column repair includes replacing single columns or blocks of columns up to the full width of a column multiplexer. The full width case is often referred to as IO repair. Row repair includes replacing blocks, rows, or even a subset of a row, down to a single word.

Spare rows and columns are incorporated to the memory itself by the memory compiler. The memory repair characteristics must be described in the memory library file provided to the Tessent MemoryBIST tools. Memory providers usually generate this file automatically and are the preferred source because the physical implementation information in the memory model cannot be qualified until post-silicon verification.

Repair Steps

Memory repair is performed in two steps. The first step consists of analyzing failures reported by the memory BIST controller during test execution to determine if the memory is repairable and, if repairable, determining the values to be applied to the repair inputs of the memory.

Chip Level Repair Architecture

This section provides an overview of the chip level repair architecture.

The figure below shows the top-level architecture of the BISR chain and the BISR controller. A central fuse box is connected to a chip-level BISR controller. The central fuse box can be instantiated inside or outside the BISR controller module. When the fuse box is located outside

the BISR controller module, extra connections between the fuse box and the BISR controller are required.

Figure 5-1. Chip-Level Configuration With BISR Controller
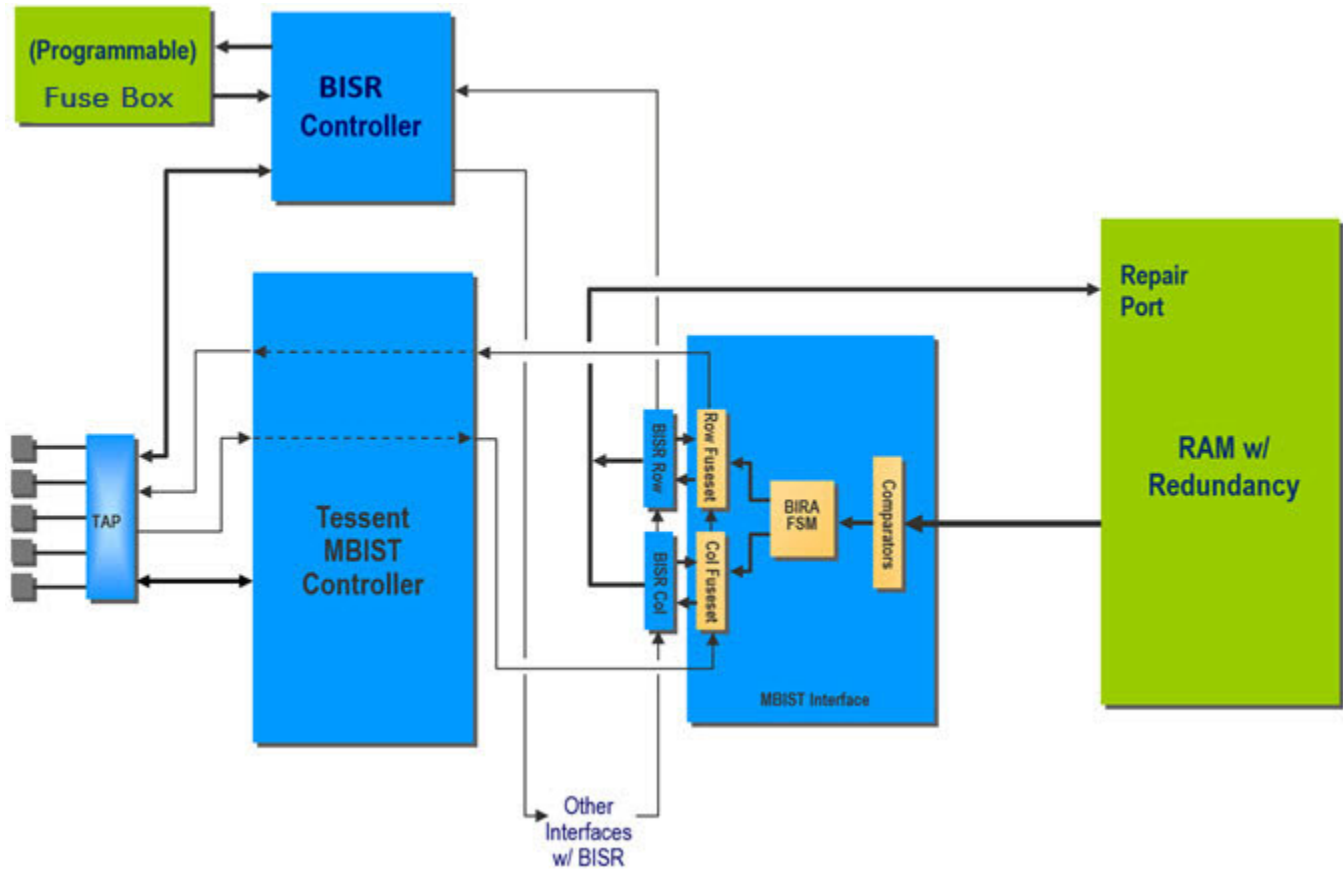
All repairable memories in the chip have a corresponding BISR register that holds a repair solution. All BISR registers in the chip are connected to form a chip-level BISR scan chain. The BISR chain is connected to a chip-level controller called a BISR controller. The BISR controller compresses the content of the BISR chain as it is scanned out of the BISR chain and writes the compressed data into the fuse box. The BISR controller can also decompress repair data from the fuse box and scan it in the BISR chain.

The BISR controller works in conjunction with a BIRA (Built-In Redundancy Analysis) module that provides the repair fuse values calculated from memory failure data.

_____ **Note** _____

The top-level architecture of the BISR chain and the BISR controller has two fundamental differences when the circuit contains power domains. These differences are covered in subsequent sections.

_____

Built-In Repair Analysis (BIRA)

Repairable memories implement spare or redundant rows or columns, enabling access to any faulty row or column to be redirected to a redundant element.

The existing memory BIST diagnostic capabilities, such as the Compare Status pins diagnostic approach and the Stop-On-Nth-Error diagnostic approach are not optimal solutions for the repair analysis of repairable memories. Each solution requires large test times or state-of-the-art high-speed memory testers. In addition, post execution analysis of the results must be done to determine if a memory is repairable and what repair, if any, is required.

To facilitate repairable memories in production testing environments, the Built-In Repair Analysis (BIRA) feature can be used to determine if a memory is repairable and the repair information based on the specified redundancy scheme.

This chapter outlines a solution for implementing and performing repair analysis in production test environments utilizing memory BIST. The repair analysis is performed during the execution of the memory BIST controller, and the repair results are scanned out of the controller after execution.

Table 5-1 shows the location of the BIRA circuit. Usually the BIRA circuitry is in the same location as the comparators. That is, BIRA is located in the BIST controller if DftSpecification/ MemoryBist/Controller/Step/comparator_location is set to shared_in_controller and is located in the memory interface if comparator_location is set to per_interface. The one exception is that it is also possible to insert the row repair BIRA circuitry in the controller even if comparator_location is set to per_interface. This enables minimizing area by maximizing sharing of address pipeline registers required by the BIRA logic. This is the default option. The location of the Row BIRA logic can be forced to be in the memory interface when comparator_location is set to per_interface by specifying row_bira_location:follow_comparators in the DftSpecification.

Table 5-1. BIRA Module Location

comparator_location:	per_interface	shared_in_controller
Row BIRA	Controller or Interface[1]	Controller
Column/IO BIRA	Interface	Controller
Row + Col BIRA	Interface	Controller

1. Row BIRA logic located at the controller when row_bira_location is set to controller. Row BIRA logic is located at the memory interface when row_bira_location is set to follow_comparators.

Limitations and Restrictions

The following limitations and restrictions apply to performing repair analysis.

- When using the preserve repair analysis register feature (PatternsSpecification property preserve_fuse_register_values: on), it is assumed that the device always remains powered up and the register contents are not affected as a result of voltage bumping. Also, the Test Access Port (TAP) cannot be reset.

- Although Tessent MemoryBIST supports insertion into a Verilog or VHDL RTL environment, only Verilog HDL is generated.

- The Row and Column BIRA engine may not always find a repair solution even if one exists, because the engine does not perform an exhaustive search of the repair solutions. To minimize area and test time the engine assigns all spare columns first unless it encounters a multi-bit error at a same address. In that case, the engine assigns a spare row even if spare columns are available. This repair strategy does not find a solution for some combinations of errors with a common row or column address appearing in a certain sequence. However, the impact on yield is negligible due to the low number of these combinations.

- The BIRA engine cannot change the fuse mapping of previously allocated spare elements to create a more optimal repair solution if errors are found during subsequent test sessions during an incremental repair flow. The low number of these occurrences have a negligible impact on yield.

- When several ShiftedIO properties are specified in the Memory BIST library with the same *bitString* value, the IO/Column BIRA does not always find a repair solution even if one exists. The memory is incorrectly declared as non-repairable if an error occurs during the same access at two or more IOs specified with the same *bitString* value. However, the impact on yield is minimal.

Built-In Self Repair (BISR)

Memories with self-repair capability must have the correct repair mapping information specified in the memory library file. Two types of memory repair interfaces are available— serial and parallel:

- Memories with serial self-repair have scan ports on the memory module. The self-repair fuses are accessed serially by scanning in the repair fuse values.

- Memories with parallel self-repair interface have ports on the memory module that control each repair fuse.

Figure 5-2 shows a module that contains two memories, MEM1 and MEM2. The memory MEM1 has a serial BISR interface and is shown with its internal BISR register. The repairable memory MEM2 has a parallel BISR interface. The parallel connections between the external BISR register and the memory parallel repair ports are shown. The BISR chain control signals are also shown. This figure illustrates how the BISR registers are connected to form a single BISR chain that is accessible using the BISR_SI and BISR_SO ports. When scanning values into the BISR chain, the content of the internal BISR register is the same as the external BISR register. The BISR_SELECT signal controls a multiplexer that selects the internal or external

BISR register. The BISR_SELECT signal controls which BISR register to scan out. The length of the internal BISR register must match that of the external BISR register.

Figure 5-2. Memories With Serial and Parallel Self-Repair Interfaces
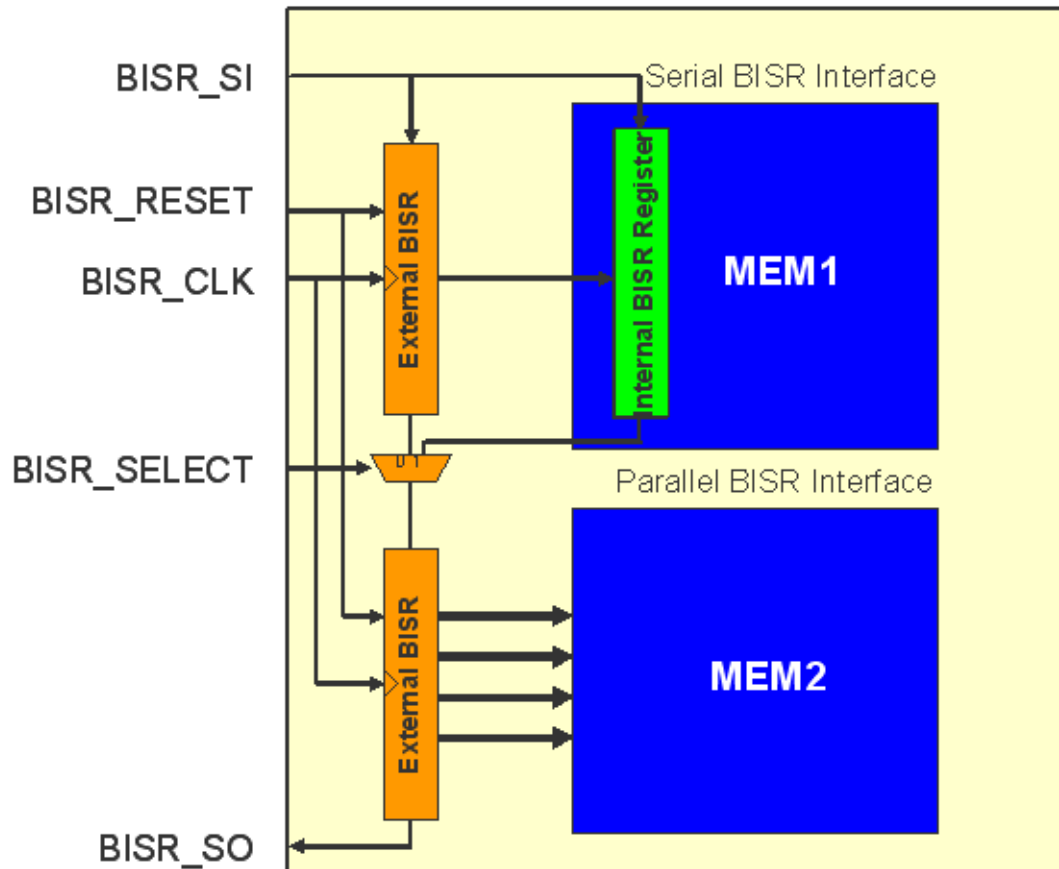
Implementing Soft Repair

Two repair methods are supported by the BISR controller: soft or hard. The repair method is specified during DFT insertion using the repair_method property inside the DftSpecification. When repair_method is set to hard, the memory repair information is stored in a fuse box that enables the repair information to be stored permanently inside the device. This is the default repair method.

When the repair_method is set to soft, the memory repair information is not stored permanently inside the device. It must be reloaded or recalculated every time the device or a power domain region is powered on. If the repair solution is stored externally, it can be loaded using a bisr_chain_access mode. Another option is to re-run the Redundancy Analysis by running memoryBist followed by a BIRA to BISR data transfer for the affected regions. A fuse box interface model is not required when implementing the soft-repair methodology. The BISR controller operation modes associated with fuse box accesses are disabled, such as the

self_fuse_box_program and verify_fuse_box autonomous modes, as well as all fuse_box_access run modes.

Trade-offs and Benefits

When using soft repair, the power-up time of a device is longer because the execution of the memoryBist with Redundancy Analysis, followed by the loading of the repair information in the BISR chains, must be performed before the functional mode of operation can begin. Furthermore, specific defects that require special operating conditions may not be detected during the initial memoryBist sequence.

However, the soft repair method enables significant area savings because it does not require the presence of an eFuse in the design. Also, this method enables an unlimited number of repair sessions. If new defects are detected and are repairable, the new repair solution can be loaded in the BISR chain.

Limitation

The BISR controller must be controlled exclusively through the MissionMode controller and the TDR of the BISR controller must never be reset to avoid resetting the BISR chains. For the same reason, the IJTAG network or TAP must not be reset, even when operating the device in functional mode.

Memory Library Preparation and Repair Registers Description

This section includes information that is not required if an IP vendor provides your memory libraries. The explanation of the repair registers is needed only if you are planning to use the BIRA logic with your own BISR implementation.

Repair Analysis for IO/Column Elements

This section describes how to specify spare IO/column-only elements, as well as the scope of repair in the memory array for each of the spare elements.

The section covers the following topics:

- IO or Column replacement mechanisms. Refer to the "IO/Column Repair" section.

- How to define the built-in repair analysis using the syntax of the memory library file. Refer to the "Implementing IO/Column Repair Analysis" section.

- Memory Library File examples for defining IO or column built-in repair analysis. Refer to the "Memory Library File Sample Syntax" section.

- Description of registers provided for each memory segment defined in the memory library file(s). Refer to the following:

  - "Repair Status Register"

  - "Fuse Registers"

- How the register status and FuseSet values are propagated to output ports on the memory interface or memory controller. Refer to the "Repair Analysis Output Ports for IO/Column Repair" section.

The final step in implementing BIRA is to certify, or validate the memory library file built-in repair analysis settings. Refer to the "Certifying TCD Memory Library Files With memlibCertify in Tessent Shell" appendix for further information. Refer to the "Verifying BISR at the Block Level" section for information on validating the repairable memories and BIRA/BISR logic at the block level.

IO/Column Repair

Memories with redundant IO or Column elements can be used to improve chip yield. IO repair replaces an entire memory sub-array and the associated columns for an IO, whereas column repair replaces a single column across one or more IOs. Each mechanism is described in the following sections.

IO Replacement

In the IO replacement mechanism, an entire memory sub-array is duplicated. The redundant element can repair any failing column associated with a memory IO.

Figure 5-3 illustrates the structure of an 8-bit repairable memory with 4-to-1 column muxing and one redundant IO. The logic for the IO replacement mechanism is highlighted in blue.

Figure 5-3. Example Memory With Redundant IO
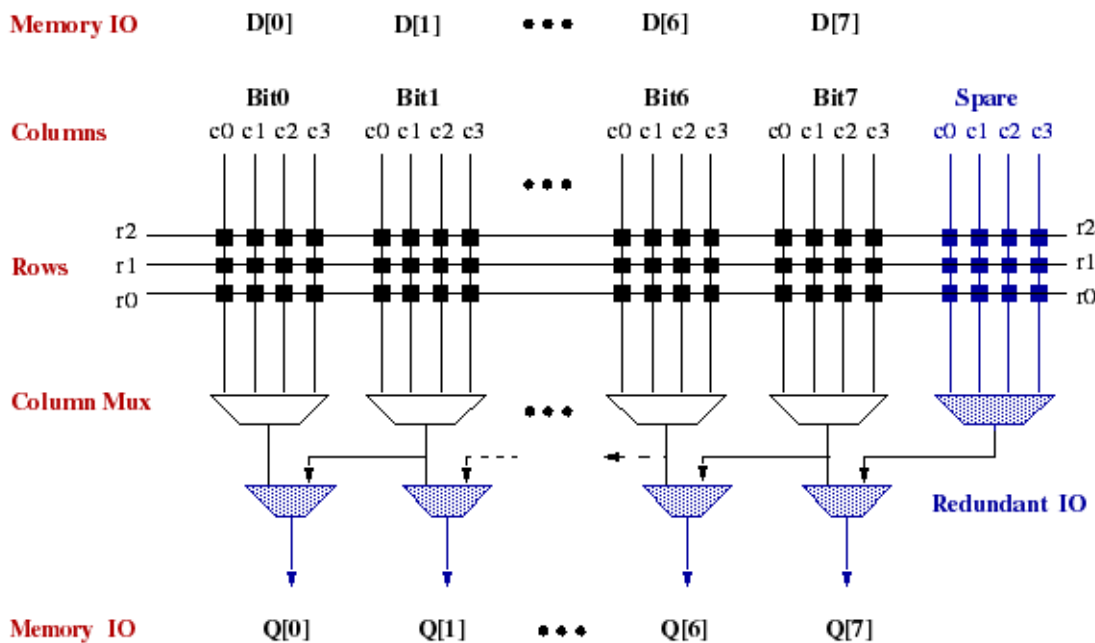
For IO replacement, the memory is repairable when one or more faults are along the same column or one or more failing columns are within the same memory IO.

For example, assume that a fault is detected on data bit Q[6] in the memory illustrated in Figure 5-3. The memory is repaired by bypassing the Bit6 sub-array as follows:

- Port D[6] writes into the Bit7 sub-array

- Port D[7] writes into the redundant sub-array

- Port Q[6] receives the output of column mux 7

- Port Q[7] receives the output of column mux in the redundant sub-array

To identify the failing IO, the built-in repair analysis circuit must log the defective IO value.

Column Replacement

In the column replacement mechanism, a single column is duplicated. The redundant element can repair one failing column across one or more memory IOs.

Figure 5-4 illustrates the structure of an 8-bit repairable memory with 4-to-1 column muxing and one redundant column. The logic for the column replacement mechanism is highlighted in blue.

Figure 5-4. Example Memory With Redundant Column
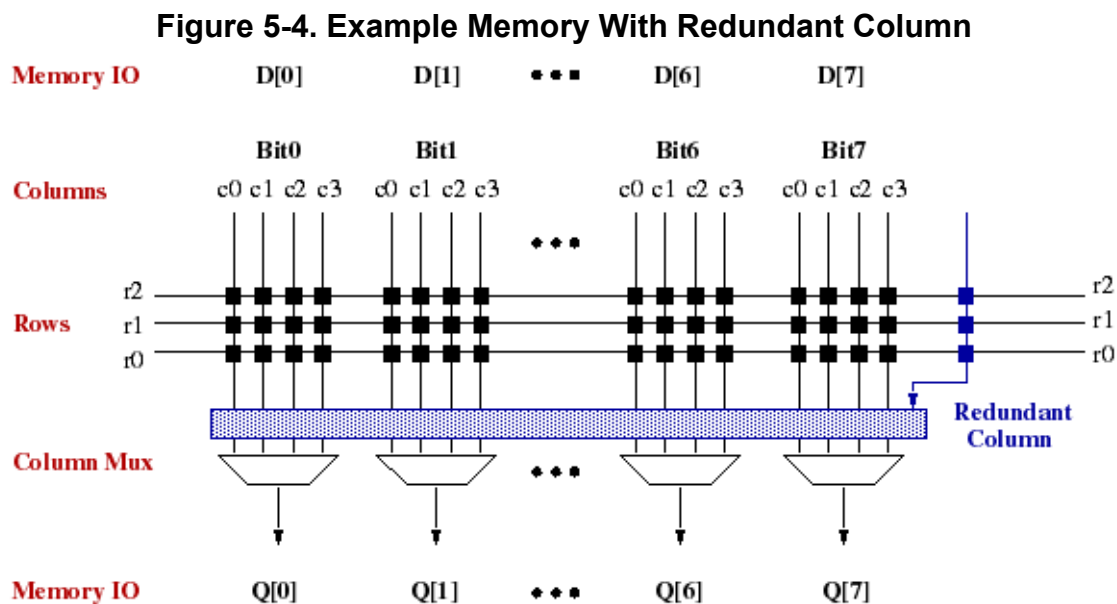
For column replacement, the memory is repairable when one or more faults are along the same column or exactly one failing column is within the specified scope of repair.

To identify the failing column, the built-in repair analysis circuit must log the defective IO value as well as the faulty column address.

Implementing IO/Column Repair Analysis

The topics in this section show how to specify IO/column repair elements, as well as the scope of repair in the memory array for each of the spare elements. An example implementation showing the correct memory library file syntax is also provided.

Tessent MemoryBIST Memory Library File Showing IO/Column Repair

To implement built-in repair analysis, you must define the RedundancyAnalysis wrapper in the memory library file.

The properties and wrappers within the RedundancyAnalysis wrapper contain information about the repairable memory segments, the number of spare elements within a segment, and the addresses to be logged for replacing a defective IO/column with a spare.

Figure 5-5 summarizes the syntax of the memory library file used to support IO/column repair analysis.

Figure 5-5. Repair Analysis Support in Memory Library File

```
RedundancyAnalysis {
   ColumnSegmentRange {
      SegmentAddress [y]: AddressPort(<name>);
         .
         . //Repeat for all SegmentAddress bits
         .
   }
   ColumnSegment (<SegmentName>) {
      ColumnSegmentCountRange [<lowRange>: <highRange>];
      NumberOfSpareElements : <int>;
      ShiftedIORange  : Data [15:0];
      FuseSet {
         Fuse [<bitIndex>]: AddressPort(<name>) |
               not AddressPort(<name>) |
               LogicHigh | LogicLow;
         FuseMap [<HighBitRange>:<LowBitRange>] {
            NotAllocated : <bitString>;
            ShiftedIO (<DataPortName>):<bitString>;
            .
            . //Repeat for all IO within ShiftedIORange
            .
         }
         .
         . //Repeat for all Fuse bits
         .
      }
   }
   .
   . //Repeat for all ColumnSegments
   .
}
```

ColumnSegmentRange Wrapper

Use the optional ColumnSegmentRange wrapper to define a portion of the memory address space where a spare element can replace a defective element.

The ColumnSegmentRange wrapper is defined in the RedundancyAnalysis wrapper of the memory library. This wrapper does not need to specified if only one RedundancyAnalysis/ColumnSegment wrapper is defined. In this configuration, the column segment encompasses the entire memory address space.

SegmentAddress Property

Use the repeatable SegmentAddress property to specify the significant column address bits that are used to encode the ColumnSegment/ColumnSegmentCountRange limits. These range limits are used to define the portion of the column address space where the segment's spare elements can replace a defective IO/Column element.

ColumnSegment Wrapper

The ColumnSegment wrapper is required for implementation of IO/column repair, and is used to define one or more segments of the memory space that have spare elements.

For IO/column repair analysis, you must specify at least one RedundancyAnalysis/ColumnSegment wrapper. Every ColumnSegment is assumed to have one spare element.

ColumnSegmentCountRange and RowSegmentCountRange Properties

Use the optional ColumnSegmentCountRange and RowSegmentCountRange properties to define the lowRange and highRange for the defined segment address bits. When a defective element is within this address range, a spare element can be allocated from this segment.

NumberOfSpareElements Property

Use the NumberOfSpareElements property to specify the number of redundant elements within the defined segment. The default value is 1.

ShiftedIORange Property

Use the ShiftedIORange property to define a group of IO/Data bits where spare elements can replace a faulty IO. When a defective element is within this group, a spare element can be allocated from this segment. The default value for this property is equal to the entire IO range.

FuseSet Wrapper

Use the mandatory FuseSet wrapper in the ColumnSegment wrapper to provide detailed information about the fuse register. The fuse register contains the bitmapping information for identifying the defective IO/column.

- **Fuse Property**

  Use the optional Fuse property to indicate the memory address ports to be logged in the fuse registers upon detecting failure. You must define the address port using the syntax:

  ```
  Fuse[<bitIndex>]: AddressPort(<name>) | not AddressPort(<name>) |
      LogicHigh | LogicLow;
  ```

  Multiple fuse register bits are possible, but they must be indexed starting from 0. The constant LogicHigh | LogicLow values can be captured in the BIRA fuse register when a failure is detected.

- **FuseMap Wrapper**

  Use the mandatory FuseMap wrapper to define the HighBitRange and the LowBitRange for the fuse used to map the defective IO to the corresponding fuse register value.

<hr>

Note

The FuseMap wrapper is independent of the Fuse property.

<hr>

o **NotAllocated Property**

Use the optional NotAllocated property to specify a fuse register value that indicates no repair is needed for the column segment. If this property is not defined, an allocation bit is used as an indicator when repair is required, and faulty IO information is logged. The allocation bit is the MSB bit of the fuse registers.

o **ShiftedIO Property**

Use the mandatory ShiftedIO property to specify the values to be logged in the fuse register, which identifies each defective IO.

Memory Library File Sample Syntax

This section provides examples of the memory library sample syntax.

Examples

Example 1: Memory With IO Redundancy

Suppose we have a 32-bit memory array with IO redundancy as shown in Figure 5-6.

Figure 5-6. Example of Memory With IO Redundancy
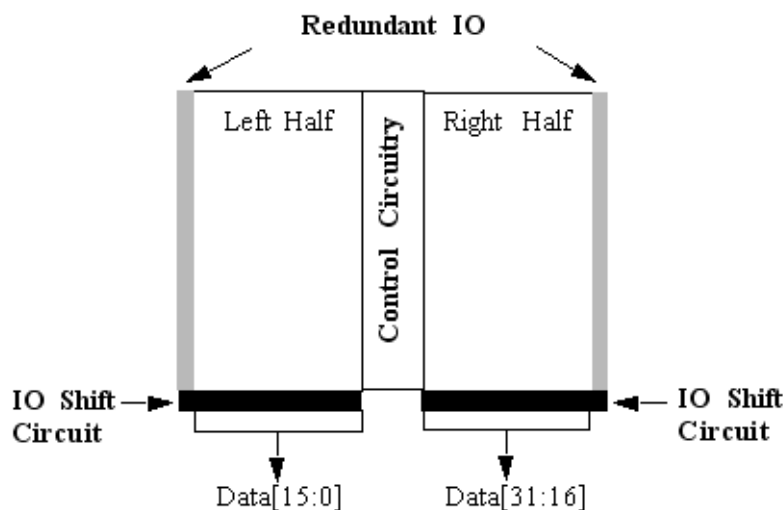
Each half of the memory array has a single redundant IO that can repair any IO within the corresponding half. Assume that each redundant IO is programmed from a 5-bit fuse as follows:

- Bits [3:0] are used to identify the defective IO.

- Bit [4] is the MSB of the fuse register; it is used as an allocation bit.

Figure 5-7 illustrates an example of the memory library file syntax to define the above repair analysis capability.

Figure 5-7. RedundancyAnalysis Wrapper Syntax for Example 1

```
RedundancyAnalysis {
    ColumnSegment (LEFT) {
          ShiftedIORange: Data[15:0];
          FuseSet {
                FuseMap[3:0] {
                    ShiftedIO(Data[0]): 4'b0000;
                    ShiftedIO(Data[1]): 4'b0001;
                    ShiftedIO(Data[2]): 4'b0010;
                    ShiftedIO(Data[3]): 4'b0011;
                    ShiftedIO(Data[4]): 4'b0100;
                    ShiftedIO(Data[5]): 4'b0101;
                    ShiftedIO(Data[6]): 4'b0110;
                    ShiftedIO(Data[7]): 4'b0111;
                    ShiftedIO(Data[8]): 4'b1000;
                    ShiftedIO(Data[9]): 4'b1001;
                    ShiftedIO(Data[10]): 4'b1010;
                    ShiftedIO(Data[11]): 4'b1011;
                    ShiftedIO(Data[12]): 4'b1100;
                    ShiftedIO(Data[13]): 4'b1101;
                    ShiftedIO(Data[14]): 4'b1110;
                    ShiftedIO(Data[15]): 4'b1111;
                }
          }
    }
    ColumnSegment (RIGHT) {
          ShiftedIORange: Data[31:16];
          FuseSet {
                FuseMap[3:0] {
                    ShiftedIO(Data[16]): 4'b0000;
                    ShiftedIO(Data[17]): 4'b0001;
                    ShiftedIO(Data[18]): 4'b0010;
                    ShiftedIO(Data[19]): 4'b0011;
                    ShiftedIO(Data[20]): 4'b0100;
                    ShiftedIO(Data[21]): 4'b0101;
                    ShiftedIO(Data[22]): 4'b0110;
                    ShiftedIO(Data[23]): 4'b0111;
                    ShiftedIO(Data[24]): 4'b1000;
                    ShiftedIO(Data[25]): 4'b1001;
                    ShiftedIO(Data[26]): 4'b1010;
                    ShiftedIO(Data[27]): 4'b1011;
                    ShiftedIO(Data[28]): 4'b1100;
                    ShiftedIO(Data[29]): 4'b1101;
                    ShiftedIO(Data[30]): 4'b1110;
                    ShiftedIO(Data[31]): 4'b1111;
                }
          }
    }
}
```

This redundancy specification implements three registers in the memory interface or memory controller for repair analysis reporting. These registers can be shifted out after the memory BIST execution to facilitate fuse box programming.

Shown in Figure 5-8, the 2-bit register BP1_RA_MEM1_REPAIR_STATUS_REG indicates the overall memory repairability. The remaining two registers correspond to the FuseSet wrappers. The 5-bit FUSE_REG registers log the defective I/O and include the allocation bit as the MSB.

Figure 5-8. FuseSet Registers for Example 1
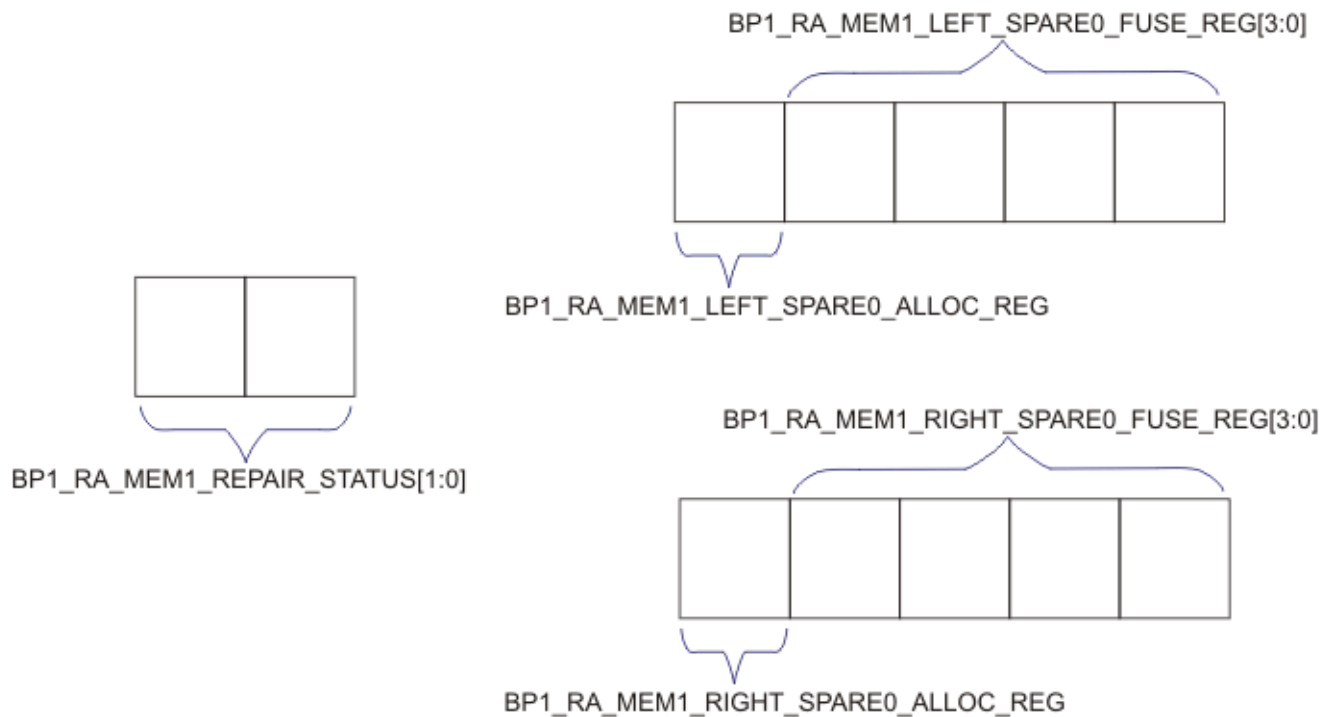
For information on the built-in repair analysis registers, refer to the "Repair Analysis Registers" section.

Example 2: Memory With IO/Column Redundancy

Suppose we have a 32-bit memory array with IO/column redundancy as shown in Figure 5-9.

Figure 5-9. Example of Memory With Column/IO Redundancy
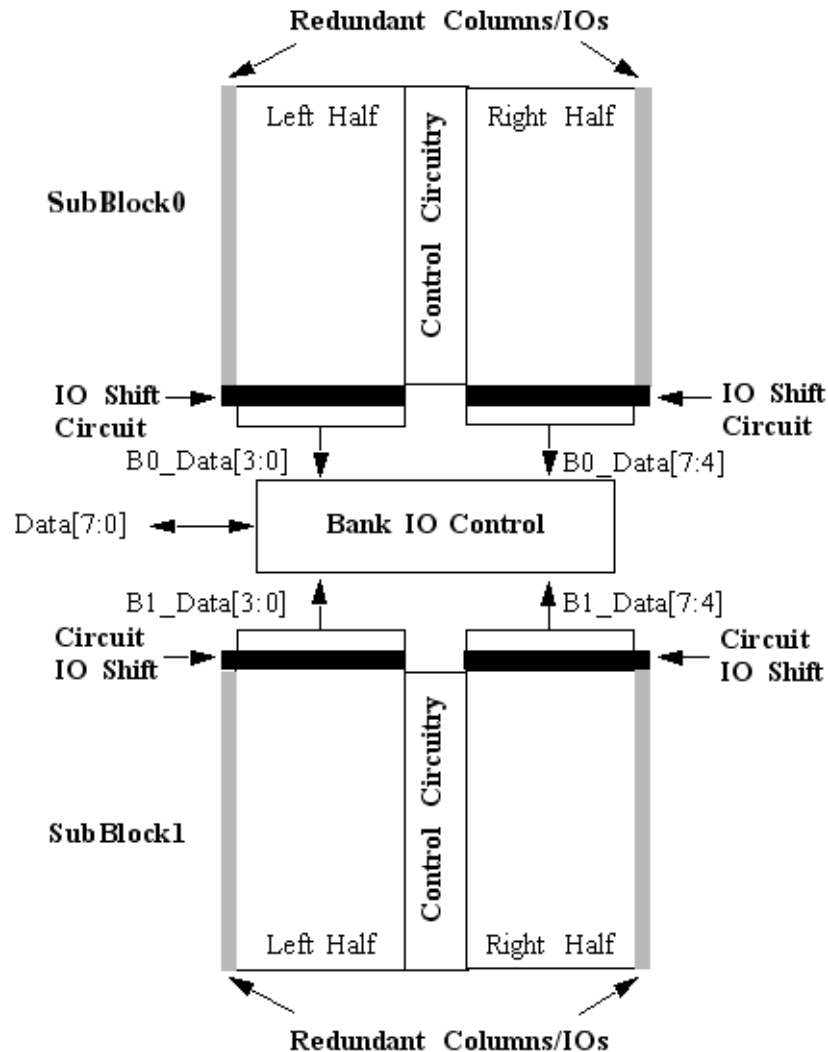
The memory is divided into two blocks in this example. Within a memory block, each half of the memory array has one redundancy IO/column that can repair any IO/column within the corresponding half.

Assume that each redundant IO is programmed by a 5-bit fuse. The fuse box bitmapping for the defective IO/column is shown in Table 5-2.

Table 5-2. Fuse Box for Mapping Defective IO/Columns

B0/B1 Left-Hand Side		B0/B1 Right-Hand Side	
Fuse Box Value	Shifted IO	Fuse Box Value	Shifted IO
00000	None	00000	None
00001	0	00001	4
00010	1	00010	5

Table 5-2. Fuse Box for Mapping Defective IO/Columns (cont.)

B0/B1 Left-Hand Side		B0/B1 Right-Hand Side	
00011	2	00011	6
00100	3	00100	7

In this example, the RedundancyAnalysis wrapper is specified as shown in Figure 5-10.

Figure 5-10. RedundancyAnalysis Wrapper Syntax for Example 2

```
RedundancyAnalysis {
    RowSegmentRange {
        SegmentAddress[0]: AddressPort(Add[13]);
    }
    ColumnSegment (SubBlock0_LEFT) {
        ShiftedIORange: Data[3:0];
        RowSegmentCountRange [1'b0:1'b0];
        FuseSet {
            FuseMap[4:0] {
                NotAllocated:        5'b00000;
                ShiftedIO (Data[0]):  5'b00001;
                ShiftedIO (Data[1]):  5'b00010;
                ShiftedIO (Data[2]):  5'b00011;
                ShiftedIO (Data[3]):  5'b00100;
            }
        }
    }
    ColumnSegment (SubBlock0_RIGHT) {
        ShiftedIORange: Data[7:4];
        RowSegmentCountRange [1'b0:1'b0];
        FuseSet {
            FuseMap[4:0] {
                NotAllocated:        5'b00000;
                ShiftedIO (Data[4]):  5'b00001;
                ShiftedIO (Data[5]):  5'b00010;
                ShiftedIO (Data[6]):  5'b00011;
                ShiftedIO (Data[7]):  5'b00100;
            }
        }
    }
    ColumnSegment (SubBlock1_LEFT) {
        ShiftedIORange: Data[3:0];
        RowSegmentCountRange [1'b1:1'b1];
        FuseSet {
            FuseMap[4:0] {
                NotAllocated:        5'b00000;
                ShiftedIO (Data[0]):  5'b00001;
                ShiftedIO (Data[1]):  5'b00010;
                ShiftedIO (Data[2]):  5'b00011;
                ShiftedIO (Data[3]):  5'b00100;
            }
        }
    }
    ColumnSegment (SubBlock1_RIGHT) {
        ShiftedIORange: Data[7:4];
        RowSegmentCountRange [1'b1:1'b1];
        FuseSet {
            FuseMap[4:0] {
                NotAllocated:        5'b00000;
                ShiftedIO (Data[4]):  5'b00001;
                ShiftedIO (Data[5]):  5'b00010;
                ShiftedIO (Data[6]):  5'b00011;
                ShiftedIO (Data[7]):  5'b00100;
            }
        }
    }
```

```
        }
    }
```

This redundancy specification implements five registers in the memory interface or memory controller to report the repair information. These registers can be shifted out after the memory BIST execution to facilitate fuse box programming.

Shown in Figure 5-11, the 2-bit register BP1_RA_MEM1_REPAIR_STATUS indicates the overall memory repairability. The remaining four registers correspond to the FuseSet wrappers and log the 5-bit encoding for the memory IO, as defined in Table 5-2. The fuse registers does not include the allocation bit because the NotAllocated property is being specified in the FuseMap definition.

Figure 5-11. FuseSet Registers for Example 2
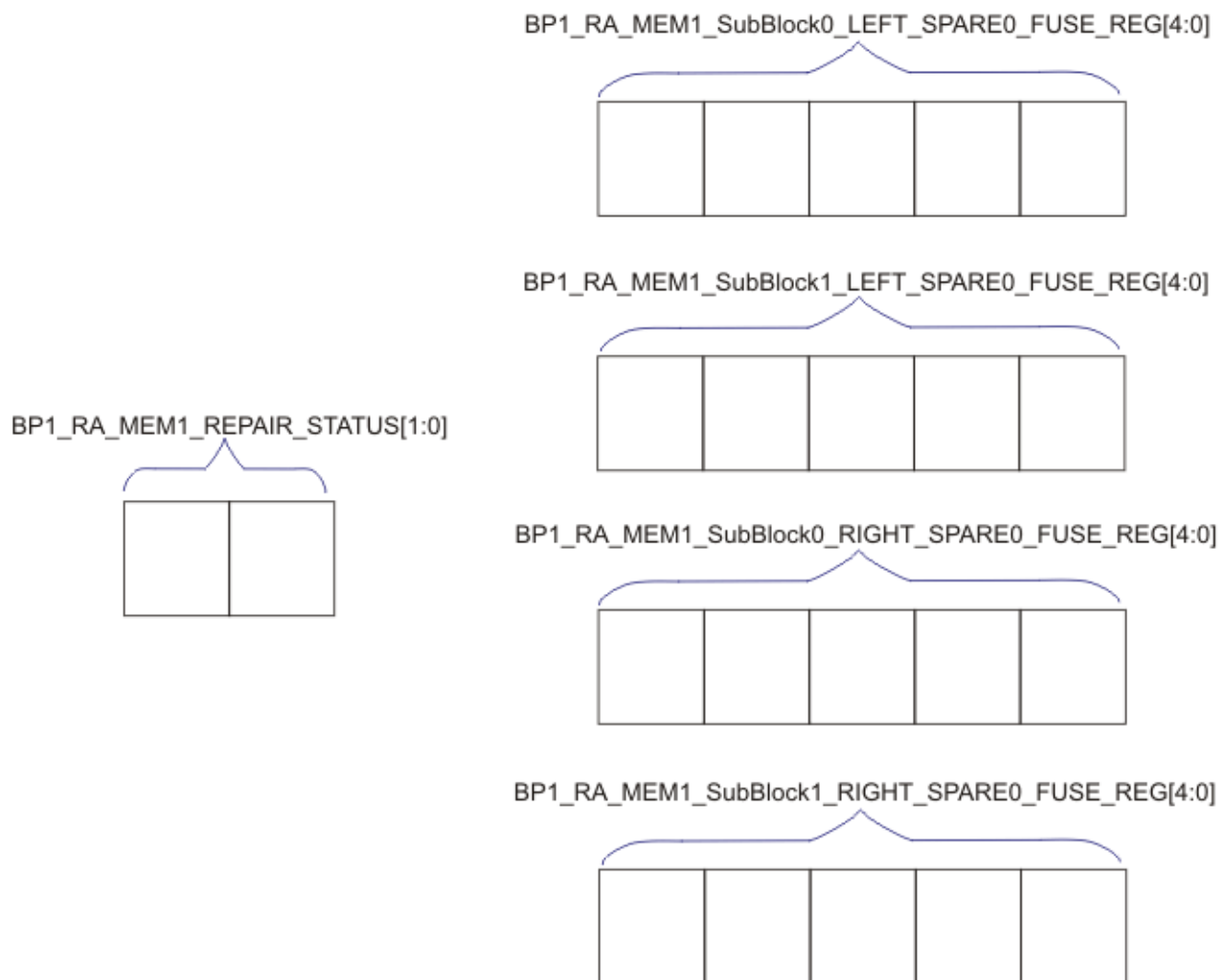
BP1_RA_MEM1_SubBlock0_LEFT_SPARE0_FUSE_REG[4:0]

BP1_RA_MEM1_SubBlock1_LEFT_SPARE0_FUSE_REG[4:0]

BP1_RA_MEM1_REPAIR_STATUS[1:0]

BP1_RA_MEM1_SubBlock0_RIGHT_SPARE0_FUSE_REG[4:0]

BP1_RA_MEM1_SubBlock1_RIGHT_SPARE0_FUSE_REG[4:0]

For information on the built-in repair analysis registers, refer to the "Repair Analysis Registers" section.

Repair Analysis Registers

When built-in repair analysis is implemented using Tessent Shell MemoryBIST, a set of registers is created in the hardware for storing the repair results from the BIST execution.

This set of registers consists of the following:

- One Repair Status Register for each memory with redundancy

- One or more Fuse Registers for each memory repair segment defined in the memory library file(s)

The comparator_location property setting in the MemoryBist/Controller/Step wrapper of the DftSpecification determines the location of the I/O and column repair analysis hardware associated with the repairable memory:

- comparator_location:shared_in_controller — The repair analysis registers are created in the memory controller.

- comparator_location:per_interface — The repair analysis registers are created in the corresponding memory interface.

Repair Status Register

In the hardware, the overall ability of the memory to be repaired is reported by the repair status register:

```
<memory_instance_name>_REPAIR_STATUS
```

The repair status register specifies whether the memory requires a repair, is not repairable, or does not require a repair (when no failure is detected). The bit decode assignments for the repair status register are described in Table 5-3.

Table 5-3. <MEMORY_INSTANCE_NAME>_REPAIR_STATUS Decodes

Bit1 Bit0	Repair Status
00	No Repair Required
01	Repair Required
1x	Not Repairable

The repair status register(s) are located on the setup chain. Therefore, these registers can be serially scanned out of the memory controller or memory interface.

Fuse Registers

For each memory segment, two registers are required for each spare element:

- Allocation Register (if the NotAllocated property is not specified)

- FuseSet Register

Allocation Register

The allocation register, if present, contains a bit that specifies whether the spare element is allocated (needs repair) or is not allocated. This bit has the following name:

```
<memory_instance_name>_<segment_name>_SCOL#_ALLOC_REG
```

A value of 0 in this bit represents NOT allocated (no repair required), and a 1 represents allocated (repair required). If allocated, the value in the FuseSet register is dictated by the FuseSet wrapper in the memory library file.

FuseSet Register

The FuseSet register contains the fuse bits as specified by the FuseSet wrapper for the segment. The register is composed of the following:

- The I/O map value corresponding to the defective element.

- If the Fuse property in the FuseSet wrapper is defined, then the FuseSet register also includes the address value driven on the memory address bus for the defective element.

Repair Analysis Output Ports for IO/Column Repair

The Repair Status and FuseSet values are propagated to output ports on the memory interface or memory controller.

The location of the BIRA module and registers depends on the comparator_location property setting in the MemoryBist/Controller/Step wrapper for each memory interface.

BIRA Port Names on the Controller When Using Shared Comparators

When setting comparator_location: shared_in_controller, the BIRA registers are located inside the memory BIST controller, and the port naming convention has the following prefix:

```
<memory_instance_name>_<segment_name>_SCOL<x>_...
```

The *<memory_instance_name>* corresponds to the memory instance defined by the instance_name property in the MemoryInterface or ReusedMemoryInterface wrappers in the DftSpecification MemoryBist/Controller/Step wrapper.

The *<segment_name>* corresponds to the ColumnSegment(*<SegmentName>*) inside the memory template wrapper.

The *<x>* index corresponds to the *N*th spare element. The spare elements are numbered starting from 0.

BIRA Port Names on the Interface When Using Local Comparators

When setting comparator_location: per_interface, the BIRA registers are located inside the memory interface, and the port naming convention has the following prefix:

```
<segment_name>_SCOL<x>_...
```

The *<segment_name>* corresponds to the ColumnSegment(*<SegmentName>*) inside the memory template wrapper.

The *<x>* index corresponds to the *N*th spare element. The spare elements are numbered starting from 0. The results of the memory repair analysis can be determined by monitoring the following ports:

- *<prefix>*_REPAIR_STATUS

This output port reports the overall repair status of the memory with redundancy. The bit decode assignments are described in Table 5-3.

- *<prefix>*_FUSE_REG

This output port reports the IO map value for the defective element.

- *<prefix>*_FUSE_ADD_REG

This output port contains the memory column address where a redundant element must be allocated.

- *<prefix>*_ALLOC_REG

This single-bit output port indicates if the spare element must be allocated.

Repair Analysis for Row Elements

This section describes how to specify spare row-only elements and the scope of repair in the memory array for each of the spare elements. This section covers the following topics:

- Row replacement mechanism. Refer to the "Row Repair" section.

- How to define the built-in repair analysis for row elements using the syntax of the memory library file. Refer to the "Implementing Row Repair Analysis" section.

- Memory Library File examples for defining row built-in repair analysis. Refer to the "Tessent MemoryBIST Memory Library File Showing Row Repair" section.

- Description of registers provided for each memory segment defined in the memory library file(s). Refer to the following:

  o "Repair Status Register"

  o "Fuse Registers"

- How the register status and FuseSet values are propagated to output ports on the memory interface/collar or memory controller. Refer to the "Repair Analysis Output Ports for Row Repair" section.

The final step in implementing BIRA is to certify, or validate the memory library file built-in repair analysis settings. Refer to the "Certifying TCD Memory Library Files With memlibCertify in Tessent Shell" appendix for further information. Refer to the "Verifying BISR at the Block Level" section for information on validating the repairable memories and BIRA/ BISR logic at the block level.

Row Repair

Memories with one or more redundant rows are often used to improve chip yield. Faulty rows in the memories are identified through failure mapping or repair analysis. Access to the faulty row addresses is redirected to the redundant elements. The chip with repaired memories becomes usable and does not need to be discarded.

Row Replacement

In the Row replacement mechanism, additional rows are built into the memory array. These redundant elements can repair any failing row within the entire memory or within a specific bank. The replacement mechanism depends on the memory design.

For instance, a redundant element might replace a single physical row in one implementation, while another scheme always replaces multiple consecutive rows even if there is only one row failure.

Figure 5-12 illustrates the structure of a repairable memory containing two banks with dedicated redundant rows.

Figure 5-12. Example Memory With Redundant Rows
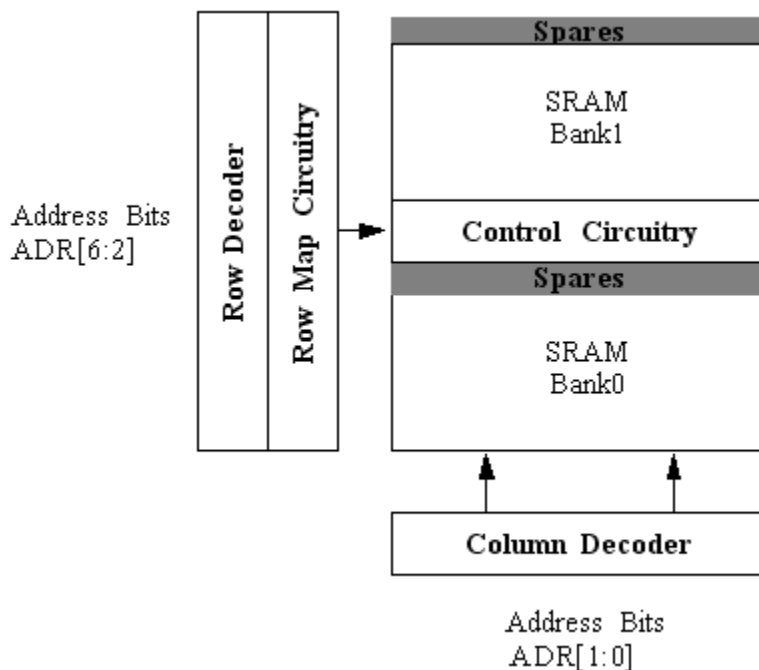
Implementing Row Repair Analysis

This section describes how to specify any row elements and the scope of repair in the memory array for each of the spare elements and provides examples of repair analysis information in the memory library file.

A set of registers is provided for each memory repair segment defined in the memory library file. For description, refer to the "Repair Analysis Registers" section.

Tessent MemoryBIST Memory Library File Showing Row Repair

To implement built-in repair analysis, you must define the RedundancyAnalysis wrapper in the memory library file.

The properties and wrappers within the RedundancyAnalysis wrapper contain information about the repairable memory segments, the number of spare elements within a segment, and the addresses to be logged for replacing a defective row location with a spare.

To specify row segments, use the RowSegmentRange and RowSegment wrappers and their contents. Refer to the descriptions starting with the "RowSegmentRange Wrapper" and "RowSegment Wrapper" sections.

Figure 5-13 summarizes the syntax of the memory library file used to support row repair analysis.

Figure 5-13. Repair Analysis/Row Support in Memory Library File

```
RedundancyAnalysis {
   RowSegmentRange {
      SegmentAddress [y]: AddressPort(<name>);
         .
         . //Repeat for all SegmentAddress bits
         .
   }
   RowSegment (<segmentName>){
      NumberOfSpareElements: <integer>;
      RowSegmentCountRange [<lowRange>: <highRange>];
      FuseSet {
         Fuse [<bitindex>]: AddressPort(<name>) |
               not AddressPort(<name>) |
               LogicHigh | LogicLow;
         .
         . //Repeat for all Fuse bits
         .
      }
   }
   .
   . //Repeat for all RowSegments
   .
}
```

RowSegmentRange Wrapper

Use the optional RowSegmentRange wrapper to define a portion of the memory address space where the spare elements can replace one or more defective rows.

The RowSegmentRange wrapper is defined in the RedundancyAnalysis wrapper of the memory library.

The following usage conditions apply:

- You do not need to specify the RowSegmentRange wrapper if only one RedundancyAnalysis/RowSegment wrapper is defined within the RedundancyAnalysis wrapper. In this case, the RowSegmentRange wrapper defaults to encompass the entire memory address space.

- If more than one RowSegment wrapper is defined within the RedundancyAnalysis wrapper, the RowSegmentRange wrapper is required, and all the segment ranges specified for all wrappers must combine to encompass the entire memory address space.

SegmentAddress Property

Use the repeatable SegmentAddress property to specify the significant row address bits that are used to encode the RowSegment/RowSegmentCountRange limits. These range limits are used to define the portion of the row address space where the segment's spare elements can replace a defective row element.

The following usage conditions apply:

- SegmentAddress [*y*] — For all specified SegmentAddress properties, *y* must start from 0 and encompass all valid integers to *n*-1, where *n* is the number of the SegmentAddress properties specified.

- *name* must identify a port defined in the memory library file with Port/Function: Address specified. If the address port is a bused port, the name must identify a single bit within the bused port.

RowSegment Wrapper

The RowSegment wrapper is required for implementation of row repair, and is used to define one or more segments of the memory space that have spare row elements.

For row repair analysis, you must specify at least one RedundancyAnalysis/RowSegment wrapper. Every RowSegment is assumed to have one spare element.

If the RedundancyAnalysis/RowSegmentRange wrapper is specified, the number of RowSegment wrappers defined must encompass the entire memory address space.

NumberOfSpareElements Property

Use the NumberOfSpareElements property to specify the number of redundant elements within the defined segment. The default value is 1.

RowSegmentCountRange Property

Use the optional RowSegmentCountRange property to specify the lowRange and highRange for a defined segment based on the significant row address bits defined in the RowSegmentRange wrapper. When a defective element is within this segment range, a spare element can be allocated from this segment. RowSegmentCountRange can be specified only when at least one RedundancyAnalysis/RowSegmentRange/SegmentAddress bit is defined.

FuseSet Wrapper

Use the mandatory RowSegment/FuseSet wrapper to map the fuse register bits to address ports on the memory. The fuse register bit logs the value of this address port for the defective element. This wrapper is specified once per RowSegment wrapper.

- **Fuse Property**

  Use the optional Fuse property to indicate the memory address ports to be captured in the fuse registers when a failure is detected. These fuse bits are defined per row segment. Constant LogicHigh and LogicLow values also can be captured by the BIRA fuse register when a failure is detected.

The following usage conditions apply:

o A minimum of one Fuse must be defined per FuseSet.

o name must be either a scalar port or a single bit of a bused port.

o name must be a port defined by a Port wrapper with Function: Address in the memory library file. Constant values LogicHigh and LogicLow can be used instead of address bits.

Memory Library File Sample Syntax

This section discusses the structure of an example memory library file for a repairable memory.

The memory library file syntax example in Figure 5-14 defines the row repair analysis capability for the memory illustrated in Figure 5-12. The memory contains 2 banks of 16 rows and 4 columns. Each bank implements 2 redundant elements.

Figure 5-14. Memory Library File Sample Syntax

```
RedundancyAnalysis {
    RowSegmentRange {
        SegmentAddress[0]: AddressPort(ADR[6]);
    }
    RowSegment (Bank0){
        NumberOfSpareElements: 2;
        RowSegmentCountRange [1'b0:1'b0];
        FuseSet {
            Fuse[3]: AddressPort(ADR[5]);
            Fuse[2]: AddressPort(ADR[4]);
            Fuse[1]: AddressPort(ADR[3]);
            Fuse[0]: AddressPort(ADR[2]);
        }
    }
    RowSegment (Bank1){
        NumberOfSpareElements: 2;
        RowSegmentCountRange [1'b1:1'b1];
        FuseSet {
            Fuse[3]: AddressPort(ADR[5]);
            Fuse[2]: AddressPort(ADR[4]);
            Fuse[1]: AddressPort(ADR[3]);
            Fuse[0]: AddressPort(ADR[2]);
        }
    }
}
```

This sample syntax of the memory library file specifies to Tessent Shell MemoryBIST the following information for row segments:

- The sample memory has two banks. Each bank has two spare elements available for repair. The two spare elements per bank can repair a failure occurring only within that bank. The FuseSet wrapper defines the address bits to be logged for each repairable element.

- Each of the two row segments is located in the address space defined by the AddressPort *ADR[6]*. The RowSegment(*Bank0*) is selected within the address space whereby *ADR[6]* is *1'b0*. The RowSegment(*Bank1*) is selected within the address space whereby *ADR[6]* is *1'b1*.

- For the row segment range, the SegmentAddress bit 0 is defined as memory address port *ADR[6]*.

- Each row segment consists of two spare elements as defined by the NumberOfSpareElements property. This means that for RowSegment(*Bank0*), there are two sets of fuses, each of which logs the value of the ports *ADR[5:2]* for the defective portion of memory within the row segment. Similarly for RowSegment(*Bank1*), there are two sets of fuses, each of which logs the value of the ports *ADR[5:2]* for the defective portion of memory within the row segment.

Repair Analysis Registers

When built-in repair analysis is implemented using Tessent Shell MemoryBIST, a set of registers is created in the hardware for storing the repair results from the BIST execution.

The following registers are provided for each memory segment defined in the memory library file(s):

- One segment Repair Status Register

- One or more Fuse Registers

The comparator_location property setting in the MemoryBist/Controller/Step wrapper of the DftSpecification determines the location of the row repair analysis hardware associated with the repairable memory:

- comparator_location:shared_in_controller — The repair analysis registers are created in the memory controller.

- comparator_location:per_interface — The repair analysis registers are created in the corresponding memory interface.

Repair Status Register

In the hardware, the overall ability of the memory to be repaired is reported by the repair status register:

```
<memory_instance_name>_REPAIR_STATUS
```

The repair status register specifies whether the memory requires a repair, is not repairable, or does not require a repair (when no failure is detected). The bit decode assignments for the repair status register are described in Table 5-4.

Table 5-4. <MEMORY_INSTANCE_NAME>_REPAIR_STATUS Decodes

Bit 1 Bit0	Repair Status
00	No Repair Required
01	Repair Required
1x	Not Repairable

The repair status register(s) are located in the setup chain. Therefore, these registers can be serially scanned out of the memory controller or memory interface.

Fuse Registers

For each memory segment, two registers are required for each spare element:

- Allocation Register (if the NotAllocated property is not specified)

- FuseSet Register

Allocation Register

The allocation register contains a bit specifying if the spare element is allocated and repair is needed, or is not allocated and no repair is needed. The register has the following name:

```
<memory_instance_name>_<segment_name>_SROW#_ALLOC_REG
```

This register is always a single bit where 0 represents NOT allocated and 1 represents allocated. If allocated, the value in the FuseSet register represents a defective portion of the memory to be repaired.

FuseSet Register

The FuseSet register contains the fuse bits as specified by the FuseSet wrapper for the segment. The register is composed of the following:

- The row map value corresponding to the defective element.

- If the Fuse property in the FuseSet wrapper is defined, then the FuseSet register also includes the address value driven on the memory address bus for the defective element.

Repair Analysis Output Ports for Row Repair

The Register Status and FuseSet values are propagated to output ports on the memory interface or memory controller.

The location of the BIRA module and registers depends on the comparator_location property setting in the MemoryBist/Controller/Step wrapper for each memory interface.

BIRA Port Names on the Controller When Using Shared Comparators

When setting comparator_location: shared_in_controller, the BIRA registers are located inside the memory BIST controller, and the port naming convention has the following prefix:

```
<memory_instance_name>_<segment_name>_SROW<x>_...
```

The *<memory_instance_name>* corresponds to the memory instance defined by the instance_name property in the MemoryInterface or ReusedMemoryInterface wrappers in the DftSpecification MemoryBist/Controller/Step wrapper.

The *<segment_name>* corresponds to the RowSegment(*<SegmentName>*) inside the memory template wrapper.

The *<x>* index corresponds to the *N*th spare element. The spare elements are numbered starting from 0.

BIRA Port Names on the Interface When Using Local Comparators

When setting comparator_location: per_interface, the BIRA registers are located inside the memory interface, and the port naming convention has the following prefix:

```
<segment_name>_SROW<x>_...
```

The *<segment_name>* corresponds to the RowSegment(*<SegmentName>*) inside the memory template wrapper.

The *<x>* index corresponds to the *N*th spare element. The spare elements are numbered starting from 0. The results of the memory repair analysis can be determined by monitoring the following ports:

- *<prefix>*_REPAIR_STATUS

This output port reports the overall repair status of the memory with redundancy. The bit decode assignments are described in Table 5-3.

- *<prefix>*_FUSE_REG

  This output port reports the IO map value for the defective element.

- *<prefix>*_FUSE_ADD_REG

  This output port contains the memory column address where a redundant element must be allocated.

- *<prefix>*_ALLOC_REG

  This single-bit output port indicates if the spare element must be allocated.

Repair Analysis for Row and IO/Column Elements

While memories can be designed to provide either row or IO/column repair capability, memories can be designed to provide both.

Repair analysis for such memories is much more complex than single-dimension repair analysis. To provide built-in repair analysis for both row AND column redundancy, the RedundancyAnalysis wrapper in the memory library file must contain both row and column repair information.

This section describes how to specify row and IO/column elements and the scope of repair in the memory array for each of the spare elements. This section covers the following topics:

- How to define the built-in repair analysis for row and IO/column elements using the syntax of the memory library file. Refer to the "Implementing Row and IO/Column Repair Analysis" section.

- Memory Library File examples for defining built-in repair analysis for row and IO/ column elements. Refer to the "Sample Memory Library File Syntax" section.

- Description of registers provided for each memory segment defined in the memory library file(s). Refer to the following:

  o "Repair Status Register"

- How the register status and FuseSet values are propagated to output ports on the memory interface or memory controller. Refer to the "Repair Analysis Output Ports for IO/Column and Row Repair" section.

The final step in implementing BIRA is to certify, or validate the memory library file built-in repair analysis settings. Refer to the "Certifying TCD Memory Library Files With memlibCertify in Tessent Shell" appendix for further information. Refer to the "Verifying BISR at the Block Level" section for information on validating the repairable memories and BIRA/ BISR logic at the block level.

Implementing Row and IO/Column Repair Analysis

This section describes how to specify any row and IO/column elements, as well as the scope of repair in the memory array for each of the spare elements and provides examples for repair analysis information in the memory library file.

Tessent MemoryBIST Memory Library File Showing Row and IO/Column Repair

To implement built-in repair analysis for rows and IO/columns, you must define the RedundancyAnalysis wrapper in the memory library file.

The properties and wrappers within the RedundancyAnalysis wrapper contain information about the repairable memory segments, the number of spare elements within a segment, and the addresses to be logged for replacing a defective location with a spare. Effectively, the description is a superset of information for row and column repair analysis.

- To specify row and column segment ranges, use the RowSegmentRange and ColumnSegmentRange wrappers with their contents. Refer to the descriptions starting with the "RowSegmentRange and ColumnSegmentRange Wrappers" section.

- To specify row and column segments, use the RowSegment and ColumnSegment wrappers with their contents. Refer to the descriptions starting with the "RowSegment and ColumnSegment Wrappers" section.

Figure 5-15 summarizes the syntax of the memory library file used to support built-in repair analysis for both row and IO/column elements.

Figure 5-15. Repair Analysis Support in Memory Library File

```
RedundancyAnalysis {
   RowSegmentRange {
      SegmentAddress [<bitIndex>]: AddressPort(<name>);
      .
      . //Repeat for all SegmentAddress bits
      .
   }//End of RowSegmentRange
   RowSegment (<segmentName>){
      NumberOfSpareElements : <integer>;
      RowSegmentCountRange [<lowRange>: <highRange>];
      ColumnSegmentCountRange [<lowRange>: <highRange>];
                 ...;
      FuseSet {
         Fuse [bitIndex]: AddressPort(<name>) |
               not AddressPort(<name>) |
               LogicHigh | LogicLow;
         .
         . //Repeat for all Fuse bits
         .
      }//End of FuseSet
   }//End of RowSegment
   .
   . //Repeat for all RowSegments
   .
   ColumnSegmentRange {
      SegmentAddress [<bitIndex>]: AddressPort(<name>);
      .
      . //Repeat for all SegmentAddress bits
      .
   }//End of ColumnSegmentRange
   ColumnSegment (<SegmentName>) {
      RowSegmentCountRange [<lowRange>: <highRange>];
      ColumnSegmentCountRange [<lowRange>: <highRange>];
      ShiftedIORange : Data [15:0];
      NumberOfSpareElements : <integer>
                 ...;
      FuseSet {
         Fuse [bitIndex]: AddressPort(<name>) |
               not AddressPort(<name>) |
               LogicHigh | LogicLow;
         .
         . //Repeat for all Fuse bits
         .
         FuseMap [<HighBitRange>:<LowBitRange>] {
            NotAllocated : <bitString>;
            ShiftedIO (<DataPortName>):<bitString>;
            .
            . //Repeat for all IO within ShiftedIORange
            .
         }//End of FuseMap
      }//End of FuseSet
   }//End of ColumnSegment
   .
   . //Repeat for all ColumnSegments
   .
```

```
}//End of RedundancyAnalysis
```

RowSegmentRange and ColumnSegmentRange Wrappers

Redundant row and columns/IO elements can be used simultaneously to replace the defective row and column elements in the memory.

Use the RowSegmentRange and ColumnSegmentRange wrappers to specify the boundary in which the spare elements defined in the RowSegment and ColumnSegment wrappers can be allocated. This segmentation also defines the area where the built-in repair analysis module calculates an optimal repair solution using the spare row and IO/column resources available for this memory region.

The RowSegmentRange wrapper defines the significant row address bits that are used by the RowSegment/RowSegmentCountRange property and the ColumnSegment/ RowSegmentCountRange property. The RowSegmentCountRange property defines the row address boundaries for the corresponding RowSegment or ColumnSegment.

The ColumnSegmentRange wrapper defines the significant column address bits that are used by the ColumnSegment/ColumnSegmentCountRange property. The ColumnSegmentCountRange property defines the column address boundaries for the corresponding ColumnSegment.

The following usage conditions apply:

- RowSegmentCountRange limits specified inside all RowSegment or ColumnSegment wrappers must encompass the entire memory space.

- RowSegment wrappers with different RowSegmentCountRange limits must not overlap.

- ColumnSegmentCountRange limits specified inside all ColumnSegment wrappers must encompass the entire memory space.

- ColumnSegment wrappers with different ColumnSegmentCountRange or RowSegmentCountRange limits must not overlap.

SegmentAddress Property

Use the SegmentAddress property in both the RowSegmentRange and ColumnSegmentRange wrappers to specify which address ports are used to define the range of the memory segment. If only one RowSegment or ColumnSegment wrapper is defined within the RedundancyAnalysis wrapper, the RowSegmentRange or ColumnSegmentRange wrapper accordingly defaults to encompass the entire memory address space.

The following usage conditions apply:

- SegmentAddress[y] — For all specified SegmentAddress properties, y must start from 0 and encompass all valid integers to n-1 where n is the number of the SegmentAddress properties specified.

- *name* must identify a port defined in the memory library file with Port/
Function:Address. If the address port is a vectored port, the name must identify a single
bit within the vectored port.

RowSegment and ColumnSegment Wrappers

Use the RowSegment and ColumnSegment wrappers to define one or more segments of the
memory that has spare elements.

At least one RowSegment and one ColumnSegment wrapper must be specified for row and IO/
column repair analysis.

If the RowSegmentRange wrapper is specified, the number of RowSegment wrappers defined
must encompass the entire memory address space.

NumberOfSpareElements Property

Use the RowSegment/NumberOfSpareElements property to specify the number of spare
elements (banks, rows, or columns) within the defined segment. The default value is 1.

RowSegmentCountRange and ColumnSegmentCountRange Properties

Use the RowSegmentCountRange property in the RowSegment and ColumnSegment wrappers
and use the ColumnSegmentCountRange property in the ColumnSegment wrapper to define the
lowRange and highRange based on the row and column address bits. When a defective element
is within this range, a spare element can be allocated from this segment.
RowSegmentCountRange and ColumnSegmentCountRange can be specified only when at least
one SegmentAddress bit is defined.

FuseSet Wrapper

Use the mandatory FuseSet wrapper to map the fuse register bits to address ports on the
memory. The fuse register bit logs the value of this address port for the defective element. This
wrapper is specified once inside each RowSegment and ColumnSegment wrapper.

- **Fuse Property**

    Use the Fuse property in both the RowSegment/FuseSet and ColumnSegment/FuseSet
    wrappers to define which address bits are required for the fuses used to replace a
    defective element with a spare element. These fuse bits are defined per row segment.
    Constant LogicHigh and LogicLow values also can be captured by the BIRA fuse
    register when a failure is detected.

    The following usage conditions apply:

    o A minimum of one Fuse must be defined per FuseSet.

    o name must either be a scalar port or a single bit of a vectored port.

o   name must be a port defined a Port wrapper with Function: Address in the memory library file. Constant LogicHigh and LogicLow values can be used instead of address bits.

Sample Memory Library File Syntax

An example memory library implementation is provided in this section for a memory with row and IO/column redundancy.

Figure 5-16 illustrates an example of the memory with row and IO/column redundancy.

Figure 5-16. Example Memory With Row and IO/Column Redundancy
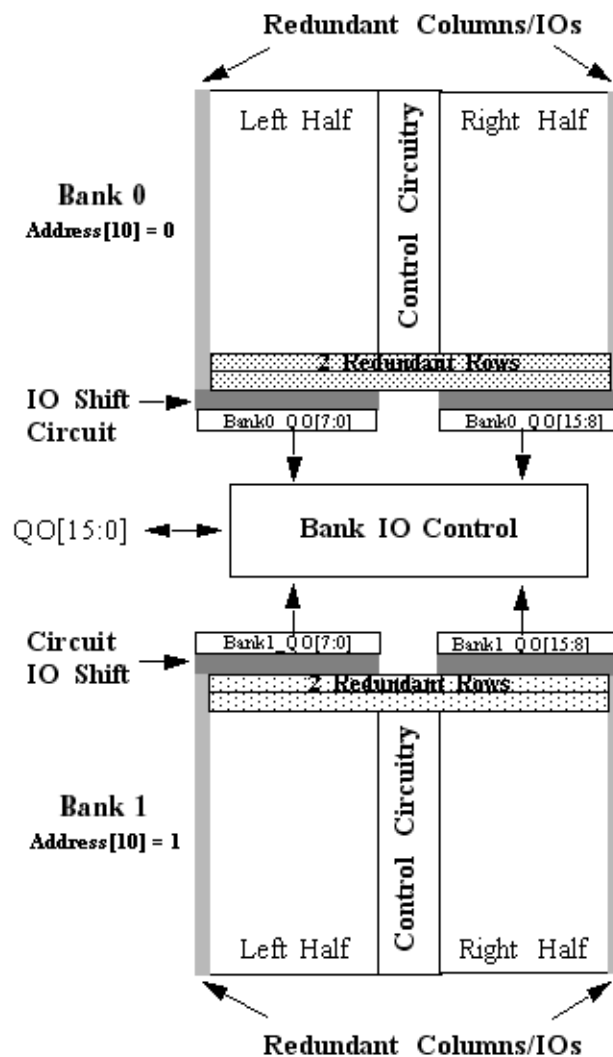
Figure 5-17 illustrates an example of the memory library file syntax to define the repair analysis capability for the example memory in Figure 5-16.

Figure 5-17. Memory Library File Sample Syntax

```
RedundancyAnalysis {
   RowSegmentRange {
      SegmentAddress[0]: AddressPort(Address[10]);
   }
   RowSegment (Bank0){
      NumberOfSpareElements: 2;
      RowSegmentCountRange [1'b0:1'b0]; // Bank 0
      FuseSet {
         Fuse[3]: AddressPort(Address[9]);
         Fuse[2]: AddressPort(Address[8]);
         Fuse[1]: AddressPort(Address[7]);
         Fuse[0]: AddressPort(Address[0]);
      }
   }
   RowSegment (Bank1){
      NumberOfSpareElements: 2;
      RowSegmentCountRange [1'b1:1'b1]; // Bank 1
      FuseSet {
         Fuse[3]: AddressPort(Address[9]);
         Fuse[2]: AddressPort(Address[8]);
         Fuse[1]: AddressPort(Address[7]);
         Fuse[0]: AddressPort(Address[0]);
      }
   }
   ColumnSegment (Bank0_Left){
      RowSegmentCountRange [1'b0:1'b0]; // Bank0
      ShiftedIORange: QO[7:0]; // Left
      FuseSet {
         FuseMap[3:0]{
            ShiftedIO(QO[0]): 4'b0001;
            ShiftedIO(QO[1]): 4'b0010;
            ShiftedIO(QO[2]): 4'b0011;
            ShiftedIO(QO[3]): 4'b0100;
            ShiftedIO(QO[4]): 4'b0101;
            ShiftedIO(QO[5]): 4'b0110;
            ShiftedIO(QO[6]): 4'b0111;
            ShiftedIO(QO[7]): 4'b1000;
         }
      }
   }
   ColumnSegment (Bank0_Right){
      RowSegmentCountRange [1'b0:1'b0]; // Bank 0
      ShiftedIORange: QO[15:8]; // Right
      FuseSet {
         FuseMap[3:0]{
            ShiftedIO(QO[8]): 4'b0001;
            ShiftedIO(QO[9]): 4'b0010;
            ShiftedIO(QO[10]): 4'b0011;
            ShiftedIO(QO[11]): 4'b0100;
            ShiftedIO(QO[12]): 4'b0101;
            ShiftedIO(QO[13]): 4'b0110;
            ShiftedIO(QO[14]): 4'b0111;
            ShiftedIO(QO[15]): 4'b1000;
         }
      }
```

```
      }
      ColumnSegment (Bank1_Left){
         RowSegmentCountRange [1'b1:1'b1]; // Bank 1
         ShiftedIORange: QO[7:0]; // Left
         FuseSet {
            FuseMap[3:0]{
               ShiftedIO(QO[0]): 4'b0001;
               ShiftedIO(QO[1]): 4'b0010;
               ShiftedIO(QO[2]): 4'b0011;
               ShiftedIO(QO[3]): 4'b0100;
               ShiftedIO(QO[4]): 4'b0101;
               ShiftedIO(QO[5]): 4'b0110;
               ShiftedIO(QO[6]): 4'b0111;
               ShiftedIO(QO[7]): 4'b1000;
            }
         }
      }
      ColumnSegment (Bank1_Right){
         RowSegmentCountRange [1'b1:1'b1]; // Bank 1
         ShiftedIORange: QO[15:8]; // Right
         FuseSet {
            FuseMap[3:0]{
               ShiftedIO(QO[8]): 4'b0001;
               ShiftedIO(QO[9]): 4'b0010;
               ShiftedIO(QO[10]): 4'b0011;
               ShiftedIO(QO[11]): 4'b0100;
               ShiftedIO(QO[12]): 4'b0101;
               ShiftedIO(QO[13]): 4'b0110;
               ShiftedIO(QO[14]): 4'b0111;
               ShiftedIO(QO[15]): 4'b1000;
            }
         }
      }
   }
```

This sample syntax of the memory library file specifies to Tessent Shell MemoryBIST the following information for row and IO/column segments:

- The sample memory has two banks (or row segments). Each bank has two spare rows and two spare IOs available for repair. The two spare row elements per bank can repair a failure occurring only within that bank. The FuseSet wrapper defines the address bits to be logged for each repairable element. A single spare IO per column segment dictates the use of two ColumnSegment wrappers per bank. A total of four (4) ColumnSegment wrappers are required — 2 for Bank0 and 2 for Bank1.

- Each of the two row and column segments is located in the address space defined by the AddressPort *Address[10]*. The RowSegment(*Bank0*) and ColumnSegment(*Bank0_xxx*) is defined within the address space whereby *Address[10]* is *1'b0*. The RowSegment(*Bank1*) and ColumnSegment(*Bank1_xxx*) is defined within the address space whereby *Address[10]* is *1'b1*.

- For the row segment range, the SegmentAddress bit 0 is defined as memory address port *Address[10]*.

- Each row segment consists of two spare elements as defined by the NumberOfSpareElements property. This means that for RowSegment(*Bank0*), there are two sets of fuses, each of which logs the value of the ports *Address[9:7]* and *Address[0]* for the defective portion of memory within the row segment. Similarly for RowSegment(*Bank1*), there are two sets of fuses, each of which logs the value of the ports *Address[9:7]* and *Address[0]* for the defective portion of memory within the row segment.

- Each ColumnSegment has FuseMap registers. The FuseMap registers capture the specified ShiftedIO fuse map value that indicates the IO on which an error was detected. FuseMap is mandatory when implementing IO/column repair. The fuse map values are used to identify the faulty IO of a memory. The fuse map values are found in the memory data sheet provided by the memory vendor.

Repair Analysis Registers

When built-in repair analysis for row and IO/column is implemented using Tessent Shell MemoryBIST, a set of registers is implemented in hardware for latching the results of the memory BIST execution.

This set of registers is provided for each memory segment defined in the memory library file(s):

- One segment Repair Status Register

- One or more Fuse Registers

The repair status and fuse registers are located on a dedicated BIRA setup chain. Therefore, these registers can be serially scanned out of either, or both, the memory controller and interface.

Repair Status Register

In the hardware, the overall ability of the memory to be repaired is reported by the repair status register. Each memory has a repair status register named:

```
<memory_instance_name>_REPAIR_STATUS
```

The repair status register specifies if the memory requires repair, is not repairable, or does not require repair (when no failure is detected). The bit decode assignments for the repair status register are described in Table 5-5.

Table 5-5. <MemoryInstance>_STATUS_REG Decodes

Bit1 Bit0	Repair Status
00	No Repair Required
01	Repair Required
1x	Not Repairable

The repair status register(s) are located in the setup chain. Therefore, these registers can be serially scanned out of the memory controller or memory interface.

Fuse Registers

For each memory segment, two registers are required for each spare element:

- Allocation Register (if the NotAllocated property is not specified)

- FuseSet Register

Allocation Register

The allocation register, if present, contains a bit that specifies whether the spare element is allocated (needs repair) or is not allocated. This register has the following name:

```
<memory_instance_name>_<segment_name>_SROW#_ALLOC_REG   // for row spares
<memory_instance_name>_<segment_name>_SCOL#_ALLOC_REG   // for col spares
```

A value of 0 in this bit represents NOT allocated (no repair required), and a 1 represents allocated (repair required). If allocated, the value in the FuseSet register holds the memory address that needs to be repaired.

FuseSet Register

The FuseSet register contains the fuse bits as specified by the FuseSet wrapper for the segment. The register is composed of the following:

- The IO map value corresponding to the defective element.

- If the Fuse property in the FuseSet wrapper for the respective memory row or column segment is defined, then the FuseSet register also includes the address value driven on the memory address bus for the defective element.

Repair Analysis Output Ports for IO/Column and Row Repair

Refer to the following sections for information on repair analysis output ports for IO/Column and Row repair:

- "Repair Analysis Output Ports for IO/Column Repair"

- "Repair Analysis Output Ports for Row Repair"

Built-In Self-Repair

There are two types of memory repair interfaces—parallel and serial. Each memory can use only one type of repair interface. A memory with a parallel repair interface has ports to allocate redundant elements. A memory with a serial repair interface has an internal shift register that is used to allocate the redundant elements.

In the Memory Tessent Core Description (TCD) or memory library file, you specify BISR details using properties inside the PinMap/SpareElement wrapper as described in the "Parallel BISR Interface" and "Serial BISR Interface" sections.

Parallel BISR Interface

When implementing a parallel BISR interface, you use the Fuse, FuseMap, and RepairEnable properties of the PinMap/SpareElement wrapper in the memory library file to specify the memory repair port name associated with the specified fuse register.

The properties of the PinMap/SpareElement wrapper provide guidance on how to connect the BISR fuse registers to the memory repair ports, as described in later sections.

Serial BISR Interface

A memory with a *serial* repair interface has an internal shift register that is used to allocate the spare elements.

When implementing the serial BISR interface, you must describe the order of the internal BISR register using RepairRegister[x] indexes in the Fuse, FuseMap, and RepairEnable properties of the PinMap/SpareElement wrapper in the memory library file.

The total BISR length (N) for a given memory is equal to the number of the RepairRegister[x] properties specified in the RowSegment and ColumnSegment wrappers combined inside a memory library file. RepairRegister[0] specifies the bit of the BISR register that is closest to the BISR scanOut port and its value is scanned in first, and RepairRegister[N-1] specifies the bit of the BISR register that is closest to the BISR scanIn port and its value is scanned in last.

A single BISR register is generated for each memory. The RepairRegister[x] indexes must be contiguous from 0 to N-1 within a memory template. All indexes between 0 and N-1 must be used, and each index can be used only once.

Be careful when describing the RepairRegister[*x*] indexes because they must match the internal memory BISR register order. The memory data sheet provides the internal BISR register ordering. Any mismatch between the order of the internal BISR chains and the order specified by the data sheet results in incorrect repair data scanned inside the memory. It is recommend that you run the memory template through the memory library certification tool (*memlibc*) to validate the memory template. This certification tool combined with fault-inserted memory simulations quickly identifies any mismatches between the memory template and the internal BISR register in the memory model. Refer to "Certifying TCD Memory Library Files With memlibCertify in Tessent Shell" for more information on this topic.

If a built-in retiming latch exists at the output of the internal BISR chain, use the Retimed property in the Port wrapper to indicate the presence of the retiming latch.

BISR-Specific Memory Library Syntax

The section provides an example memory library file that shows the use of the Port wrapper to declare memory repair ports, as well as the use of the PinMap wrapper to specify the mapping of the repair solution to the memory.

Figure 5-18 summarizes the syntax of the example memory library file used to support built-in self-repair.

Figure 5-18. Repair Support in Memory Library File

```
Port (<portName>) {
    Function: BisrParallelData | BisrSerialData |
            BisrClock | BisrReset | BisrScanEnable;
    Direction: Input | Output;
    ReTimed: On | (Off);
}
RedundancyAnalysis {
    RowSegmentRange {
        ...
    }
    ColumnSegmentRange {
        ...
    }
    RowSegment (<segmentName>){
        ...
        PinMap {
            SpareElement {
                RepairEnable : <repairPortName> |
                    RepairRegister[bitIndex];
                Fuse [<bitIndex>]: <repairPortName> |
                    RepairRegister[bitIndex];
                LogicLow: <repairPortName> |
                    RepairRegister[bitIndex];
            }
            .
            . / Repeat for all SpareElements
            .
        }
    }
    .
    . /Repeat for all RowSegments
    .
    ColumnSegment (<SegmentName>) {
        ...
        PinMap {
            SpareElement {
                FuseMap [<bitIndex>]: <repairPortName> |
                    RepairRegister[bitIndex];
                RepairEnable: <repairPortName> |
                    RepairRegister[bitIndex];
                Fuse [<bitIndex>]: <repairPortName> |
                    RepairRegister[bitIndex];
                LogicLow: <repairPortName> |
                    RepairRegister[bitIndex];
            }
        }
    }
}
```

Port Wrapper

The Port wrapper in the memory library file is used to declare the memory repair ports on a repairable memory.

Depending on the type of repair interface that is used in a memory, repair ports are specified with the following port functions, as specified in the Function property of the Port wrapper:

- For *Parallel* BISR interface:

    o BisrParallelData

- For *Serial* BISR interface:

    o BisrSerialData (with Port/Direction : input | output;)

    o BisrClock (with Port/Polarity : (activeHigh) | activeLow;)

    o BisrReset (with Port/Polarity : (activeHigh) | activeLow;)

    o BisrScanEnable (with Port/Polarity : (activeHigh) | activeLow;)

> _____ **Note** _____
>
> All ports with Function:Select are intercepted (gated) when a serial BISR interface is used. This forces the memory to be deselected when shifting through the internal BISR chain.

PinMap SpareElement Wrapper

The contents of the PinMap wrapper enables you to apply the repair solution, computed by the BIRA engine, to the memory repair ports or the serial repair register of the memory.

The PinMap/SpareElement wrapper specifies the mappings, or connections, from the BISR fuse registers to the corresponding memory repair ports. Tessent Shell MemoryBIST generates and instantiates self-repair hardware and connections for a memory when the DftSpecification MemoryInterface/repair_analysis_present property is "auto" (the default setting) and the memory library file for the memory contains a RedundancyAnalysis wrapper. The contents of the PinMap:SpareElement wrapper are interpreted differently depending on the type of repair interface used, as explained in the following sections.

Parallel Memory Repair Interface

Using the PinMap/SpareElement wrapper, you can specify mappings from the BISR fuse registers to the corresponding memory repair ports. These pin mappings directly connect the BISR fuse register ports to the memory repair ports.

Serial Memory Repair Interface

Using the PinMap/SpareElement wrapper, you must specify the order of the internal BISR register. Tessent Shell MemoryBIST builds the corresponding external BISR register according to this order. The RepairRegister[*x*] index specifies the location of each fuse in the external BISR register.

BISR Usage Conditions

When using the BISR feature, you must declare the PinMap/SpareElement wrapper(s) in the appropriate RowSegment or ColumnSegment wrappers.

- When specified inside either the RowSegment or ColumnSegment wrappers, the PinMap wrapper may contain multiple SpareElement wrappers. The number of PinMap/SpareElement wrappers inside the RowSegment wrapper must be equal to the NumberOfSpareElements property.

You also can specify the RepairEnable, Fuse, FuseMap, and LogicLow properties in the PinMap/SpareElement wrapper as described in the following sections. The repair interface determines which usage conditions apply:

- For the Parallel Repair Interface:

  o The *repairPortName* must be either a scalar port or a single bit of a vectored port.

  o The *repairPortName* must be a port defined in the memory library file corresponding to a Port/Function: BisrParallelData setting.

- For the Serial Repair Interface:

  o The RepairRegister[*x*] value must be specified where *x* is the index of the Repair Enable register inside the BISR chain.

  o The following Port/Function properties must be declared in the memory library file:

    - BisrSerialData (with Port/Direction: input | output;)

    - BisrClock (with Port/Polarity: (activeHigh) | activeLow;)

RepairEnable Property

Use the RepairEnable property inside the PinMap/SpareElement wrapper to specify the port of the memory or the BISR chain register that enables the allocation of a repair element.

Fuse Property

Use the Fuse [*<bitIndex>*] property inside the PinMap/SpareElement wrapper to specify the port of the memory or the BISR register that controls the address of a spare row or column element. You can use this property multiple times. This property maps each bit in FuseSet/Fuse to the corresponding repair ports on the memory. Each PinMap/SpareElement wrapper index must match a FuseSet/Fuse index.

FuseMap Property

Use the FuseMap [<*bitIndex*>] property inside the PinMap/SpareElement wrapper to specify the memory port that controls the port of a spare IO element. This property can be specified only inside the ColumnSegment wrapper and must be repeated for each FuseMap bit. This property maps the shifted IO fuse map bits to the corresponding repair ports on the memory. Each PinMap/SpareElement/FuseMap index must be within the FuseSet/Fuse[*x:y*] index range.

LogicLow Property

Use the LogicLow property inside the PinMap/SpareElement wrapper to insert a BISR register that is not connected to any BIRA output in the BISR chain. This BISR register captures a constant value during a BIRA-to-BISR transfer if BIRA is present. When no BIRA is present for this memory, the register holds its value during a BIRA-to-BISR transfer operation.

When using a memory with a parallel BISR interface, the memory repair port that this BISR is connected to must be specified.

When using a memory with a serial BISR interface, the index of this BISR register must be specified using RepairRegister[<*bitIndex*>].

Memory Library Examples With Parallel and Serial BISR Interface

The sections below demonstrate two simple examples of repairable memories with their corresponding memory library file:

- "Row Repair and Parallel BISR Interface"

  This example shows a memory with two redundant rows using a parallel BISR interface. The redundant rows are allocated using the REN0, RR0[3:0], REN1, RR1[3:0] ports on the memory module.

- "Column Repair and Serial BISR Interface"

  This example shows a memory with one redundant column using a serial BISR interface. The redundant column allocation is accessed using the serial BISR ports on the memory module.

Row Repair and Parallel BISR Interface

The example shown in the figure below is of a memory with two spare rows using a parallel BISR interface.

Figure 5-19. Example Memory With Two Spare Rows Using Parallel BISR Interface
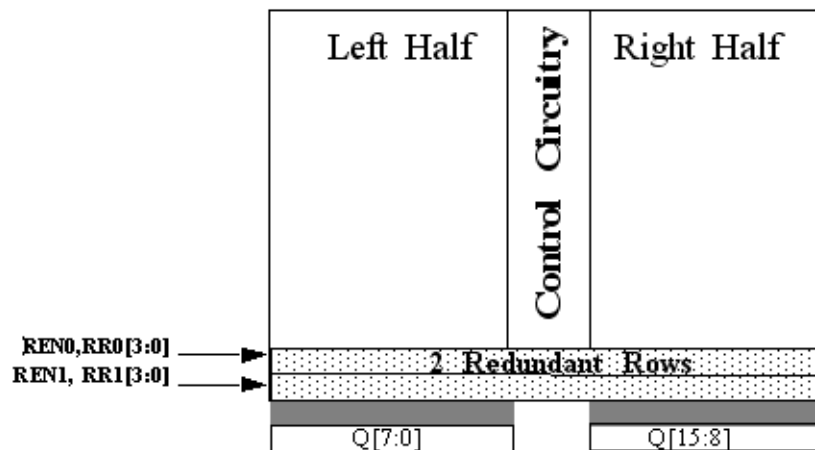
Figure 5-20 illustrates the memory library file fragment associated with the memory illustrated in Figure 5-19.

Figure 5-20. Memory Library File Fragment for Two Spare Rows Using Parallel BISR Interface

```
// MemoryRepair ports for Bank B0
Port(REN0) {
    Function: BisrParallelData;
    Direction: Input;
}
Port(RR0[3:0]) {
    Function: BisrParallelData;
    Direction: Input;
}
Port(REN1) {
    Function: BisrParallelData;
    Direction: Input;
}
Port(RR1[3:0]) {
    Function: BisrParallelData;
    Direction: Input;
}
RedundancyAnalysis {
    RowSegment (My_RowSeg1){
        NumberOfSpareElements: 2;
        FuseSet {
            Fuse[3]: AddressPort(Address[9]);
            Fuse[2]: AddressPort(Address[8]);
            Fuse[1]: AddressPort(Address[7]);
            Fuse[0]: AddressPort(Address[0]);
        }
        PinMap {
            SpareElement {
                RepairEnable: REN0;
                Fuse[0]: RR0[0];
                Fuse[1]: RR0[1];
                Fuse[2]: RR0[2];
                Fuse[3]: RR0[3];
            }
            SpareElement {
                RepairEnable: REN1;
                Fuse[0]: RR1[0];
                Fuse[1]: RR1[1];
                Fuse[2]: RR1[2];
                Fuse[3]: RR1[3];
            }
        }
    }
}
```

Column Repair and Serial BISR Interface

The example in the figure below shows a memory with one spare column using a serial BISR interface.

Figure 5-21. Example Memory With One Spare Column and Serial BISR Interface
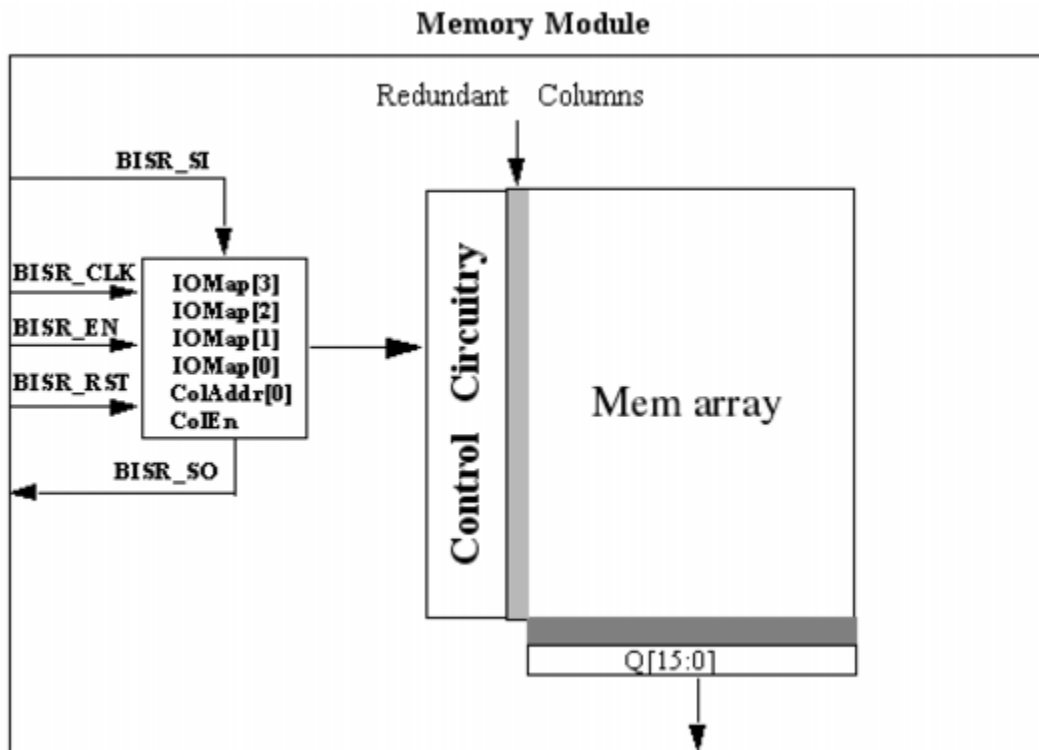
Figure 5-22 illustrates the memory library file fragment associated with the memory illustrated in Figure 5-21.

Figure 5-22. A Fragment of the Memory Library File Sample Syntax

```
// MemoryRepair ports for Bank B0
Port(BISR_CLK) {
    Function: BisrClock;
    Direction: Input;
}
Port(BISR_RST) {
    Function: BisrReset;
    Direction: Input;
}
Port(BISR_SI) {
    Function: BisrSerialData;
    Direction: Input;
}
Port(BISR_SO) {
    Function: BisrSerialData;
    Direction: Output;
}
RedundancyAnalysis {
    ColumnSegment (My_ColSeg){
        ShiftedIORange: QO[15:0];
        FuseSet {
            Fuse[0]: Address[10];
            FuseMap[3:0]{
                ShiftedIO(QO[0]): 4'b0000;
                ShiftedIO(QO[1]): 4'b0001;
                ShiftedIO(QO[2]): 4'b0010;
                ShiftedIO(QO[3]): 4'b0011;
                ShiftedIO(QO[4]): 4'b0100;
                ShiftedIO(QO[5]): 4'b0101;
                ShiftedIO(QO[6]): 4'b0110;
                ShiftedIO(QO[7]): 4'b0111;
                ShiftedIO(QO[8]): 4'b1000;
                ShiftedIO(QO[9]): 4'b1001;
                ShiftedIO(QO[10]): 4'b1010;
                ShiftedIO(QO[11]): 4'b1011;
                ShiftedIO(QO[12]): 4'b1100;
                ShiftedIO(QO[13]): 4'b1101;
                ShiftedIO(QO[14]): 4'b1110;
                ShiftedIO(QO[15]): 4'b1111;
            }
        }
        PinMap {
            SpareElement {
                Fuse[0]: RepairRegister[1];
                RepairEnable: RepairRegister[0];
                FuseMap[0]: RepairRegister[2];
                FuseMap[1]: RepairRegister[3];
                FuseMap[2]: RepairRegister[4];
                FuseMap[3]: RepairRegister[5];
            }
        }
    }
}
```

Implementing BIRA and BISR Logic

This section shows how to insert repair logic in your circuit. The emphasis is on the insertion of the BISR logic. BIRA logic is generated for a memory instance at the same time as the rest of the memory BIST logic (controller and interface) as long as the RepairAnalysis wrapper is present in the memory library file and that the repair_analysis_present property is not set to "Off" for that instance.

The primary BISR logic insertion task are as follows:

1. Inserting BISR chains in a block

2. Connecting a BISR controller to existing BISR chains

3. Connecting a BISR controller to an external fuse box

> _____ **Note** _____
> A fuse box is not needed if the soft repair method is implemented. Refer to the "Implementing Soft Repair" section for more information.

4. Connecting a BISR controller to system logic

The first two tasks are used in a bottom-up design methodology where BISR chains are inserted in circuit blocks (or cores) before being connected at the chip top level to the BISR controller.This is the most frequently used method for inserting the repair logic.

Inserting BISR Chains in a Block

BISR chains are automatically inserted if memories instantiated in the design have spare resources described in their memory library file as described in the Memory library. BISR registers associated to repairable memories are connected together to form scan chains.

Assigning Memories to Power Domains

By default, all repairable memories are part of the same power domain and a single BISR chain is created. However, if more than one power domain exists, multiple BISR chains are required. The power domain of a memory instance is determined by its bisr_power_domain_name attribute that can be specified in two ways.

The preferred method consists of reading a UPF or CPF file corresponding to the design using the read_upf or read_cpf command. The file should be loaded in Tessent Shell during the setup. For example, the command

```
SETUP> read_upf moda.upf
```

reads the UPF file corresponding to design moda and automatically sets the attribute for all memories.

The alternative method consists of manually setting the bisr_power_domain_name attribute using the set_attribute_value command. For example, the command

```
SETUP> set_attribute_value {memA_inst mem6_inst} -name
bisr_power_domain_name -value pdgA
```

assigns memory instances memA_inst and mem6_inst to a power domain labeled pdgA.

_____**Note**_____

The value of "none" is a reserved string and you cannot specify it as a power domain group label. Refer to the PatternsSpecification MemoryBisr/Controller/ power_domain_group_labels property description for more information.

For sub-blocks or physical blocks containing repairable memories, there can be one or more BISR scan interfaces in place, and setting the bisr_power_domain_name attribute on the sub or physical block instance does not assign the contained BISR chains to the specified

PowerDomainGroup. In this case, the attribute must be set on the BISR_SI pin of each BISR chain you want to assign to a particular PowerDomainGroup. For example, the command

```
SETUP> set_attribute_value {core/blockA/pdgA_bisr_si} -name
bisr_power_domain_name -value pdgA
```

does assign the BISR chain with the pdgA_bisr_si input pin within the "blockA" sub or physical block, to PowerDomainGroup "pdgA". Any remaining BISR chains within blockA are associated with the PowerDomainGroup specified on the ICL pin attribute for this child block. The default power domain group name for the BISR_SI ICL pins are the ones used during process_dft_specification for the child block. The BISR_SI power domain group names are saved as pin attributes inside the ICL for the child block when extract_icl is performed.

The manual specification method can be used to create a logical sub-division of large power domains and accelerate repair for a subset of memories in each domain during system power up.

Controlling the BISR Chain Order

BISR chains are connected according to the content of the BisrSegmentOrderSpecification wrapper containing a list of memory instances defining the BISR chain order. When a DEF file is provided, the BISR segments are ordered using an algorithm that optimizes the routing based on the memory coordinates. If a DEF file is not provided, the memories are sorted alphabetically within each power domain group.

The wrapper is contained in a file named *<design_name>.bisr_segment_order* that is automatically generated in the current directory when entering the analysis mode, or when you run the create_bisr_segment_order_file command. Analysis mode is automatically entered when check_design_rules is performed without any DRC errors occurring.

Figure 5-1 shows the contents of a chip-level BISR segment order file with three power domains. The default power domain is assigned to all repairable memories that are not explicitly assigned to a domain. It is often associated to the portion of the design whose power is never switched off.

Example 5-1. Example Chip-Level BISR Segment Order File

```
//-----------------------------------------------------
//  File created by: Tessent Shell
//         Version:
//       Created on:
//-----------------------------------------------------

BisrSegmentOrderSpecification {
PowerDomainGroup(-) {
    OrderedElements {
      memA;
    }
  }
```

```
PowerDomainGroup(pd_A) {
    OrderedElements {
      blockA_clka_i1/pd_A_bisr_si;
      blockA_clka_i2/pd_A_bisr_si;
      blockB_clka_i1/bisr_si;
    }
  }
  PowerDomainGroup(pd_B) {
    OrderedElements {
      blockA_clka_i1/pd_B_bisr_si;
      blockA_clka_i2/pd_B_bisr_si;
    }
  }
}
```

You can affect the BISR chain ordering in two ways if the default BISR chain order is not satisfactory. In the preferred method you read in a DEF file corresponding to the design using the read_def command, and create BISR segment ordering that MemoryBIST automatically optimizes based on memory placement and physical block or sub-block BISR_SI pin coordinates. The physical blocks and sub-blocks have MemoryBIST insertion and BISR chain ordering done separately. You should load the file in Tessent Shell during the setup mode at the same time you read in the other design files. For example, the command

```
SETUP> read_def modA.def
```

reads the DEF file corresponding to design modA. MemoryBIST determines the BISR chain ordering based on the memory placement and block BISR_SI pin placement information found in the DEF file. If a block's BISR_SI pin does not specify the coordinates, but the parent block does, MemoryBIST uses the parent block coordinates for ordering optimization.

You can also independently override the placement of BISR registers and associated pipeline registers using the BisrElement and BisrElement/Pipeline wrappers and specify their insertion into a particular parent design instance. The tool determines the BISR chain ordering for these registers based on the parent design BISR_SI pin placement information found in the DEF file.

The second way consists of manually modifying the order of memory instance names within the *<design_name>.bisr_segment_order* file before running the process_dft_specification command. If check_design_rules is re-run either during the same Tessent Shell invocation or a subsequent one, and you want to reuse the modified *<design_name>.bisr_segment_order* file, you must specify the set_dft_specification_requirements while still in setup mode. For example, the command:

```
SETUP> set_dft_specification_requirements -bisr_segment_order_file
modA.bisr_segment_order
```

indicates that the file *modA.bisr_segment_order* should be preserved and used to determine the BISR chain order when the next create_dft_specification command is run. The file name of the BISR segment order file is arbitrary and does not need to be the default name.

The process_dft_specification command creates an annotated BISR segment order file with a "*.annotated*" suffix in the current directory. An additional copy of the annotated file is also stored in the following file within the tsdb_outdir:

```
<tsdb_outdir>/instrument/<design_name><design_id>_rtl_mbisr.instrument/
<design_name><design_id>_mbisr.bisr_segment_order
```

The annotated file in the current directory is intended for you to use if BISR chain re-ordering is needed. The BISR segment order file in the TSDB documents the implemented BISR chain and is intended as a reference.

The annotated file contains comments that indicate repair group membership and BISR chain register lengths. For example, the section below shows the annotated version of the BISR segment order file shown in Figure 5-1:

```
//----------------------------------------------------
// File created by: Tessent Shell
// Version:
// Created on:
//----------------------------------------------------

BisrSegmentOrderSpecification {
  PowerDomainGroup(-) {
    // ${controller_inst}/toBisr --> {OrderedElement …} -->
    //    ${controller_inst}/fromBisr
    OrderedElements {
      memA; // RepairGroup:None   BISRLength:10
    }
  }
  PowerDomainGroup(pd_A) {
    // ${controller_inst}/toBisr_pd_A --> {OrderedElement …} -->
    //    ${controller_inst}/fromBisr_pd_A
    OrderedElements {
      blockA_clka_i1/pd_A_bisr_si;
      blockA_clka_i2/pd_A_bisr_si;
      blockB_clka_i1/bisr_si;
    }
  }
  PowerDomainGroup(pd_B) {
    // ${controller_inst}/toBisr_pd_B --> {OrderedElement …} -->
    //    ${controller_inst}/fromBisr_pd_B
    OrderedElements {
      blockA_clka_i1/pd_B_bisr_si;
      blockA_clka_i2/pd_B_bisr_si;
    }
  }
}
```

The first comment line in the PowerDomainGroup wrapper helps you to understand the BISR chain order. It shows that the BISR chain scan input shifts into the OrderedElement items in the order listed from top to bottom, and finally to the BISR chain scan output.

Additionally, each memory instance within the current design has a comment showing any applicable membership within a repair sharing group, as well as the length of the BISR register associated with that memory or repair sharing group. In the example above, memA is not part of any repair sharing group and the BISR register associated with memA has a length of 10. An example of an annotated physical block BISR segment order file with repair sharing is shown in the "BISR Segment Order File" topic.

Turning Off the Insertion of BISR Registers

It is possible to disable the generation of BISR registers for specific memory instances by running a set_memory_instance_option command with the use_in_memory_bisr_dft_specification option set to off.

```
SETUP> set_memory_instance_option blockA_clka_i1/mem4
use_in_memory_bisr_dft_specification off
```

The example above disables the generation of the BISR register for memory instance mem4, even if the memory has spare resources. However, this option is rarely used.

You can also disable BISR insertion in the design by using the following command:

```
set_dft_specification_requirement -memory_bisr_chains off
```

Excluding Child Block BISR Chains

When a parent design does not implement memory repair, yet integrates sub-blocks or physical blocks that already contain BISR chains, the chains must not be connected or used and be properly tied off. The method outlined below is used when the alternative of re-running the memory BIST DFT flow on the child blocks, with BISR chain insertion disabled, is not feasible.

The first step in excluding a child block BISR chain is to remove the chain connection from the *<design_name>.bisr_segment_order* file before the create_dft_specification command is run. This file is originally created when check_design_rules is run. If check_design rules is subsequently re-run and you want to reuse the modified *<design_name>.bisr_segment_order* file, you need to specify the set_dft_specification_requirements when still in setup mode. For example, the command:

```
SETUP> set_dft_specification_requirements -bisr_segment_order_file
modA.bisr_segment_order
```

indicates that the file modA.bisr_segment_order should be preserved and used to determine the BISR chain contents and order when the next create_dft_specification command is run. Note that the file name of the bisr_segment_order file is arbitrary.

The second step is to use the set_attribute_value command to add an "allowed_no_destination connection_rule_option to the pin, as shown in this example:

```
set_attribute_value [get_icl_ports pd_B_bisr_so -of_modules core]
    -name connection_rule_option -value allowed_no_destination
```

The second step also safeguards against accidental removal of BISR chain connections from the .bisr_segment_order file with the unintentional result of having a portion of the design not allowing repair. If the second step is not completed, an ICL extraction error results and points out that the BISR chain is not accessible.

The two steps outlined above are repeated as needed for each child block BISR chain that is to be excluded.

Generic Fuse Box

Several vendors provide fuse boxes, and they have a large number of interfaces and protocols that are not common to each other. Because of this complexity, the fuse box read and write protocols are encapsulated inside of a fuse box interface.

The interface is custom-designed for each fuse box. However, this is a one-time effort for a given technology. Also, Siemens EDA can provide this interface for certain technologies or design a new interface under certain conditions.

The fuse box can be instantiated inside the BISR controller or can be external to the BISR controller. The location is controlled with the MemoryBisr Controller/fuse_box_location property in the DftSpecification. The recommended design flow is to incorporate a vendor provided fuse box and interface prior to processing the DftSpecification, for either internal or external fuse box locations. Note that this is a requirement for a fuse box located externally. When the recommended design flow is not followed for an internal fuse box instantiation, the tool generates and uses a generic fuse box interface module that is valid for basic simulation purposes only.

The fuse box module is specified as follows:

- Internal Fuse Box

  When the fuse box is internal to the BISR controller, and only one TCD FuseBoxInterface wrapper (tcd_fusebox) is present, the DftSpecification fuse_box_interface_module property is inferred from the tcd_fusebox if the fuse_box_interface_module property is not specified. If multiple tcd_fusebox library files are present the fuse_box_interface_module property must be specified.

  If the recommended design flow is not followed in that a tcd_fusebox is not present, and the fuse_box_interface_module is not specified in the DftSpecification, a generic fuse box interface model is generated to include a generic fuse box that encapsulates specific fuse box read and write protocols. The generic interface and fuse box models are generated as template files and are used only for simulation purposes.

- External Fuse Box

  When the fuse box is located outside the BISR controller, a fuse box interface must be instantiated in the design to provide a valid fuse box read and write interface to the BISR controller. Generally, external fuse boxes are used when the fuse box is shared for purposes other than memory repair. Refer to "Connecting the BISR Controller to an External Fuse Box" for further information.

The following sections provide detailed information on:

Fuse Box Interface Signals

The interface between the BISR controller and the fuse box interface is simple.

If Controller/fuse_box_location is set to external in the DftSpecification, then connections are made between the fuse box interface and the BISR controller. If fuse_box_location is set to internal, and no fuse box interface is specified in a tcd_fusebox wrapper or DftSpecification fuse_box_interface_module property, the BISR controller instantiates the generic fuse box interface module and the port names must match the port names generated in the following file within the TSDB:

```
tsdb_outdir/instruments/<design_name>_generic_fusebox_interface.v
```

Table 5-6 lists all fuse box interface ports that the BISR controller uses to communicate with the fuse box interface. This table also provides the interface input/output pins, their corresponding generic fuse box names, and descriptions. All control signals have an active high port polarity.

Fuse Box Protocol

Two operations are common to all fuse boxes: Read and Write. Two additional operations are specific to non-addressable fuse boxes: Transfer and Program.

The TCD FuseBoxInterface/programming_method property or DftSpecification AdvancedOptions/FuseBoxOptions/programming_method must be set to buffered for non-addressable fuse boxes. An additional signal, programFB, is also connected to the fuse box interface.

Figure 5-23 shows the BISR controller fuse box access protocol during a Read operation cycle, and Figure 5-24 shows the BISR controller fuse box access protocol during a Write operation cycle. These figures do not show the programFB signal, but if present, its value is assumed to be 0.

With the exception of the doneFB and fuseValue signals, the signals identified in the Generic Fuse Box Port Name column in Table 5-6 are generated by the BISR controller. The doneFB and fuseValue outputs are generated inside the generic fuse box interface module. All BISR controller outputs (selectFB, writeFB, FBAccess, address) change on the rising edge of the clock. The doneFB and fuseValue outputs are sampled on the rising edge of clock if the fuse box is instantiated within the BISR controller (fuse_box_location:internal). If not (fuse_box_location:external), then doneFB and fuseValue are sampled on the falling edge of clock.

> **Note**
> The fuse box interface asserts a logic high on doneFB for only one cycle and then returns to a logic low.

When the fuse box is internal to the BISR controller, the clock used by the fuse box interface can be balanced with the clock of the controller. In this case, all flops can be updated and sampled synchronously on the rising edge, yielding a higher speed of operation.

When the fuse box is external to the BISR controller the clocks might not be balanced. The controller uses a slightly different timing where outputs still change on the rising edge, but inputs (such as doneFB and fuseValue) are sampled on the falling edge to account for potential skew between the controller and the fuse box interface. A lower speed of operation is possible in this case.

Figure 5-23 shows two variations of the waveforms that depend on the value of the TCD FuseBoxInterface/align_access_en_with_address or the DftSpecification AdvancedOptions/ FuseBoxOptions/align_access_en_with_address property. If set to on, the FBAccess pulse aligns in the first cycle with the address to be read. If set to off, the FBAccess pulse occurs one cycle ahead of the address to be read, allowing a slightly faster fuse box access. A setting of auto defaults to on if fuse_box_location is set to external, and to off if fuse_box_location is set to internal. Another difference is that the FBAccess pulse can overlap with the doneFB pulse of a previous access. In Figure 5-23, FBA2 occurs in the same cycle as DFB1.

The tool will force the align_access_en_with_address property to on when you enable hard incremental repair by setting the DftSpecification max_fuse_box_programming_sessions property to a value greater than 1. When hard incremental repair is performed, the BISR controller asserts selectFB and reads the test insertion flags from the fuse box in the first clock cycles after selection (refer to Figure 5-49 for the fuse box organization). These first operations may not meet the minimum timing margin requirements for some eFuse interfaces. Aligning the FBAccess and Address signals provide the extra timing margin between the selectFB and FBAccess signals that ensure interoperability with the current eFuse interface circuits. When you do not implement hard incremental repair, the BISR controller performs the first eFuse accesses several clock cycles after it asserts selectFB. In this case, the align_access_en_with_address property does not need to be enabled as the eFuse timing requirements between the selectFB and FBAccess signals are met.

Figure 5-23. BISR Controller Fuse Box Read Access Protocol
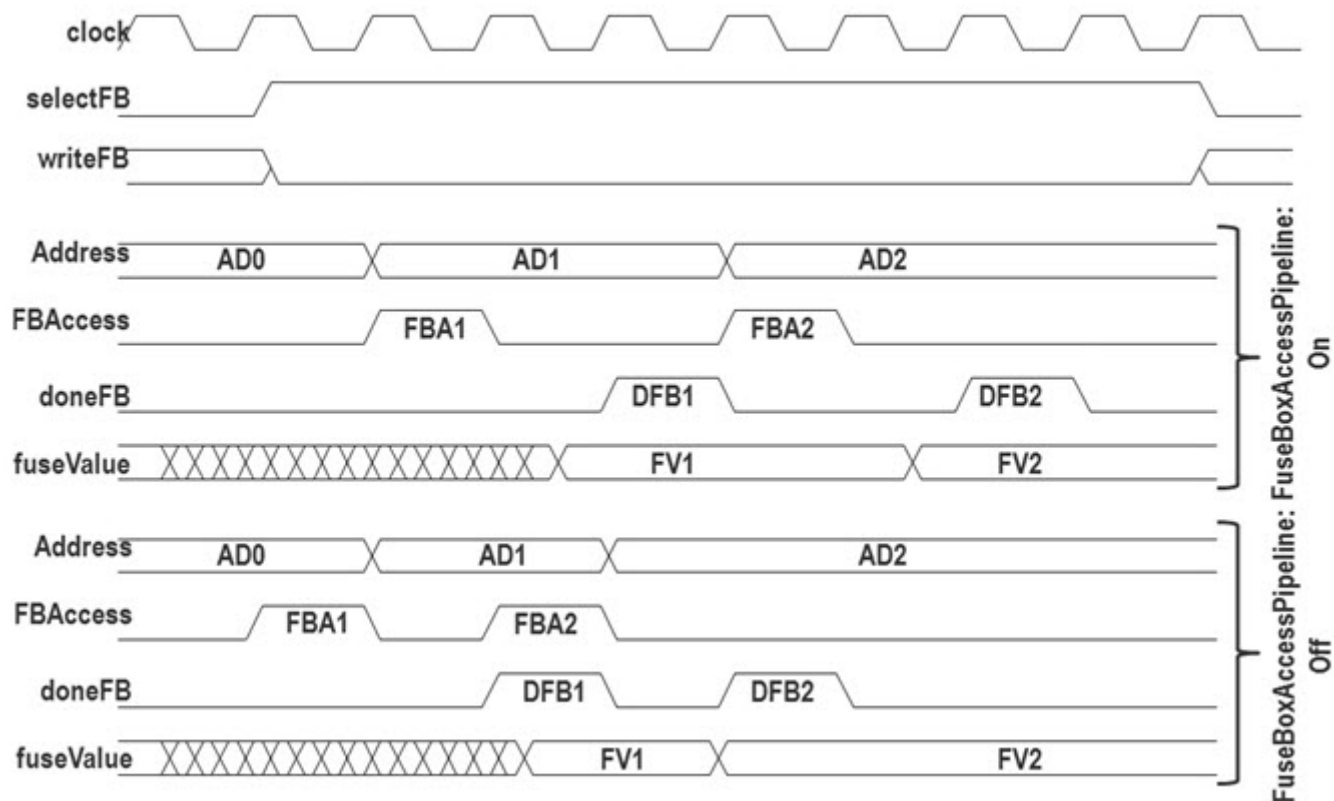
Figure 5-24 shows the waveforms of a Write operation for the two programming methods: unbuffered, which is the default, and buffered. The method is selected by specifying the FuseBoxInterface/programming_method or DftSpecification AdvancedOptions/ FuseBoxOptions/programming_method property. A Write operation using the unbuffered method takes much longer than for the buffered method. The reason is that a fuse is actually blown in the fuse box, and typical programming times are in the microseconds. The fuse box programming voltage (vddq) is applied during this operation. The buffered method only writes to registers in the fuse box interface, and the operation can be performed in two clock cycles. The programming voltage is not applied during this operation.

The selectFB signal is set high when the BISR controller is enabled. The writeFB signal is set high to indicate that the BISR controller is preparing a fuse box Write operation cycle. Then the BISR controller sets the Address port (the fuse address that needs to be written to) and asserts the FBAccess port high for one cycle. This port initiates the Write operation cycle. The BISR controller keeps the Address, writeFB, and selectFB signals stable until the generic fuse box interface sends a logic one on the doneFB output port, which indicates that the Write operation cycle has completed.

Figure 5-24. BISR Controller Fuse Box Write Access Protocol
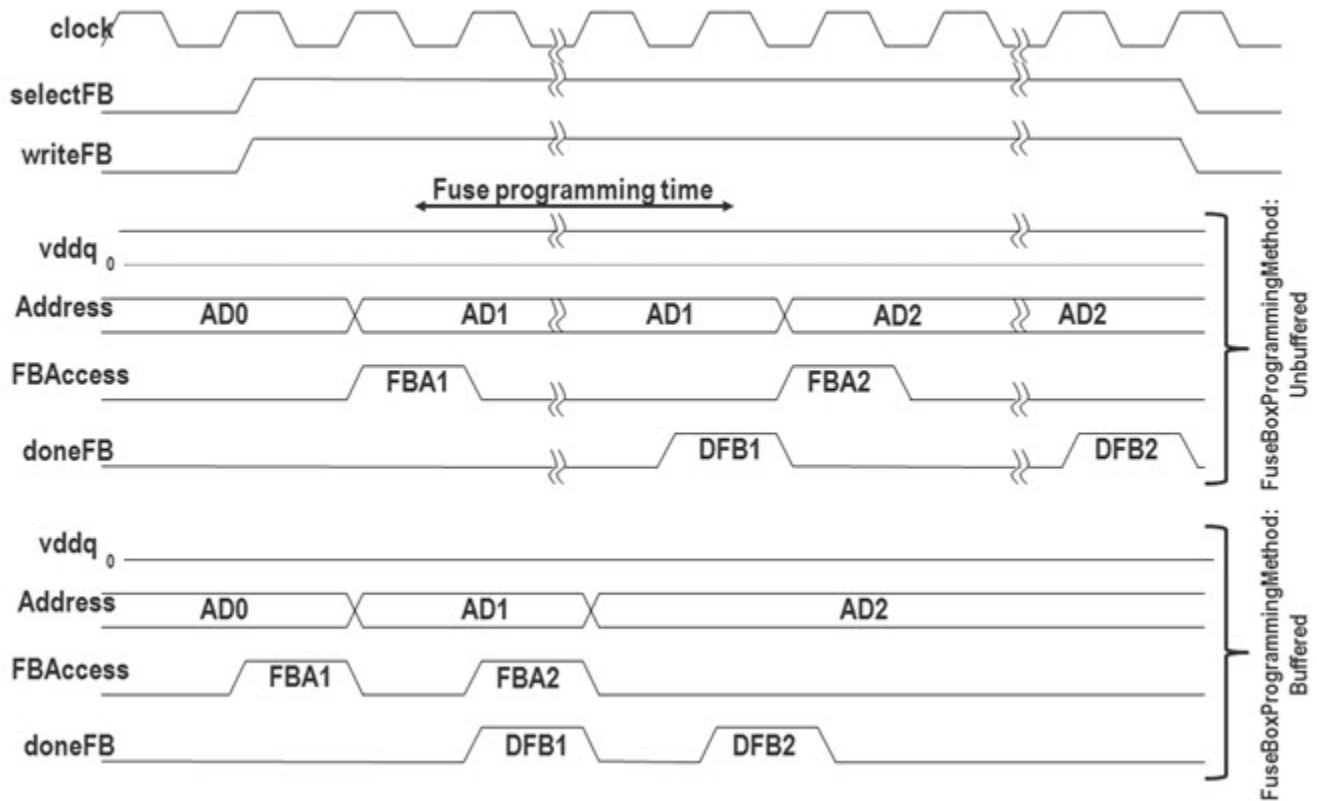
Figure 5-25 shows the protocol to initiate the Transfer and Program operation when using the buffered programming method. The programFB signal is high for both the Transfer and Program operations, contrasted to being low for the Read and Write operations. The writeFB signal is low for the Transfer operation and high for the Program operation. The selectFB signal is also high for all operations but toggles between each operation. This makes it easier to identify the beginning of each operation. No handshake occurs between the TAP and fuse box interface for these operations that use the FBAccess and doneFB signals. The simulation testbench or test program includes a pause that corresponds to the duration of the operation.

Figure 5-25. BISR Controller Fuse Box Transfer and Program Protocol
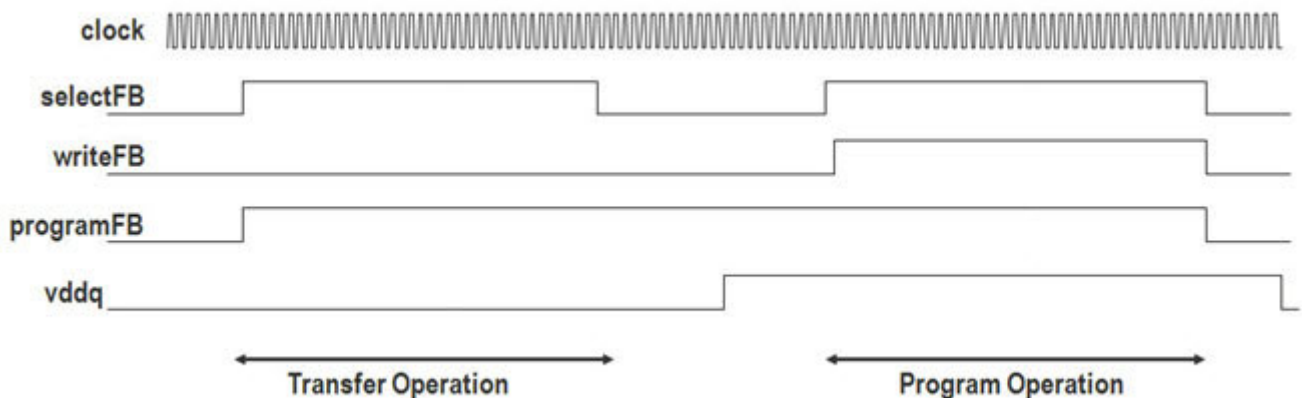
Table 5-6. Fuse Box Interface Signals

Function	Direction	Generic Fuse Box Port Name	Size	Description
FuseBoxClock	Input	clock	1	This port is the fuse box clock signal. This clock is generated by the BISR controller and is derived either from the TAP clock (TCK) or a functional clock. The clock is derived from TCK and is free-running whenever the BISR controller is enabled from the TAP. The clock is derived from the functional clock and enabled when entering the functional power-up mode. The clock is disabled 20 clock cycles after the falling edge of the selectFB input of the fuse box interface.
FuseBoxAddress	Input	Address	n	This port specifies the read or write address to the fuse box. Each address points to a single bit in the fuse box regardless of fuse box organization. This signal is updated on the rising edge of the clock.
FuseBoxWrite	Input	writeFB	1	This port is used during the fuse box write cycle. This signal remains high for the entire fuse box interface write cycle. This signal is updated on the rising edge of the clock.
FuseBoxBufferTransfer	Input	programFB	1	This optional port is present when the buffered programming method is used. The signal remains low during Read and Write operations and high during Transfer and Program operations. This signal is updated on the rising edge of the clock.
FuseBoxSelect	Input	selectFB	1	The fuse box select port is raised when the BISR controller is enabled to perform read/write operations to the fuse box. This signal is updated on the rising edge of the clock.

Table 5-6. Fuse Box Interface Signals (cont.)

Function	Direction	Generic Fuse Box Port Name	Size	Description
FuseBoxAccess	Input	FBAccess	1	This signal goes high for one clock cycle when the BISR controller initiates the fuse box interface read or write operation. This signal is updated on the rising edge of the clock.
Programming-VoltagePin	Input	vddq	1	This pin provides the high voltage required to blow the fuse. This pin usually is connected directly to a chip pin.
TestMode	Input	TM	1	The test mode port is used to isolate the fuse box during scan. All outputs from the fuse box that are in an unknown state during scan test mode should be gated by TM to prevent unknown values from being propagated to scannable registers.

Table 5-6. Fuse Box Interface Signals (cont.)

Function	Direction	Generic Fuse Box Port Name	Size	Description
WriteDurationCounter	Input	strobeCntVal	32	This bus input port is driven by the BISR controller when performing a fuse box write cycle. A write delay counter must be generated inside the generic fuse box and loaded with the value on the strobeCntVal port when the writeFB port is high. The counter decrements when the FBAccess port is high and stops when it reaches 0. When the counter reaches 0, the FuseBoxDone port goes high and indicates to the BISR controller that the write cycle is completed. The default write duration delay corresponds to the value specified in the DftSpecification AdvancedOptions/FuseBoxOptions/write_duration property and can be overwritten by the MemoryBisr/Controller/fuse_box_write_duration property in the PatternsSpecification. The testbench calculates the initial write duration counter value based on the clock speed and the fuse_box_write_duration property specified. The fuse box keeps all fuse box interface ports static until the strobe counter reaches 0. This port is optional. If your fuse box does not require a counter to control the write duration, this property can be omitted.

Table 5-6. Fuse Box Interface Signals (cont.)

Function	Direction	Generic Fuse Box Port Name	Size	Description
FuseBoxInterfaceReset	Input	FBreset	1	This optional input is driven by the BISR controller. It is used to reset flip-flops in complex fuse box interfaces. Its active value is 1 and is applied asynchronously with respect to the clock when the BISR controller is inactive or configured in bisr_chain_access mode. The reset is released synchronously when the BISR controller is configured in its autonomous or fuse_box_access run modes. This signal is present when the TCD FuseBoxInterface/Interface/reset property or the DftSpecification ExternalFuseBoxOptions/reset property is specified.
FuseValue	Output	fuseValue	1	This port is the 1-bit output value from the fuse box. The fuse value corresponds to the fuse box value stored at the address specified by the Address port. When fuse_box_location is set to external, the fuseValue signal is sampled on the falling edge of the BISR clock. When fuse_box_location is set to internal, the fuseValue signal is sampled on both edges of the BISR clock.
FuseBoxDone	Output	doneFB	1	This port indicates to the BISR controller that the fuse box access (write or read) is completed. When fuse_box_location is set to external, the FuseBoxDone signal is sampled on the falling edge of the BISR clock. The FuseBoxDone signal is sampled on the rising edge of the BISR clock when fuse_box_location is set to internal. The FuseBoxDone signal is used only during Autonomous modes of the BISR controller. The signal is not monitored when the fuse box is accessed through the TAP (fuse_box_access mode). The time allowed for the access is determined by the testbench.

Generic Fuse Box Module

A generic fuse box interface module is generated when the MemoryBisr/Controller/ fuse_box_location property is set to internal in the DftSpecification, and no fuse box interface is specified in a tcd_fusebox wrapper or DftSpecification fuse_box_interface_module property. The generic fuse box interface is a module that is instantiated by the BISR controller and provides a simple interface access to a fuse box. The generic fuse box interface contains the actual fuse box and the interface logic.

Figure 5-26 illustrates a high-level overview of the generic fuse box interface module. Depending on the actual fuse box module, some inputs might not be used to implement the interface logic. The generic fuse box interface module is created in the TSDB at the following location:

```
tsdb_outdir/instruments/<design_name>_<design_id>_mbisr.instrument/
```

The generic fuse box module is composed of the following files:

```
<design_name>_<design_id>_tessent_mbisr_generic_fusebox_interface.v_template
<design_name>_<design_id>_tessent_mbisr_generic_fusebox.v
<design_name>_<design_id>_tessent_mbisr_generic_fusebox_interface ->
     <design_name>_<design_id>_tessent_mbisr_generic_fusebox_interface.v_template
```

This template can be used as an example to write your own generic fuse box interface module. This behavioral simulation interface model is compatible with the buffered and unbuffered AdvancedOptions/FuseBoxOptions/programming_method modes. The last file listed is a symbolic link to the template file.

-----Note-----

Any customization of the template or symbolic link is overwritten and lost if the DftSpecification is reprocessed.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Figure 5-26. Sample Generic Fuse Box Interface Module
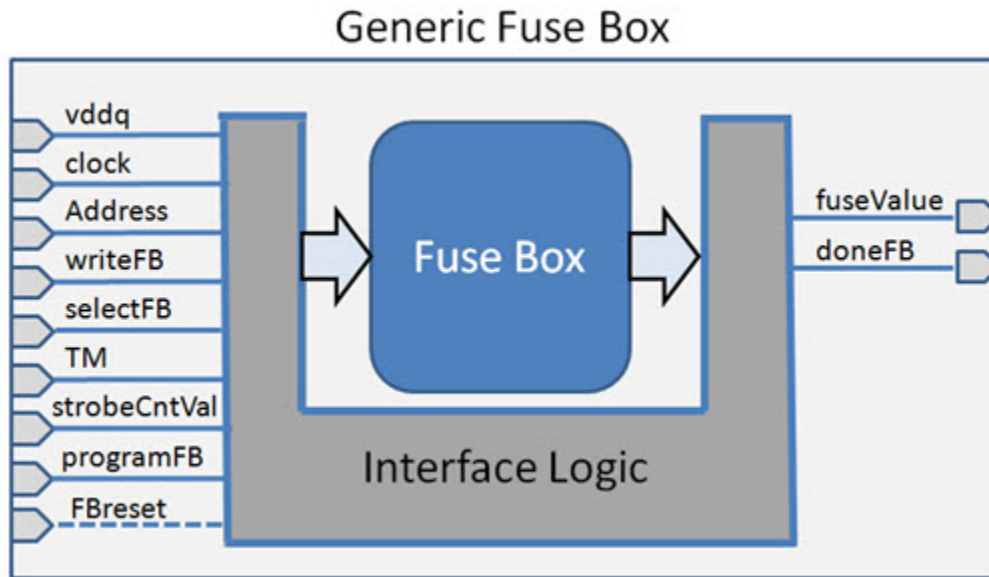
Generic Fuse Box

___Note_____

The generic fuse box template is intended to be used for evaluation or experimentation purposes only and should never be used in a real design. The generic fuse box simulation model does not perform any validation on fuse write/read timing requirements.

_____

When using the generic fuse box model, the following warning is shown during simulations:

```
**
**WARNING
**
** You are using the Generic FuseBox model as your fuse box simulation model.
** This fuse box model is instantiated inside the <design_name>_genericFuseBox
** module and should only be used for early verification of the BISR logic.
** You must edit the <design_name>_genericFuseBox module and instantiate
** the actual fuse box model that was provided by your fuse box vendor.
**
```

BISR Controller Fuse Box Operations

The BISR controller reads and writes data to the fuse box one bit at a time. The BISR controller assumes that each fuse box address holds a 0 when not programmed. The BISR controller only accesses the fuse box in write cycle to program the 1s.

If a fuse box has an output that is more than 1 bit wide, the generic fuse box interface must provide muxing logic to address each bit individually. This logic would be part of the Interface Logic shown on the right side of the fuse box in Figure 5-26.

During a write cycle, the Address port of the generic fuse box module specifies the fuse box location of a bit that a logic 1 is written. The interface logic must be designed so that the correct fuse is written during a write cycle.

Determining the Fuse Box Size

The BISR controller uses an efficient compression algorithm to minimize the number of fuses required to store the repair information. The following formulas enable you to calculate an estimate of that number.

> **Note**
> A fuse box is not needed if the soft repair method is implemented. Refer to the "Implementing Soft Repair" section for more information.

Because fuse boxes typically come in fixed sizes, the formula helps you choose the appropriate size. The Tessent Core Description (TCD) FuseBoxInterface/number_of_fuses property then can be assigned the total number of fuses in the fuse box OR the number calculated with the formula if the fuse box is used to store information other than memory repair information. By default, the number of fuses is set to 2, raised to the power of the number of address ports specified with the TCD FuseBoxInterface/Interface/address property.

Two formulas are available: a simplified formula for the case when a single BISR chain is used, and a general formula for the multi-chain case. Refer to the "Single-Chain Case" and "Multi-Chain Case" sections for more information.

Single-Chain Case

The single-chain case formula is as follows:

```
NumberOfFuses = (NumberOfRepairs * buffer_size)
              + ((NumberOfRepairs + 1) * (zero_counter_bits + 1))
```

where:

- *NumberOfRepairs* is the maximum number of repairs anticipated for the entire chip. This number typically varies between 3 and 10. The large majority of the chips requiring repair only need a single repair. Chips requiring 10 or more repairs are likely to fail non-memory tests (for example, scan tests), making the chips non-repairable and posing a reliability problem.

- *buffer_size* is the largest portion of a BISR register associated with a single spare row or column resource. If a memory has more than one spare resource, the BISR register is composed of repair data for all of the memory's spare resources. You should examine each portion of the memory's BISR register and its association with each spare resource to determine *buffer_size*. You can find the register size for each spare resource by looking at the PinMap/SpareElement wrapper in the TCD file for the memory.

For example, the RedundancyAnalysis wrapper for a memory with two spare rows and one spare column is shown below:

```
RedundancyAnalysis {
  RowSegment(rs) {
    NumberOfSpareElements : 2;
    FuseSet{
      Fuse[0] : AddressPort(A[2]);
      Fuse[1] : AddressPort(A[3]);
      Fuse[2] : AddressPort(A[4]);
    }
    PinMap {
      SpareElement {
        RepairEnable : RR0[3];
        Fuse[0] : RR0[0];
        Fuse[1] : RR0[1];
        Fuse[2] : RR0[2];
      }
      SpareElement {
        RepairEnable : RR1[3];
        Fuse[0] : RR1[0];
        Fuse[1] : RR1[1];
        Fuse[2] : RR1[2];
      }
    }
  }

  ColumnSegment(cs) {
    ShiftedIORange : Q[7:0];
    NumberOfSpareElements : 1;
    FuseSet {
      Fuse[0] : AddressPort(A[0]);
      Fuse[1] : AddressPort(A[1]);
      FuseMap[2:0]{
        ShiftedIO(Q[0]) : 3'b000;
        ShiftedIO(Q[1]) : 3'b001;
        ShiftedIO(Q[2]) : 3'b010;
        ShiftedIO(Q[3]) : 3'b011;
        ShiftedIO(Q[4]) : 3'b100;
        ShiftedIO(Q[5]) : 3'b101;
        ShiftedIO(Q[6]) : 3'b110;
        ShiftedIO(Q[7]) : 3'b111;
      }
    }
    PinMap {
      SpareElement {
        RepairEnable : CR0[5];
        Fuse[1] : CR0[4];
        Fuse[0] : CR0[3];
        FuseMap[2] : CR0[2];
        FuseMap[1] : CR0[1];
        FuseMap[0] : CR0[0];
      }
    }
  }
}
```

The memory BISR register for this example is 14 bits in length to hold the repair data described in all the SpareElement wrappers:

o   Four bits to hold the repair data for the first spare row, highlighted in green

o   Four bits to hold the repair data for the second spare row, highlighted in orange

o   Six bits to hold the repair data for the spare column, highlighted in red

Since *buffer_size* is the largest portion of a BISR register associated with a single spare resource, the *buffer_size* value for this memory is six. You must repeat the process outlined above for each memory in the current design and you use the largest *buffer_size* found in the *NumberOfFuses* equation.

- *zero_counter_bits* is the size of the counter used to count strings of consecutive 0s in the BISR chain once loaded with the BIRA results. By default, this counter size is the rounded up log_2 of the BISR chain length.

- *buffer_size* and *zero_counter_bits* are extracted automatically from the circuit and reported in the *<design_name>_<design_id>_mbisr_controller.tcd* file located in the TSDB *instrument/<design_name>_<design_id>_mbisr.instrument* folder after process_dft_specification is run. An example of this file is shown in Figure 5-27. If you need to know the value of these parameters to calculate the number of fuses, you can run process_dft_specification once and locate them in the tcd file. Note that a preliminary run of process_dft_specification might be necessary to generate the default BISR chain order as described in the "Controlling the BISR Chain Order" section.

Figure 5-27. Example MBISR TCD File Reporting BISR Statistics

```
...
Core(top_rtl_tessent_mbisr_controller) {
    MemoryBisrController {
        version                         : xxxx.x;
        zero_counter_bits               : 6;
        external_fuse_box               : OFF;
        fuse_box_address_bits           : 10;
        max_fuse_box_address            : 1023;
        fuse_box_size                   : 512;
        max_bisr_chain_length           : 65536;
        max_bisr_length_bits            : 16;
        buffer_size                     : 10;
        ...
    }
}
```

Multi-Chain Case

The multi-chain case formula has a first component that is identical to the single-chain case plus some overhead associated with the presence of additional chains; each chain is associated with a group. The formula is as follows:

```
NumberOfFuses = (NumberOfRepairs * buffer_size) + ((NumberOfRepairs + 1) *
(zero_counter_bits+1)) + (NumberOfGroups - 1) * (fuse_box_address_bits +
(zero_counter_bits + 1))
```

where:

- *NumberOfGroups* — is the number of groups in the circuit. Groups typically are associated with power domains. They also can be associated with memories that are assigned different repair priority levels within a power domain.

- *fuse_box_address_bits* — is the number of address bits required to access the fuse box. If the fuse box is used only to store memory repair information, the number of address bits is simply the $\log_2$ of the number of fuses in the fuse box, and the value is reported in the MBISR TCD file as shown in Figure 5-27 and as described in the "Single-Chain Case" section. However, if the fuse box is shared for other purposes, the number of address bits is larger.

_____ **Note** _____

For the multi-chain case, the longest BISR chain determines the *zero_counter_bits* value. The individual chain lengths, as well as the number of groups, are reported in the MBISR TCD file as shown in Figure 5-27 and as described in the "Single-Chain Case" section.
_____

Incremental Repair Case

Hard incremental repair involves programming the fuse box in more than one test insertion, for example during wafer probe, package test, final test or even system test.

Multiple insertions are enabled by specifying the max_fuse_box_programming_sessions property in the DftSpecification/MemoryBisr/Controller wrapper with a value greater than 1, which is the default value, or by selecting the "unlimited" option.

When max_fuse_box_programming_sessions is specified as "unlimited", fuse box compression is disabled and the repair solution from the BISR chains are stored uncompressed in the fuse box. Subsequent repair solutions can be written directly into the fuse box without affecting the repair solution from previous insertions. This enables an unlimited number of self_fuse_box_program autonomous mode programming sessions. The number of fuses required for this use must be equal to, or greater than the total number of bits in the BISR chain. Disabling the fuse box compression hardware is also useful in the rare case where more than 50% of the repair registers are expected to be used. In this case, the compression algorithm becomes inefficient and may result in requiring more fuses than there are bits in the BISR chain.

When the max_fuse_box_programming_sessions property is set to 2 or more, the number of fuses is calculated as follows:

```
NumberOfFuses = (max_fuse_box_programming_sessions * FusesPerSession) +
(((max_fuse_box_programming_sessions - 1) * (fuse_box_address_bits + 1))
 + 1)
```

where:

- *FusesPerSession* — is the number of fuses calculated for a single session as explained in the "Single-Chain Case" and "Multi-Chain Case" sections. The formula is slightly pessimistic because it assumes that almost all repairs need to be performed during the first programming session and that all programming sessions are used. However, the number of fuses necessary to perform a number of repairs could be significantly less depending on the statistical distribution of the repairs. Conversely, the total number of repairs performed with the calculated number of fuses could be significantly higher depending on the number of programming sessions that are used.

  For example, suppose that NumberOfRepairs is set to 10 in the calculation of FusesPerSession for a circuit with a single BISR chain and that the result of the calculation is 250. Also suppose that max_fuse_box_programming_sessions is set to 2 so that the total NumberOfFuses is 512 for the incremental repair case. This number of fuses enables performing up to 10 repairs if both programming sessions are used and if most repairs are performed during the first programming session. However, if all repairs are performed in a single programming session, up to 20 repairs can be performed with this number of fuses.

- *fuse_box_address_bits* — is the number of address bits required to access the fuse box. If the fuse box is used only to store memory repair information, the number of address bits is simply the $\log_2$ of the number of fuses in the fuse box, and the value is reported in the MBISR TCD file as shown in Figure 5-27 and as described in the "Single-Chain Case" section. However, if the fuse box is shared for other purposes, the number of address bits is larger.

Creating and Inserting the BISR Controller

This sections covers BISR controller function, implementation and operation.

Understanding the BISR Controller

The BISR controller hardware is created at the top level of the chip automatically. The hardware implements several functions and can be used to perform the following operations:

- Compress the repair information and write the result into the fuse box

- Decompress the fuse box contents and shift the contents into the chip BISR chain

- Initialize or observe BISR chain content via the TAP

- Read and program fuses via the TAP

The BISR controller is accessed using the TAP. The BISR chain control ports are automatically connected to the BISR controller. The BISR controller is also connected to the fuse box. The BISR controller provides external access to the BISR chain and the fuse box via the TAP. The BISR controller can also be used in an Autonomous run mode that provides BISR chain and fuse box access. This access handles all the repair information inside the chip.

When a chip is powered on, the content of the BISR chain is unknown. Because the content of the BISR chain is driving the memory repair ports, its content must be cleared or programmed with the repair information before the chip can be used. The BISR chain is asynchronously reset when holding the BISR controller functional repair enable input pin active. The two methods to initialize the BISR chain with repair information are as follows:

- *Apply a low-to-high transition on the BISR controller functional repair enable input pin*

  When the transition occurs, the BISR controller loads the repair information from the fuse box into the BISR chain. Typically, this method is used in a system where the BISR controller repair enable port is tied to a power-on reset signal.

  If multiple power domain groups are present, the BISR controller has input ports named PowerDomainGroupEnable_<pdg_label> corresponding to each power domain group. These input ports are used to select the BISR segments to initialize when the BISR controller is enabled. These ports provide the ability to power-up individual BISR chains from selected power domains in the system, while preserving the repair information in other power domains.

- *Initialize through the TAP using the Autonomous run mode*

  One of the autonomous operations is power_up_emulation, which emulates a functional power-up reset. Typically, this method is used during manufacturing when the chip is accessed only through the TAP.

The BISR controller provides several ways to program the fuse box. The fuse box can be programmed either externally via the TAP or internally using the autonomous self-fuse box programming mode with the repair information content of the BISR chain.

BISR Controller Run Modes

The BISR controller has three run modes:

- autonomous

  In this mode, the controller operation does not require any data patterns from the tester. The repair information and compression/decompression are done inside the chip. Seven autonomous operations can be performed: self_fuse_box_program, power_up_emulation, verify_fuse_box, rotate_bisr_chain, calculate_bisr_chain_length, load_bisr_chain, and clear_bisr_chain.

- bisr_chain_access

  In this mode, the chip-level BISR chain is accessed from the TAP TDI and TDO ports. Valid operations in this mode are enable_rotation (if no BISR controller is present), enable_bira_capture, and select_bisr_registers.

- fuse_box_access

  In this mode, the fuse box is accessed from the TAP through the BISR controller. The two valid operations in this mode are program and read. The program operation can be used to write to a single fuse address from the TAP. The read operation can be used to read a single fuse address from TAP. The fuse box is addressed one bit at a time irrespective of the actual fuse box organization.

Several vendors provide fuse boxes, and these fuse boxes have a large number of interfaces and protocols that are not common to each fuse box. Because of this complexity, the fuse box read and write protocols are encapsulated inside of an interface. The interface is custom designed for each fuse box. However, this is a one-time effort for a given technology. Also, Siemens EDA can provide this interface for certain technologies or design a new interface under certain conditions.

The fuse box can be instantiated inside the BISR controller or can be external to the BISR controller.

- Internal Fuse Box

  When the fuse box is internal to the BISR controller, a generic fuse box model is used to include the fuse box that encapsulates the specific fuse box read and write protocols. A generic fuse box model is generated as a template file. The fuse box must be instantiated, and the read and write protocols must be implemented inside this module.

- External Fuse Box

  When the fuse box is located outside the BISR controller, a fuse box interface must be instantiated in the design to provide a valid fuse box read and write interface to the BISR controller. Generally, external fuse boxes are used when the fuse box is shared for purposes other than memory repair.

Connecting the BISR Controller to an External Fuse Box

The BISR controller supports having an external fuse box. Typically, an external fuse box is used when the fuse box is shared for multiple functions and accessing it via the TAP is not suitable for writing or reading fuse box functions that are not related to memory repair. Another situation that occurs is when the fuse box has features not supported by the BISR controller.

Tip

For information and examples on integrating a TSMC eFuse with a Tessent MemoryBIST BISR controller, refer to Knowledge Base Article MG596434 *"Tessent MemoryBIST TSMC 28nm and Newer eFuse Support (Tessent Shell Flow)"*, available from the Support Center at https://support.sw.siemens.com.

When an external fuse box is used, the fuse_box_location property in the MemoryBisr/ Controller wrapper must be set to external. The fuse_box_interface_module property located in the same wrapper can be used to specify the library module for the external fuse box. If this is not specified, the library module is inferred from the design instance specified in the design_instance property of ExternalFuseBoxOptions wrapper. If neither of these properties are specified and only a single tcd_fusebox file exists in the design, the fuse box module is inferred from this tcd_fusebox description.

The design instance for the external fuse box must already be instantiated in the design. Typically, the fuse box is instantiated within a module that also contains interface logic. If the external fuse box is only used for memory repair, all input ports of the module should be tied off, and the output ports should be left open. When executing the process_dft_specification command, all input ports are first disconnected, and then connected to the BISR controller module. If the external fuse box is also used for functional purposes, the input and output ports of the interface logic should already have connections to the functional logic. In this case, the process_dft_specification command muxes the input ports and connects to the existing output port connections for fuse box connections.

The core description for the external fuse box can automatically be read in during module matching. Refer to the set_design_sources -format tcd_fusebox command description for information about specifying where it is searched for. Refer to the read_core_descriptions command description to learn how to read the core description explicitly. Finally, you can refer to the set_module_matching_options command description for information about the name matching process.

If the instantiated module has a core description with a FuseBoxInterface wrapper, then connections between the BISR controller and the fuse box interface are done automatically. If a core description is not available, or not complete, explicit connections can be made in the MemoryBisr/Controller/ExternalFuseBoxOptions/ConnectionOverrides wrapper. When completing explicit connections within the DftSpecification ConnectionOverrides wrapper, note that it is mandatory to specify the following properties:

- done

- read_data

- write_en

- select

- access_en

- address

- write_duration_count

A user cannot omit one of the ports listed above from the tcd_fusebox and simply provide the single missing port in the ConnectionOverrides wrapper. It is recommended that the core description for the fuse box interface be as complete as possible. All ports described in the core description interface wrapper must exist on the actual design module instance. The design module instance is allowed to have additional ports not described in the library module Interface wrapper, as long as they are specified in the ConnectionOverrides wrapper before executing the process_dft_specification command.

Connecting a BISR Controller to System Logic

Typically, system logic is connected to the BISR controller for initiating memory repair and monitoring the progress of the operation.

All connections are specified in the DftSpecification configuration file in the MemoryBisr/ Controller wrapper. The system logic must provide a minimum of three input connections for initiating memory repair on power up.

- The BISR controller input clk must be driven by an appropriate functional clock. The connection is made by specifying the repair_clock_connection property.

- The BISR controller input resetN is the signal used to reset the BISR chain(s) and initiate memory repair. The connection is made by specifying the repair_trigger_connection property.

- The BISR controller repair mode input specifies the BISR controller autonomous run mode the controller initiates when it detects a rising edge on resetN. The three-bit repair mode connection is made by specifying the repair_mode_connection property. Refer to the property description for additional information and the run-mode encoding for this input.

    _____ **Note** _____
    All BISR chains maintain their values if resetN remains high after the controller execution. If resetN is set low and the selected run mode is Functional Power-Up or BISR Chain Length Calculation, the BISR chain is cleared. If there are multiple power domain groups, only the enabled BISR chains are cleared.

If the design contains more than one power domain group, the following connections are also required inside the MemoryBisr:Controller wrapper:

```
DftSpecification {
  MemoryBisr {
    Controller {
      PowerDomainOptions {
        PowerDomainName(pdgA) {
          enable_from_pmu_connection : pin_or_net_name ;
        }
        PowerDomainName(...) {}
      }
    }
  }
}
```

BISR controller outputs can be connected to system logic for monitoring the progress of the power-up operation. Those connections are optional but recommended to make sure that the memory repair information of the fuse box has been successfully transferred to all memories before accessing them for the first time.

- The BISR controller output BisrGo indicates that the BISR operation was successful when its value is 1. The state of BisrGo is only valid when BisrDone is also 1. The connection is made by specifying the AdvancedOptions:bisr_pass_connection property.

- The BISR controller output BisrDone indicates that an Autonomous mode of operation (for example, power-up) is completed when its value is 1. The connection is made by specifying the AdvancedOptions:bisr_done_connection property.

When the design contains more than one power domain, additional connections are required. At a minimum, the PowerDomainOptions:PowerDomainName:enable_from_pmu_connection property must be specified for each input of the BISR controller associated to a power domain. This input determines whether the power domain is loaded with repair information on the next low-to-high transition of the functional input with the repair_trigger_connection property.

Additional properties are available to monitor BISR controller output monitors dedicated to each group. The PowerDomainOptions:PowerDomainName:busy_to_pmu_connection property and the PowerDomainOptions:PowerDomainName:done_to_pmu_connection property. The "busy" output is high when the repair information for the corresponding power domain group is being loaded in the BISR chains. The "done" output is high once the repair information for the corresponding power domain has been loaded successfully in the BISR chains. You must monitor the global BisrGo output to confirm that the operation was successful.

All properties associated to controller inputs are not repeatable whereas all properties associated to controller outputs are repeatable so that you can connect a same output to multiple destinations.

Choosing a Functional Repair Clock

The distributed architecture and conservative clocking methodology used for self-repair require some care in the selection of the functional repair clock used to apply repair during chip power up.We recommend that you use a functional clock of 50 MHz or less in that functional mode.

Note that all other modes used for manufacturing use the TAP clock (TCK), which is 10 MHz by default. Using such a low frequency simplifies timing closure. At 10 MHz, self-repair takes approximately 1 ms to complete for a BISR chain with 5000 bits, which is sufficient to repair hundreds of memories. If this time needs to be shortened, you can use a faster functional clock. However, you must consider the following factors when making a decision because they might limit the maximum achievable frequency:

- Clock balancing of the BISR chain. The BISR chain is distributed throughout the chip, and some segments of the BISR chain might be part of pre-designed circuit blocks. Balancing the BISR clock to achieve a higher speed of operation might be difficult.

- Clock edge uncertainty. Retiming registers are inserted between BISR registers so that balancing the BISR clock is not necessary. However, these retiming registers might limit the maximum achievable frequency because of the uncertain time of occurrence of the falling BISR clock edge with respect to its rising edge.

- Distribution of the BISR registers. The BISR chain is distributed throughout the chip, and some segments of the BISR chain might be part of pre-designed circuit blocks. Long wires between BISR registers might limit the maximum achievable frequency. Two mechanisms are available to alleviate timing issues:

  o **Pipeline Registers**

    - You can insert a pipeline register before and after each BISR register for memory instances specified with the BisrElement wrapper. You can independently configure the clocking for each pipeline register as negative or positive edge.

    - You can insert multiple pipeline registers where you want within each power domain BISR chain using the ": pipeline" declaration in the BisrSegmentOrderSpecification wrapper.

  o **Placement-Based BISR Chain Routing**

    - MemoryBIST optimizes the routing of the BISR chain based on placement when you use the BisrSegmentOrderSpecification wrapper and provide a DEF file that specifies memory coordinates. You load the DEF file using the read_def command prior to running the check_design_rules command. If you do not provide a DEF file, you can specify the BISR chain order to minimize the length of BISR connections and group by power domain.

    - You can independently override the placement of BISR registers and associated pipeline registers and specify their insertion into a particular parent design

instance using the BisrElement and BisrElement/Pipeline wrappers. The BISR chain ordering is then determined based on the parent design BISR_SI pin placement information found in the DEF file.

Top-Level Verification and Pattern Generation

This section covers fuse box programming, verifying the top-level BISR and generation of manufacturing test patterns.

Fuse Box Programming

The two methods for programming fuses using the BISR controller are the following:

- Using the Autonomous mode as described in the "Autonomous Self Fuse Box Program" section. This is the default method used to store memory repair information contained in BISR registers.

- Using the TAP as described in the "FuseBox Access" section. This method is used mostly for diagnosis and for storing information other than memory repair.

Note

The "Verifying BISR at the Block Level" section describes these two methods in detail.

Autonomous Self Fuse Box Program

In the Autonomous self_fuse_box_program run mode, the BISR controller has the capability to rotate the BISR chain, compress its content, and write the content to the fuse box. The BISR chain must hold the correct memory repair information before the autonomous fuse box programming starts. There are two methods to load the memory repair information into the BISR chain:

- Run memory BIST followed by a BIRA-to-BISR transfer

- Scan in the BISR information through the TAP

FuseBox Access

The BISR controller provides read and write access to the fuse box via the TAP when the BISR controller is running in the FuseBoxAccess mode.

In this run mode, a setup chain is accessed through the TDI and TDO ports of the chip. This setup chain contains a register that holds the fuse box address to write. A single bit of the fuse box is written at a time. For each write access to the fuse box, the fuse address value is scanned through the TAP into the setup chain of the BISR controller. The TAP must then pause to provide sufficient delay as specified in the fuse box data sheet. Multiple fuse box addresses might be written by successively scanning the address and pausing until the write duration delay is completed. The write duration delay is controlled by the PatternsSpecification property fuse_box_write_duration, and the done signal from the fuse box Interface is not monitored in this mode.

Same as for the write access, the read access can only read one bit at a time from the fuse box. For each bit to be read from the fuse box, the bit address is scanned into the BISR controller through the TAP. The bit value is then captured in the BISR controller setup chain and is scanned out during the next shift instruction. The done signal is not monitored when the fuse box is accessed through the TAP. The time allowed for the access is determined by the testbench. Two clock cycles are allowed if the DftSpecification fuse_box_location property is set to "internal" and three clock cycles if it is set to "external". The PatternsSpecification property test_time_multiplier can be specified to allow more time for an access.

To program the fuse box using the FuseBoxAccess mode, the BISR chain content must be scanned out, compressed externally using the *CompressBisrChain* script (which is included in the Tessent products release), and written one fuse at a time. The BISR chain must initially hold the correct memory repair information. The write_memory_repair_dictionary command is used to create the configuration file that is required for running the *CompressBisrChain* script. This file also serves as a useful verification reference as it contains all the fuse box parameters, as well as an ordered list of BISR register ICL instances listed per power domain group.

Verifying BISR at the Block Level

The verification of repairable memories involves extra tasks that are not required on memories without self-repair. This can be achieved by performing the verification tasks outlined below at the design block level in a bottom-up flow.

Note that the tasks listed below assume there is no BISR controller present in the block. In the case where a BISR controller is present, refer to the "Verifying Top-Level BISR" section.

- Executing Fault-Inserted Memory BIST

- Performing BIRA-to-BISR Capture

  o Scan External BISR Chain into the Internal BISR Chain

    Memories with a serial BISR interface require this extra step. A BISR chain rotation shift is required to copy the content of the external BISR chain into the internal BISR chain registers.

    For memories with a parallel BISR interface, this step is not required because the output of the external BISR registers is driving the memory repair ports directly. However, if a design has one or more memories with serial BISR interfaces, all BISR registers must perform the shift rotation, including the BISR registers for the memories with parallel BISR interface.

- Executing Post-Repair Memory BIST

These verification tasks are specified inside the PatternsSpecification configuration file for a memory BIST controller with BISR hardware as shown in Figure 5-28. For complete reference information for the PatternsSpecification configuration file syntax, refer to the "Configuration-Based Specification" chapter of the *Tessent Shell Reference Manual*. Additionally, the write_memory_repair_dictionary command creates a configuration file that can serve as a useful verification reference as it contains an ordered list of BISR register ICL instances listed per power domain group.

Figure 5-28. Example PatternsSpecification to Verify BIRA and BISR

```
PatternsSpecification(core,rtl,signoff) {

Patterns(MemoryBist_P1) {
    tester_period : 100ns;
    ClockPeriods {
      clka : 9.00ns;
    }
    TestStep(ClearBisrChain) {
      MemoryBisr {
        run_mode : bisr_chain_access;
        Controller(core) {
          BisrChainAccessOptions {
            default_write_value : all_zero;
          }
        }
      }
    }
    TestStep(PreRepair) {
      MemoryBist {
        run_mode : run_time_prog;
        Controller(core_rtl_tessent_mbist_c1_controller_inst) {
          DiagnosisOptions {
            compare_go : on;
          }
          RepairOptions {
            check_repair_status : non_repairable;
          }
        }
      }
    }
    TestStep(CaptureBiraRotate) {
      MemoryBisr {
        run_mode : bisr_chain_access;
        Controller(core) {
          BisrChainAccessOptions {
            enable_rotation : On;
            enable_bira_capture : On;
          }
        }
      }
    }
    TestStep(PostRepair) {
      MemoryBist {
        run_mode : run_time_prog;
        reduced_address_count : on;
        Controller(core_rtl_tessent_mbist_c1_controller_inst) {
          DiagnosisOptions {
            compare_go : on;
            compare_go_id : on;
          }
        }
      }
    }
  }
}
```

Executing Fault-Inserted Memory BIST

Executing Fault-Inserted Memory BIST

In this task, faults are inserted into the memory using the procedure documented by the memory provider, which usually requires adding command-line options when starting the simulation.

MemoryBIST is invoked by TestStep (PreRepair) as shown in Figure 5-28. As the MemoryBIST algorithm verifies the memory, the BIRA module is collecting failure information and computing the repair information for the memory. Assuming that the memory is repairable, the repair solution is available from the BIRA module when the MemoryBIST controller completes the execution.

Performing BIRA-to-BISR Capture

In this task, the BIRA values are captured into the external BISR register.

This step is implemented in TestStep (CaptureBiraRotate) shown in Figure 5-28. The transfer from the BIRA to BISR registers is performed in the same way for memories with a serial or parallel repair interface. After the capture, a full rotation of the BISR chain is performed. The rotation enables the transfer of the repair information from the external to the internal BISR register of memories using a serial BISR interface. The rotation also verifies that memories using a parallel BISR interface can be mixed with memories using a serial BISR interface.

You specify BISR chain emulation for lower-level blocks with the SimulationOptions/ emulate_bisr_chains_in_lower_physical_blocks property and the design view for these blocks with the LowerPhysicalBlockInstances property in the PatternsSpecification. Refer to the emulate_bisr_chains_in_lower_physical_blocks property description of the auto (default) switch for details on the interaction of these properties and the resulting simulation view. Note that simulations with the ijtag_graybox view may fail if a lower-level block has memories with a serial repair interface and the BisrChainAccessOptions/select_bisr_registers property is set to "internal" or "internal_only", or if the DftSpecifcation AdvancedOptions/ power_up_chain_select property is set to "internal". This is because the ijtag_graybox design view does not include the memory internal repair chains and can introduce unknown values into the BISR chain. To resolve the simulation issues, you can specify the full design view for the relevant lower-level blocks, which may increase simulation time.

Executing Post-Repair Memory BIST

This task re-runs the MemoryBIST algorithm used in the first test step. During this task, the repair information is used by the repairable memories, and the redundant elements replace the defective memory elements. If the memory has been repaired correctly, the GO signal from the MemoryBIST controller should be high at the end of the simulation. Assuming that all fault

inserted memories are repairable, compare failures during this test step often indicate a problem with the memory library file.

Verifying Top-Level BISR

The sign-off process for a chip with BISR hardware verifies the FuseBoxAccess, BisrChainAccess and Autonomous modes of operation of the BISR controller via the TAP.

Figure 5-29 shows a portion of the default PatternsSpecification configuration file generated for the TAP Access mode, while Figure 5-30 shows a portion of the default PatternsSpecification configuration file used for the Autonomous modes.

It is not necessary to run MemoryBIST with fault-inserted memories at the top level of the chip. This type of verification is better performed at the block level where MemoryBIST can be run on the full address space of all memories. At the top level, MemoryBIST only needs to be run with reduced_address_count: on with the BISR registers being reset to 0. This is the only type of interaction that, by default, is verified among memory BIST, BIRA, and BISR.

The default and recommended top-level BISR verification process therefore assumes that the connections between the BIRA and BISR registers, as well as the BISR registers and memory repair ports, have been verified at the assembly level already. Therefore, the testbenches generated at the top level only verify the connections between the BISR registers, the BISRcontroller, the TAP, and the fuse box. You can always modify the PatternsSpecificationconfiguration file to verify additional connections.

The write_memory_repair_dictionary command creates a configuration file that can serve as a useful verification reference. This file contains all the fuse box parameters, as well as an ordered list of BISR register ICL instances listed per power domain group.

FuseBox Access Mode

The FuseBox Access mode consists of writing and reading the fuse box through the TAP. This mode exercises the fuse box interface that the Autonomous modes also use. The FuseBox Access mode can also be used during verification of the functional fuse box interface when the fuse box is shared with other applications (that is, fuse_box_location: External in the DftSpecification).

In Figure 5-29, the first TestStep (FuseBoxProgram) programs two fuses at address 0 and 15. Those addresses are arbitrary, and extra addresses can be added easily to exercise more address lines of the fuse box, if necessary. The next TestStep (FuseBoxRead) reads the fuse box contents and expects a 0 from all locations except at 0 and 15, where it expects a 1. The read address range can be restricted as part of the read_address command. When a range is not specified, the default range is 0 to number_of_fuses, respectively.

Figure 5-29. Example PatternsSpecification Configuration for FuseBox Access Mode

```
Patterns(MemoryBisr_TapAccessMode) {
    TestStep(FuseBoxProgram) {
      MemoryBisr {
        run_mode : fuse_box_access;
        Controller(top_rtl_tessent_mbisr_controller_inst) {
          FuseBoxAccessOptions {
            operation : program;
            write_address : 0;
            write_address : 15;
          }
        }
      }
    }
    TestStep(FuseBoxRead) {
      MemoryBisr {
        run_mode : fuse_box_access;
        Controller(top_rtl_tessent_mbisr_controller_inst) {
          FuseBoxAccessOptions {
            operation : read;
            read_address(0) : 1;
            read_address(15) : 1;
          }
        }
      }
    }
  }
```

Autonomous Modes

The functions accomplished by the Autonomous modes include the following:

- Calculating the BISR chain length

- Compressing the bit pattern contained by the BISR chain

- Programming fuses

- Reading fuses

- Decompressing the repair information

- Comparing it to the original contents of the BISR chain

The BISRChainAccess mode also is exercised during verification of the Autonomous modes to load the bit pattern before TestStep(SelfFuseBoxProgram) and unload the same bit pattern after TestStep(VerifyFuseBox):

- The first TestStep (BisrLoaderReset) in Figure 5-30 initializes the BISR chain and calculates its length. The BISR chain length is used in subsequent test steps.

- The second TestStep performs a BisrChainAccess where the BISR chain is loaded with an arbitrary bit pattern. Only bits 0 and 4 of the BISR register are set in this example; these bits act as an arbitrary repair solution inside this BISR register. Note that the bit index on the BISR chain is 0 for the bit closest to TDO. While the BISR chain is loaded with the bit pattern, the values of the BISR chain are shifted out and compared on TDO. When multiple BISR scan chains are present, all chains are concatenated to shift in the bit pattern, by default.

- The third TestStep (SelfFuseBoxProgram) rotates the arbitrary bit pattern of the BISR chain, compresses it and programs the appropriate fuses. Because the time required to program such a simple bit pattern is significantly less than the maximum time allowed during manufacturing, the max_repair_count property is used to shorten the simulation time. This property is for verification purposes only and is not included in the configuration file used for manufacturing purposes. The SelfFuseBoxProgram mode does not disturb the content on the BISR chains.

  If the fuse box interface has an internal buffer (for example, the interface has a port with write_buffer_transfer property), you can postpone the final fuse box programming to a subsequent test step by setting the inhibit_buffer_to_fuse_transfer property to On.

- The fourth TestStep (VerifyFuseBox) verifies that the contents of the fuse box, once decompressed, corresponds to the initial bit pattern that was loaded in the BISR chain. Knowing that the content of the BISR chain was not disturbed after the SelfFuseBoxProgram step, the decompressed values from the fuse box should match the content of the BISR chain. As for all Autonomous modes, the GO and DONE output signals are set to 1 if the operation is successful.

Figure 5-30. PatternsSpecification Configuration for Autonomous Modes

```
Patterns(MemoryBisr_AutonomousMode) {
    TestStep(BisrLoaderReset) {
      MemoryBisr {
        run_mode : autonomous;
        Controller(top_rtl_tessent_mbisr_controller_inst) {
          AutonomousOptions {
            operation : power_up_emulation;
          }
        }
      }
    }
    TestStep(BisrChainAccess) {
      MemoryBisr {
        run_mode : bisr_chain_access;
        Controller(top_rtl_tessent_mbisr_controller_inst) {
          BisrChainAccessOptions {
            enable_bira_capture : off;
            select_bisr_registers : external;

BisrRegisterAccessOptions(core_inst2.blockB_clka_i1.memA_bisr_inst)
{
            write_value(0) : 1;
            write_value(4) : 1;
          }
        }
      }
    }
    TestStep(SelfFuseBoxProgram) {
      MemoryBisr {
        run_mode : autonomous;
        Controller(top_rtl_tessent_mbisr_controller_inst) {
          AutonomousOptions {
            operation : self_fuse_box_program;
            max_repair_count : 1;
          }
        }
      }
    }
    TestStep(VerifyFuseBox) {
      MemoryBisr {
        run_mode : autonomous;
        Controller(top_rtl_tessent_mbisr_controller_inst) {
          AutonomousOptions {
            operation : verify_fuse_box;
          }
        }
      }
    }
}
```

Autonomous Mode for Memory BIST-Only Verification

In the example shown below, an autonomous mode of the controller called clear_bisr_chain initializes all flip-flops in the BISR chain without calculating its length. This enables quick

initialization of the memory repair inputs when performing memory BIST verification. The TestStep (bisr_clear) is added automatically in the PatternsSpecification file for memory BIST controllers that are testing repairable memories.

Figure 5-31. PatternsSpecification Configuration for Autonomous Modes for MemoryBIST-Only Verification

```
Patterns(MemoryBist_P1) {
ClockPeriods {
      clkb : 12.0ns;
      clka : 3.0ns;
}
TestStep(clear_bisr) {
      MemoryBisr {
        run_mode : autonomous;
        Controller(top_rtl_tessent_mbisr_controller_inst) {
          AutonomousOptions {
            operation : clear_bisr_chain;
          }
        }
      }
    }
    TestStep(run_time_prog) {
      MemoryBist {
        run_mode : run_time_prog;
        reduced_address_count : on;

Controller(core_inst1.blockA_clka_i1.blockA_l1_i1_blockA_l2_i1_blockA_rtl
_tessent_mbist_c1_controller_inst) {
        }
      }
    }
}
```

Functional Mode Verification

The BISR controller has a functional mode of operation enabling memories to be repaired in the system. You must verify that the proper functional inputs are applied to the BISR controller and that your system correctly interprets the various BISR controller outputs in that functional mode of operation. You perform this verification with manually created functional patterns.

This section describes the protocol between the system logic and the BISR controller, as well as the equations to determine approximate repair time. This section also shows how you can use a ProcedureStep to partially verify the functional mode of operation in combination with other BISR manufacturing tests.

BISR Protocol (Single Power Domain Group)

The figure below and the following section outlines the BISR protocol for a single power domain group.

The controller reads, decompresses, and then shifts the fuse box content into the BISR chain using a system clock. Assuming that the TAP has been reset, a low-to-high transition on the resetN functional input initiates the repair. The Done output of the controller indicates that the repair operation is complete, and the Go output indicates if the operation was successful. A third output, bisrCEDis, indicates when the actual memory repair takes place. This signal also is used to disable memories with a serial repair interface. Figure 5-32 shows the protocol for a single power domain.

Figure 5-32. BISR Controller Protocol in Functional Mode (Single Power Domain Group)
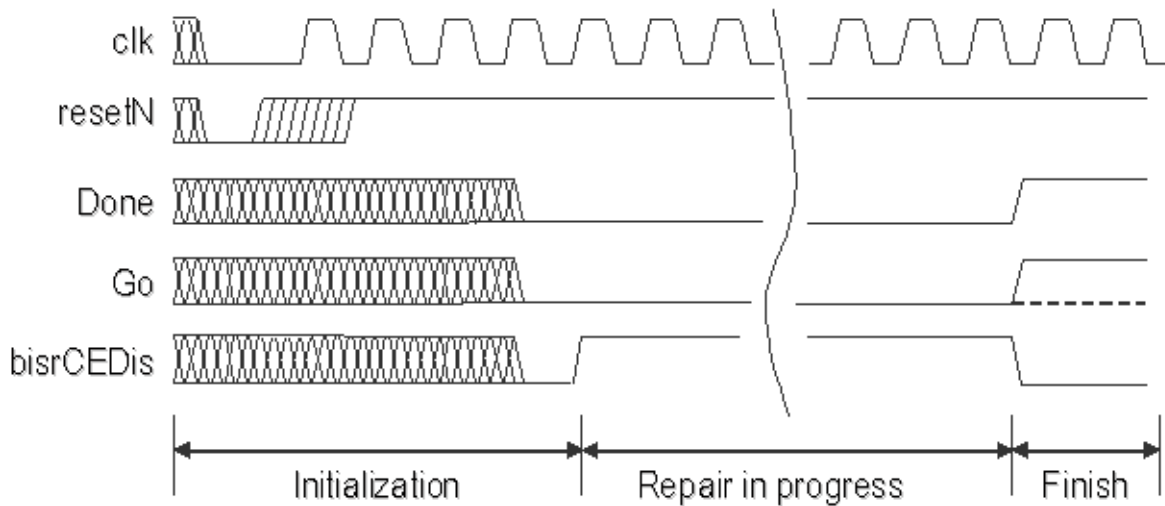
The DftSpecification/MemBisr/Controller wrapper properties associated with the controller ports are as follows:

- repair_clock_connection : clk;

- repair_trigger_connection : resetN;

- AdvancedOptions/bisr_done_connection : Done;

- AdvancedOptions/bisr_pass_connection : Go;

- PowerDomainOptions/PowerDomainName(*pdg_name*)/busy_to_pmu_connection : bisrCEDis;

The resetN input signal is synchronized using the system clock input clk. The resetN input must be low for a sufficient amount of time to asynchronously reset the four flip-flops of the synchronizer circuit. The amount of time depends on the cell library that you used to synthesize your circuit. The BISR controller outputs are initialized synchronously using a gated version of the functional clock input. The initialization is complete 3 cycles after a definite high level was detected on resetN.

The Done and Go outputs remain low until the repair completes. The Done output then goes high, and the Go output also goes high if the repair was successful, but stays low if not. The bisrCEDis output goes high one cycle after the initialization and goes low once repair completes.

___ **Note** ___

📄 Memories with a serial repair interface might not be available immediately after bisrCEDis goes low. You should wait a few clock cycles to make sure that the signal had time to propagate to all memories that are not on the same clock domain as the BISR controller. Timing scripts (SDC and STA) assume a false path in that case. Functional simulations must be used to verify this asynchronous interface.

BISR Protocol (Multiple Power Domain Groups)

The BISR protocol is slightly different when multiple power domain groups are present.

Figure 5-33 illustrates the BISR protocol for a circuit with three power domain groups. Signals clk (repair clock), resetN (repair enable), Done, and Go are the same as in Figure 5-32. New signals in Figure 5-33 include a 3-bit input bus (PwrDomGrpEn, corresponding to the DftSpecification/MemoryBisr PowerDomainName(*pdg_name*)/enable_from_pmu_connection property) and a set of outputs for each of the three groups. Signal names associated with a group have the same suffix, and is the power domain group name that you assigned during the DftSpecification step. The three groups are as follows:

- G0 (_G0 suffix)

- G1 (_G1 suffix)

- G2 (_G2 suffix)

For each group, Figure 5-33 shows four signals:

- toBisrSi_G*x* is the serial input of the BISR chain.

- bisrClk_G*x* is the clock of the BISR chain.

- PDGDone_G*x* (AdvancedOptions/bisr_done_connection property) is a signal indicating that the memories of group *x* have been repaired and are ready to use.

- bisrCEDis_G*x* (PowerDomainOptions/PowerDomainName(*pdg_name*)/ busy_to_pmu_connection property) indicates that repair is in progress for this group.

Other signals such as a chain serial output (fromBisr_Gx) and a reset (bisrRstn_Gx) are specific to each group but are not shown.

Figure 5-33. BISR Controller Protocol in Functional Mode (Multiple Power Domain Groups)
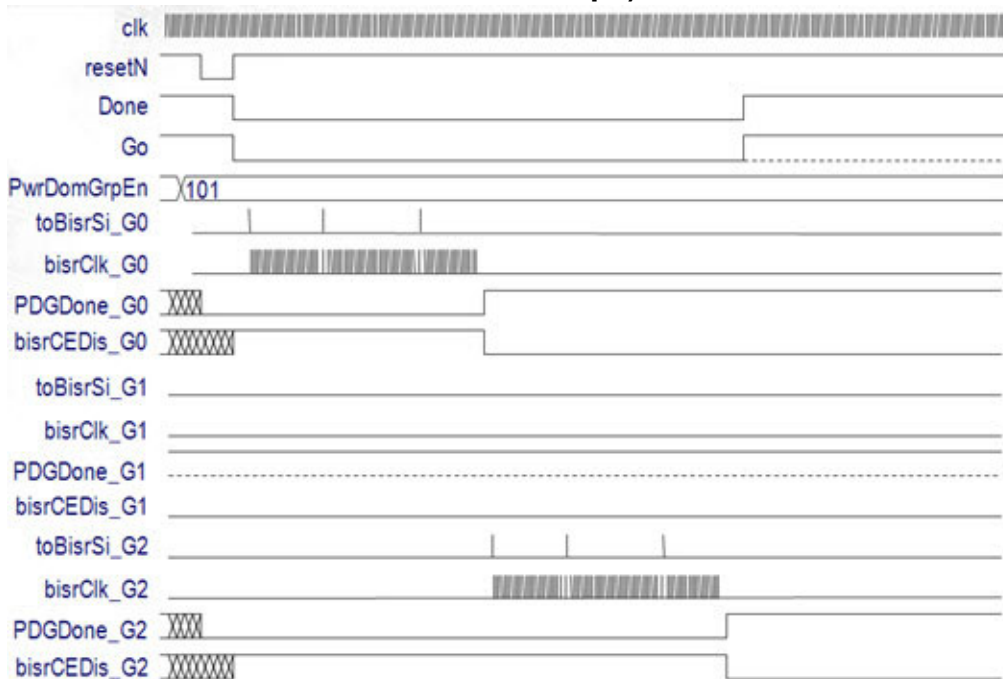
The PwrDomGrpEn input bus determines which memory groups are repaired. Each bit of the bus is associated with a group, and any combination of groups is allowed. The least significant bit of the bus is associated with group G0. In the illustrated example, bits 0 and 2 are set to 1, which indicates that groups G0 and G2 are to be repaired, whereas the state of group G1 is preserved. Group G1 was repaired previously because the PDGDone_G1 signal has a value of 1.

The PwrDomGrpEn input must be stable before the falling edge of resetN. The two inputs are combined to control the asynchronous reset of a flip-flop. Therefore, it is recommended to have the two signals being generated from the flip-flops using different clock edges of the same clock. The PDGDone_Gx signal of the selected groups immediately goes low on the falling edge of resetN.

___ **Note** ___

The relationship between PwrDomGrpEn and resetN is not constrained by the timing scripts (SDC and STA), that assume a false path. Functional simulations must be used to verify this interface.

As is the case for the single power domain group, the BISR controller starts a few cycles after the rising edge of resetN. The groups are repaired in the order determined by the DftSpecification power_domain_priority_order property. In our example, group G0 is repaired first followed by group G2. The activity on toBisrSi_Gx and bisrClk_Gx for each group shows when the repair occurs for each group. The PDGDone_G0 signal is set to 1 as soon as repair for group G1 is complete.

The repair was successful if the Done and Go global outputs are Done=Go=1.

Repair Time Calculation

Memories should not be accessed when repair is in progress. The duration of the repair operation is variable and is bounded as follows:

$T_{repair\_min} = T_{clk} *$ bisrChainLength

$T_{repair\_max} = T_{repair\_min} +$ (readDuration $* T_{clk} *$ numberFuses)

where:

- T_{clk} — period of the repair clock, clk

- bisrChainLength — length of BISR chain

- readDuration — number of clock cycles required for the fuse box interface to return a fuse value to the BISR controller

- numberFuses — number of fuses allocated for memory repair in the fuse box

Rarely are all fuses used, and the BISR chain length typically is much larger than the number of fuses allocated for memory repair. Therefore, the repair time always is much closer to the lower bound, T_{repair_min}. However, the repair time does not actually reach the lower bound because even if no memory repair is necessary, a minimum amount of overhead exists because of the circuit initialization and the reading of some fuses.

The readDuration factor is nominally 2 if fuse_box_location is set to internal, and 3 if fuse_box_location is set to external and align_access_en_with_address is set to on or auto. These values always enable a close estimation of the calculated repair time, but depending on the fuse box timing characteristics and the fuse box interface design, the calculated repair time may be slightly below the upper bound. Specifying more than 2 (or 3) clock cycles might be necessary to read fuses from the fuse box, especially at higher frequencies. However, a row of fuses is read during a single access and the result is stored at the fuse box output. Because the BISR controller requests only one fuse value at a time, the fuse box interface can return the values quickly if they are already available at the fuse box output and only initiate a slower fuse box access when necessary.

For example, if a fuse box that contains 1024 fuses reads 32 fuses at a time, a fuse box access is only performed for about 3% of the BISR controller requests:

```
((32 access requests)/ 1024 total fuses) * 100 = 3.13%
```

If a fuse box access takes 10 clock cycles, this means that the effective readDuration factor is 2.25 instead of 2, as derived below:

```
32 fuse box accesses total, each yields 32 fuse values

clock cycle breakdown for each fuse box access:
    Fuse 1:     10 clock cycles
    Fuse 2-31:  2 clock cycles each, for readDuration = 2

    Total:      72 clock cycles per 32 fuse values

Effective readDuration = (32*72 cycles)/1024 fuses = 2.25 cycles/fuse
```

However, the error on the upper bound of the repair time is only a few percentage points because the repair time is dominated by the time required to load the BISR chain.

For circuits with more than one power domain group, the bisrChainLength term of the equations shown previously is the sum of all BISR chain lengths of the groups that are scheduled for repair as indicated by the PwrDomGrpEn input bus.

Time Out Condition

The integrity of the BISR chain is verified and its length is calculated each time a power domain is powered up. This occurs by asynchronously resetting the BISR chain and inserting a leading 1 in front of the repair data loaded in the BISR chain. The BISR controller counts the number of clock cycles until the leading 1 appears at the output of the chain or until a maximum count is reached. In both cases, the Done output goes high. However, the Go output is high in the first case and low in the second case, indicating that the BISR chain is defective.

The maximum count is set, by default, to 4 times the longest calculated BISR chain length. This large maximum count is to accommodate arbitrarily large changes (ECOs) in the BISR chain configuration that could be made after generating the BISR controller. The BISR controller automatically adapts to the new configuration without having to regenerate the BISR controller. The maximum count can be modified using the DftSpecification max_bisr_chain_length property. The same maximum count is used for all power domain groups.

Combining Functional Mode with Manufacturing Tests

It is possible to combine the functional mode of operation with other modes used for manufacturing. This method enables using the automated verification infrastructure to verify that the functional inputs to the BISR controller are controlled correctly from the system logic. *However, this method does not verify how the system logic behaves in response to the various BISR controller outputs.*

This method consists of using a ProcedureStep wrapper in a patterns set to describe a user-defined input sequence to the BISR controller. Typically, a user-defined sequence calls an SVF file or an iProc to modify pin settings, combined with iRunLoop or ProcedureStep tester_cycles, tck_cycles or wait_time properties. A user-defined sequence exercising the functional power-up

mode of operation of the BISR controller can replace the test step exercising the power-up emulation mode in the post-repair PatternsSpecification. The power-up emulation mode test step is generated by default and is identified by the properties shown below, and is used within the post-repair test pattern example shown in Figure 5-43:

```
MemoryBisr {
  run_mode : autonomous;
  Controller(chip_rtl_tessent_mbisr_controller_inst) {
    AutonomousOptions {
      operation : power_up_emulation;
    }
  }
}
```

An example iProc named custom_init_bisr_chains is shown in Figure 5-34 that can be used as a template when the functional reset pin "bisr_rstn" is a primary input of the device, and the primary input is listed as a DataInPort in the ICL. The code highlighted in green represents a user-defined functional power-up sequence that is created. The sequence shown sets the resetN port on the BISR controllers to 0 and transitions it to 1, that starts the BISR controller and serially loads the BISR chain with repair data contained in the fuse box.

This iProc can then be sourced from a ProcedureStep to initialize the BISR chains, as shown in Figure 5-35. Note that the iProc requires the ICL instance name of the BISR controller as its argument. After the BISR chain is initialized from the "bisr_rstn" functional input, the post-repair memory BIST can then be performed on the repaired memory.

Figure 5-34. BISR Controlled From Functional Inputs

```
iTopProc custom_init_bisr_chains {
  {bisr_controller_instance {}}
  {initial_ireset 0}
} {

  # This argument selects one or all BISR controllers in the design.
  # The default is to trigger all BISR controllers.
  if { $bisr_controller_instance ne "" } {
    set bisr_ctl_inst_c [get_icl_instances $bisr_controller_instance]
  } else {
    set bisr_ctl_inst_c [get_icl_instances -filter \
      {tessent_instrument_type==mentor::memory_bisr && \
      tessent_instrument_subtype==controller} -silent]
  }

  # This argument selects whether to reset the ICL network at the
  # beginning of the pattern. The default is to exclude the reset.
  if { $initial_ireset } {
    iNote "Resetting the IJTAG network state."
    iReset
  }

  #
  # Insert the sequence to start the BISR controller here.
  # This sequence must set the BISR controller resetN port to 0,
  # then to 1.
  #
  # In this example, all BISR controllers are connected to a
  # common primary input port.
  #
  set func_reset bisr_rstn

  iNote "Asserting functional reset signal $func_reset to 0"
  iForcePort $func_reset 0b0;
  iApply;

  iNote "Asserting functional reset signal $func_reset to 1"
  iForcePort $func_reset 0b1;
  iApply;

  #
  # Determine the BISR chain length for each BISR controller.
  # Wait for the controller(s) to download the repair values
  # from the fuse box into the BISR chains.
  #
  foreach_in_collection ctl_inst $bisr_ctl_inst_c {
    set ctl_inst_name [get_single_name $ctl_inst]
    set ctl_mod_name [get_attribute_value_list \
      -name module_name $ctl_inst]

    iClock ${ctl_inst_name}.clk
    set controller_period_ns [get_iclock_option \
      [get_icl_pins clk -of_inst $ctl_inst] -period -in ns]

    set memory_repair_info_dict [write_memory_repair_dictionary \
```

```
            -fuse_box_controller_icl_instance $ctl_inst_name -return_dict]
    dict with memory_repair_info_dict {
      set fuse_box_init_cycles \
        [expr {ceil(1.0*$fuse_box_init_duration / $controller_period_ns)}]
      set fuse_box_read_cycles \
        [expr {ceil(1.0*$fuse_box_read_duration / $controller_period_ns)}]
      set pdg_num [dict size $pdg_length]
      set bisr_chain_length \
        [expr [join [dict values $pdg_length] +] + $pdg_num]
      set run_length [expr {1.0*($fuse_box_size * $fuse_box_read_cycles \
        + $fuse_box_init_cycles + $bisr_chain_length \
        + ($pdg_num * $zero_counter_bits) + ($pdg_num * 3)) }]
    }

   iNote "Initializing BISR chains for BISR controller $ctl_inst_name"
    iRunLoop $run_length -sck ${ctl_inst_name}.clk
  }
}
```

Figure 5-35. Post-Repair MemoryBIST With BISR Controlled From Functional Inputs

```
Patterns(MemoryBist_postrepair_P1) {
  ProcedureStep(functional_bisr_load) {
    iCall(custom_init_bisr_chains) {
      iProcArguments {
        bisr_controller_instance : top_tessent_mbisr_controller_inst;
      }
    }
  }
  TestStep(TP3_1_run_time_prog) {
    MemoryBist {
      run_mode: run_time_prog;
      reduced_address_count: off;
      Controller(top_tessent_mbist_controller_inst) {
        DiagnosisOptions {
          compare_go: on;
          compare_go_id: on;
        }
      }
    }
  }
}
```

Creating Multiple Load Scan Patterns With Repairable Memories

Multiple load scan patterns can be used to generate scan patterns through ROM and RAM memories. Before generating multiple load patterns on repairable memories, any repair information that was previously generated by the execution of MemoryBist, must be applied to the memory repair ports. It is important to load the repair information before each execution of

multiple load patterns, since the repair information may be cleared when the test_enable signal is de-asserted at the end of the scan patterns.

The required repair information needed to initialize the memory repair ports is automatically created when executing process_dft_specification with MemoryBISR present in the design. The following file is generated in the Tessent instrument container:

```
<tsdb_outdir>/<design>_<design_id>_mbisr.instrument/
              <design>_<design_id>_tessent_mbisr.pdl
```

This file contains an iTopProc that must be called as part of the test setup to initialize the BISR chains before generating multiple load patterns. Figure 5-36 and Figure 5-37 show examples of how this iProc is utilized to initialize the BISR chains, then create and write out the multiple load patterns.

Figure 5-36. iProc Usage for Physical Block Level With No BISR Controller

```
set_context patterns -scan

# Read design
set_current_design blka

import_scan_mode int_mode

set_static_dft_signal_values memory_bypass_en 0
set instrument_path tsdb_outdir/instruments/blka_rtl_mbisr.instrument
set pdl_file blka_rtl_tessent_mbisr.pdl
source ${instrument_path}/${pdl_file}
set_test_setup_icall init_bisr_chains -non_retargetable

set_current_mode multiple_load -type internal

set_system_mode analysis

add_faults<memory I/O faults>
set_pattern_type -multiple_load on
create_patterns

write_tsdb_data -replace
write_patterns multiple_load_serial.v -verilog -serial -replace
write_patterns multiple_load_parallel.v -verilog -parallel -replace
```

Figure 5-37. iProc Usage for Chip Level With BISR Controller Present

```
set_context patterns -scan_retargeting

# Read design
set_current_design chip

add_core_instances -instance core_inst1/wrapper_i1/blka_i1 \
                    -mode multiple_load
import_clocks
```

```
set instrument_path tsdb_outdir/instruments/chip_rtl_mbisr.instrument
set pdl_file chip_rtl_tessent_mbisr.pdl
source ${instrument_path}/${pdl_file}

set_test_setup_icall init_bisr_chains -front

check_design_rules

read_patterns tsdb_outdir/logic_test_cores/blka_gate.logic_test_core/\
    blka.atpg_mode_multiple_load/blka_multiple_load_stuck.patdb
set_external_capture_options -pll_cycles 5 [lindex [get_timeplate_list] 0]
write_pattern chip_multiple_load_repair_W_blka_i1_parallel.v -parallel \
    -v -replace -param_list {SIM_TOP_NAME TB}
write_pattern chip_multiple_load_repair_W_blka_i1_serial.v -serial -v \
    -replace -param_list {SIM_TOP_NAME TB}
```

When calling set_test_setup_icall at the chip level, you must specify the "-front" command line option to ensure that the memory repair procedure is run at the beginning of the pattern. Not doing so results in an error during pattern generation, because in this condition, access to the repair logic is blocked because of the scan test signals.

You must provide an ATPG model of the eFuse during pattern generation in order meet the E14 design rule requirements. If such a model is not available, you can instead provide an input constraint on the fuseValue output pin of the eFuse interface module. To do this, create a pseudo-port associated with the fuseValue output pin of the fuse box interface instance. Then, add a constant 0 input constraint on the "fuseValue" pseudo-port, as shown in the following example:

```
add_primary_inputs chip_rtl_tessent_mbisr_controller_inst/ \
        chip_rtl_tessent_mbisr_generic_fusebox_interface_instance/fuseValue \
        -pseudo_port_name fuseValue
add_input_constraints -c0 fuseValue
```

For more details, refer to the add_primary_inputs and add_input_constraints commands.

Generating Your Manufacturing Test Patterns

Several variations are possible but the only difference is that some of the test patterns can be grouped together to form a single test pattern.

Figure 5-38 illustrates an example BISR manufacturing flow. The test patterns are organized in four groups— Initialization Test Pattern, Pre-Repair Test Patterns, Repair Test Patterns, Post-Repair Test Patterns identified with a TP0, TP1, TP2, and TP3 prefix, respectively.

Figure 5-39 to Figure 5-43 show an example manufacturing configuration file section for each group. Test patterns groups are described within the MemoryBisr and MemoryBist wrappers within the TestStep wrapper.

Figure 5-38. BISR Manufacturing Flow Example
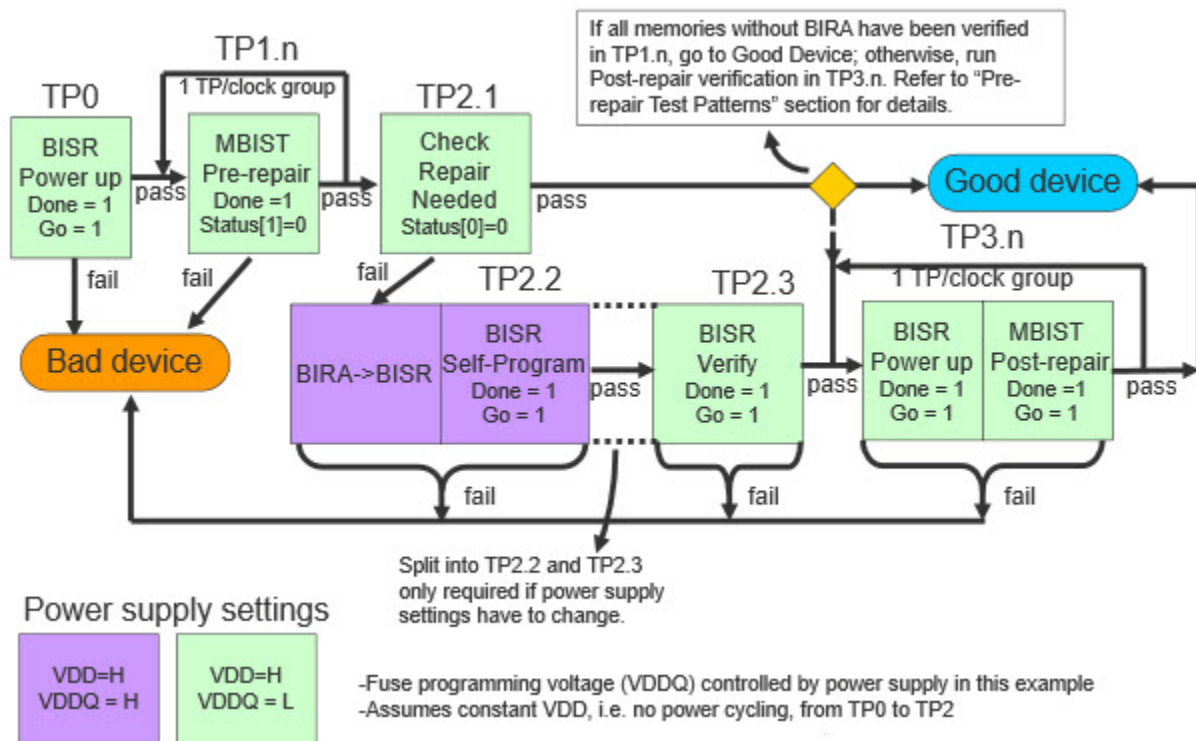
Initialization Test Pattern

The Initialization test pattern clears all BISR registers so that the spare resources of repairable memories are not used during the execution of the Pre-Repair group of test patterns. This pattern also is used to determine the length and verify the integrity of the BISR chain.

The figure below shows an example of an Initialization test pattern.

Figure 5-39. Initialization Test Pattern Configuration

```
TestStep(TP0_PowerUpEmulation) {
  AdvancedOptions {
    network_end_state : reset;
  }
  MemoryBisr {
    run_mode : autonomous;
    Controller(chip_rtl_tessent_mbisr_controller_inst) {
      AutonomousOptions {
        operation : power_up_emulation;
      }
    }
  }
}
```

Pre-Repair Test Patterns

The Pre-Repair group of test patterns tests all memories and determines if any memory cannot be repaired either because the memory does not have any spare resources or has an insufficient number to repair all failures encountered.

- For controllers testing only memories *without* spare resources, the GO output of the memory BIST controller is compared to "1" (compare_go:on). The GO_ID registers also can be inspected for collecting diagnostic information (compare_go_id:on).

- For controllers testing at least one memory *with* spare resources, check_repair_status: non_repairable is used to determine if any memory is non-repairable. There are several possible settings for compare_go and compare_go_id depending on the requirements. These are described in detail in Table 5-9. All combinations ultimately result in checking that REPAIR_STATUS[1] of all memories is 0. If not, the pattern fails. Note that memories without spare resources also have a REPAIR_STATUS register. This enables quick identification of failing memories without having to inspect or even scan GO_ID_REGs when setting compare_go:on and compare_go_id:off.

- If the design contains memory BIST controllers that were generated with software version 7.0-SP03 or Tessent v9.0, the memories without built-in self-repair (BIRA) are not tested in the Pre-Repair memory BIST patterns. An extra step is required to verify the memories without BIRA for these controllers. We recommend that you run Post-Repair memory BIST patterns if the design contains any memory BIST controllers that were generated with 7.0-SP03 or Tessent v9.0.

- If the design contains memory BIST controllers that were generated with software version 7.0-SP01 or older, the compare_go_id property is turned off by default, and memories without BIRA are not tested in the Pre-Repair memory BIST patterns. We recommend that you run Post-Repair memory BIST patterns if the design contains any memory BIST controllers generated with software version 7.0-SP01 or older.

- When generating the manufacturing patterns in the TSDB directory, Tessent Shell issues a warning if it detects that some memories without BIRA are not tested in the Pre-Repair memory BIST patterns. If this warning is present, you must run the Post-Repair manufacturing pattern to verify the memories without BIRA.

Because many testers can generate only one clock period and its integer multiples at a time, a separate Patterns or MemoryBist wrapper is generated by default for each clock group. Each wrapper generates a separate test pattern file (for example, WGL file). Typically, each test pattern file includes a TAP reset at the beginning of a pattern. However, this is NOT the case for Pre-Repair patterns (exclude_initial_ireset_from_patterns: on). The reason is that the state of all BIST controllers of all clock groups must be preserved until the Repair group of test patterns is completed.

If a failure occurs during any of the test patterns of the Pre-Repair group, the chip is declared bad, and the tester can proceed to test another chip. However, if all Pre-Repair test patterns pass, the tester can proceed to the Repair group of test patterns.

The figure below shows an example of Pre-Repair test patterns.

Figure 5-40. Pre-Repair Test Patterns Example

```
Patterns(MemoryBist_PreRepair_P1) {
    tester_period : 7.0;
    tck_ratio : 4;
    ClockPeriods {
      clka : tester;
    }
    AdvancedOptions {
      exclude_initial_ireset_from_patterns : on;
    }
    TestStep(TP1_1_PreRepair) {
      AdvancedOptions {
        network_end_state : reset;
      }
      MemoryBist {
        run_mode : hw_default;
        reduced_address_count : off;
        Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
          RepairOptions {
            check_repair_status : non_repairable;
          }
          DiagnosisOptions {
            compare_go : on;
          }
        }
      }
    }
  }
```

Repair Test Patterns

The Repair group includes the following four test steps:

- The first step (*TP2_1_CheckRepairNeeded*) checks RepairStatus[0] of all memories with spare resources. If different than 0, the pattern fails and repair is needed. Note that check_repair_status and split_patterns_file are the only properties allowed in this step. (For exceptions, see the "Pre-Repair Test Patterns" section regarding controllers generated with software versions 7.0-SP03 and Tessent v9.0.) If a compare failure occurs, the tester proceeds with the next step.

- The second step (*TP2_2_CaptureBira*) of the Repair group transfers the contents of the built-in repair analysis (BIRA) registers to the BISR registers. The load_bisr_chain autonomous mode of operation performs this step in a single clock cycle and is recommended to optimize test time.

- The third programming step of the Repair group (*TP2_2_SelfFuseBoxProgram*) takes the contents of the BISR chain, compresses them, and programs the fuses. The time required to perform this step is a function of the write duration time (fuse_box_write_duration: *time*) and the number of fuses available for memory repair (number_of_fuses_for_repair: *int* | auto) that are specified in the PatternsSpecification and DftSpecification respectively. For example, if 256 fuses are available, and it takes

10us to program each fuse, the test pattern takes approximately 1.3ms to run because, on average, only 50% of fuses are assumed to need programming. After this time, the Go and Done outputs of the BISR controller are compared to 1, and a miscompare indicates a bad chip. Otherwise, the next step is performed. Note that fuse_box_write_duration can be overridden in the PatternsSpecification TestStep/MemoryBisr/Controller wrapper if the time to program a single fuse needs to be adjusted.

- The fourth verification step (*TP2_3_VerifyFuseBox*) reads the fuse box content, decompresses it, and applies the decompressed repair information to the input of the BISR chain while comparing the input to the output of the BISR chain. (The output still contains the repair information calculated by the BIRA circuit.) Note that for memories with a serial repair interface, the internal BISR chain is selected as the reference for the comparison with the decompressed repair information. The BISR chain contains a copy of the repair information after the first step of the Repair group (*TP2_2_CaptureBiraRotate*). Note that this test step does not always generate a separate test pattern (*TP2.3*) as shown in Figure 5-38. Refer to the "Tester Settings Considerations" section for more details.

The figures below show examples of Repair test patterns.

Figure 5-41. Example Configuration for TP2_1 Repair Test Patterns

```
Patterns(MemoryBist_CheckRepairNeeded) {
  ClockPeriods {
    clka : tester;
    clkb : tester;
  }

  AdvancedOptions {
    exclude_initial_ireset_from_patterns : on;
  }
  TestStep(TP2_1_CheckRepairNeeded) {
    AdvancedOptions {
      network_end_state : reset;
    }
    MemoryBist {
      run_mode : check_repair_needed;
      Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
      }
      Controller(chip_rtl_tessent_mbist_c2_controller_inst) {
      }
    }
  }
}
```

Figure 5-42. Example Configuration for TP2_2 Repair Test Patterns

```
TestStep(TP2_2_CaptureBira) {
  AdvancedOptions {
    split_patterns_file : on;
  }
  MemoryBisr {
    run_mode : autonomous;
    Controller(chip_rtl_tessent_mbisr_controller_inst) {
      AutonomousOptions {
        operation : load_bisr_chain;
      }
    }
  }
}


TestStep(TP2_2_SelfFuseBoxProgram) {
  MemoryBisr {
    run_mode : autonomous;
    Controller(chip_rtl_tessent_mbisr_controller_inst) {
      AutonomousOptions {
// Fuse box programming is disabled by default in manufacturing
// patterns to avoid accidental fuse programming.
// Change the operation to 'self_fuse_box_program' to enable
// fuse box programming.
        operation : self_fuse_box_program;
      }
    }
  }
}
TestStep(TP2_3_VerifyFuseBox) {
  AdvancedOptions {
    split_patterns_file : on;
  }
  MemoryBisr {
    run_mode : autonomous;
    Controller(chip_rtl_tessent_mbisr_controller_inst) {
      AutonomousOptions {
        operation : verify_fuse_box;
      }
    }
  }
}
```

Post-Repair Test Patterns

The Post-Repair group of test patterns is similar to the Pre-Repair group.

The main difference is that no memory failures are allowed in any of the test patterns, and the GO output of all memory BIST controllers is checked (compare_go: on). Another significant difference is that each test pattern, one per clock group, is self contained. That is, the chip can be powered down between each clock group, and the BISR chain is initialized with the repair information programmed in the fuse box.

The default configuration file specifies the power_up_emulation operation to read the repair information. This means that reading of the fuse box is performed using the TAP clock. However, you might choose instead to use the power_up_emulation operation using the system clock available during power up. A ProcedureStep, which can call an SVF file or an iProc to modify pin settings, along with combinations of ProcedureStep tester_cycles, tck_cycles, or wait_time properties can be added to your configuration file to that effect as explained in the "Functional Mode Verification" section.

Test Time Reduction Options

You only include controllers testing memories with repair in the test program. For memories using multiple power domain groups, you only load the necessary BISR chains for each clock group.

The figure below shows an example of Post-Repair test patterns.

Figure 5-43. Example Configuration for Post-Repair Test Patterns

```
Patterns(MemoryBist_P1) {
  tester_period : 7.0;
  tck_ratio : 4;
  ClockPeriods {
    clka : tester;
  }
  TestStep(load_bisr) {
    MemoryBisr {
      run_mode : autonomous;
      Controller(chip_rtl_tessent_mbisr_controller_inst) {
        AutonomousOptions {
          operation : power_up_emulation;
        }
      }
    }
  }
  TestStep(TP3_1_run_time_prog) {
    MemoryBist {
      run_mode : run_time_prog;
      reduced_address_count : off;
      Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
        DiagnosisOptions {
          compare_go : on;
          compare_go_id : on;
        }
      }
    }
  }
}
```

Tester Settings Considerations

This topic outlines important points that should be reviewed for proper tester configuration and execution of repair patterns.

During execution of Pre-Repair Test Patterns and Repair Test Patterns, the circuit under test must remain powered up without interruption. The state of input pins also should remain constant between test patterns. The state of the BIRA and BISR registers must be preserved from one test pattern to the next.

The Post-Repair Test Patterns generated for each clock group are independent, and the circuit under test can be powered down between test patterns. The state of the BISR registers is restored at the beginning of each test pattern.

During execution of Repair test patterns, it might be necessary to drive the programming voltage pin of the fuse box from a dedicated power supply. This is the case for fuses requiring programming currents exceeding the drive capacity of tester data channels (20mA is typical). The TestStep of the PatternsSpecification where it is necessary to change the settings of the dedicated power supply contain an optional property, force_voltage(*pin_name*):*value*, that causes splitting of the test patterns as shown in Figure 5-38. The TP2.2 test pattern including

test steps *TP2_2_CaptureBira* and *TP2_2_SelfFuseBoxProgram* is performed with the dedicated power supply set at the voltage required to program the fuse box, whereas *TP2_1_CheckRepairNeeded* and *TP2_3_VerifyFuseBoxTP2.3* are run with the voltage set to allow reading the fuse box. The test pattern file (for example, WGL) contains a comment indicating that the settings of the power supply need to be changed.

The first step of the Repair group (*TP2_1_CheckRepairNeeded*) needs to be in a separate test pattern so that a simple pass/fail criterion can be used to identify the good chips from the chips that require repair. Significant test time can be saved this way in some circumstances. Because this test step is the first in the MembistPVerify wrapper, the beginning of the test pattern is already defined. However, the end of this test pattern must be identified by a force_voltage property as explained in the previous paragraph, or by a split_patterns_file: on property setting inserted in the next test step (*TP2_2_CaptureBira*).

Manufacturing Flow Variations

Several variations of the manufacturing flow are possible to optimize yield and test time.

One variation of the flow illustrated in Figure 5-38 is described in the section "Testing Repair Solution Before Fuse Programming" that enables testing the repair solution prior to fuse programming. The section "Performing Fuse Programming During Any Test Insertion" describes another variation that enables performing fuse programming during any test insertion (for example, final test), provided that the fuse array has never been programmed before.

Testing Repair Solution Before Fuse Programming

Testing your repair solution before programming the fuses might save some test time for circuits using fuses with very long programming times. Referring to Figure 5-44, the variation requires inserting the Post-Repair block of patterns (TP3.n) prior to the fuse programming pattern (TP2.3). The *BISR Power up* operation (AutonomousOption/operation: power_up_emulation) has been removed from the Post-Repair patterns. Therefore, the circuit cannot be powered down during execution of these patterns, and the initial TAP reset must be inhibited as for the Pre-Repair block of patterns (TP3.n) (exclude_initial_ireset_from_patterns: on) to preserve the BISR chain state.

If compare failures occur during execution of this new block of patterns, then at least one memory is still not functional after repair. The cause is probably bad spare resources, in which case the circuit should be discarded. If no compare failures occur, then the repair solution is effective and fuse programming can be performed.

TP2.2 is different for memories with a serial or parallel BISR interface. The BIRA to BISR transfer for memories with a parallel BISR interface can be performed quickly using the load_bisr_chain mode of operation. For memories with a serial BISR interface, it is necessary to perform a full BISR chain rotation using the rotate_bisr_chain mode of operation and specifying enable_bira_capture : on.

Figure 5-44. Flow Variation: Testing Repair Solution Before Fuse Programming
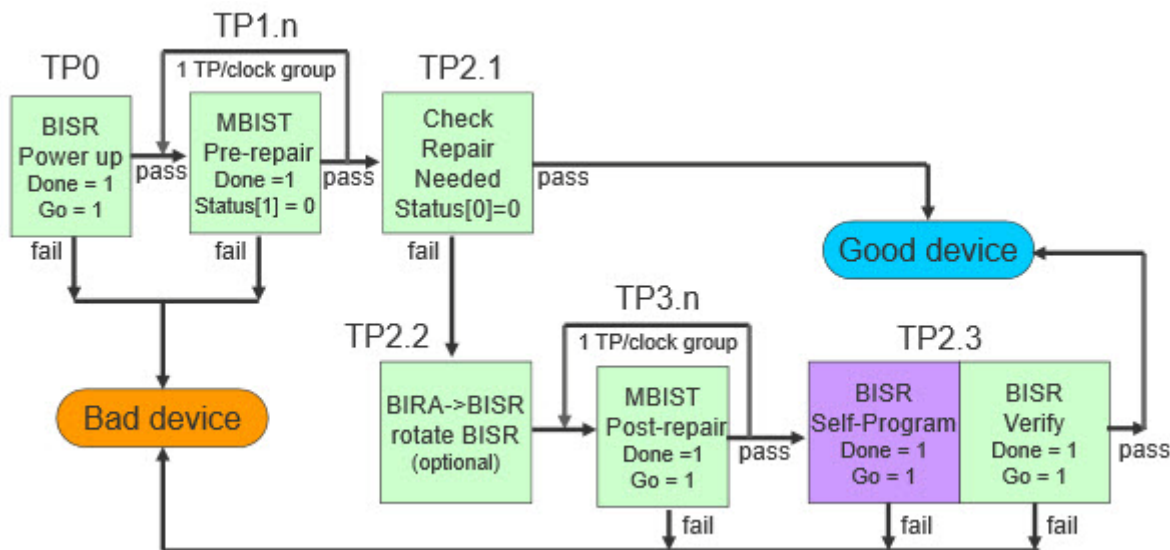
Performing Fuse Programming During Any Test Insertion

The test flows of Figure 5-38 and Figure 5-44 normally are used during the first test insertion (that is, wafer test to increase yield). During subsequent test insertions (for example, final test), the Post-Repair test patterns (that is, TP3.n) are reapplied, possibly under different test conditions to verify the quality of the circuits. Circuits failing at this test insertion are discarded. The number of circuits usually is small so that yield is not affected significantly. However, repairing circuits during any test insertion is possible (provided the circuits have not been repaired previously) by first checking if the fuse box has been programmed previously.

This is done by reading a number of fuses starting at address 0. The number is determined by the largest of the MemoryBisr/Controller/AdvancedOptions/repair_word_size and zero_counter_bits, and is usually less than 32. The following shows an example of a TestStep wrapper reading the first 32 locations of the fuse box and expecting a value of 0:

```
TestStep (membisr_CheckBlankFuseBox) {
   MemoryBisr {
     run_mode: fuse_box_access;
     Controller(chip_rtl) {
       FuseBoxAccessOptions {
         read_address(31:0): 0;
         operation : read;
       }
     }
   }
}
```

If all fuses return a value of 0, then the fuse box was never programmed and the repair flow of Figure 5-38 can be performed as shown by the "pass" path in Figure 5-45 below. Otherwise, the Post-Repair test pattern is run.

Figure 5-45. Flow Variation: Performing Fuse Programming During Any Test Insertion
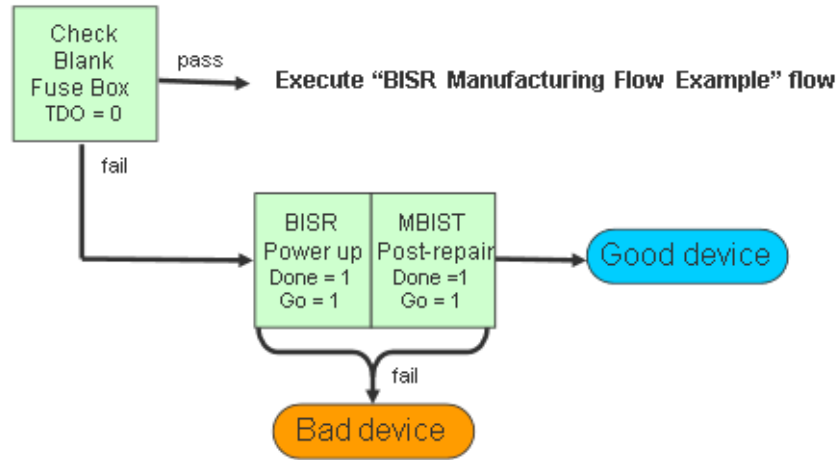
BISR Chain Test and Diagnosis

The default BISR implementation in Tessent Shell MemoryBIST only tests the connections necessary to repair memories during the post-repair steps. The following sections describe how to perform a complete test of the BISR chains, as well as connections to and from the BIRA registers.

In high-reliability systems, you may want to structurally test all connections during manufacturing test to maximize the probability of being able to perform an incremental repair in the field. Additionally, the BISR chain test can also be used to debug data connection issues between the BISR and BIRA registers as well as connection issues related to control signals, such as shift enable, reset, clock, and chain selection.

The main faults targeted by the BISR chain test are highlighted in the figure below by the red dots and red lines. The remaining faults of the BIRA registers are covered using ATPG scan tests. The location of the faults other than those directly on the shift data path can be easily identified.

Figure 5-46. BISR Chain Test Fault Targets
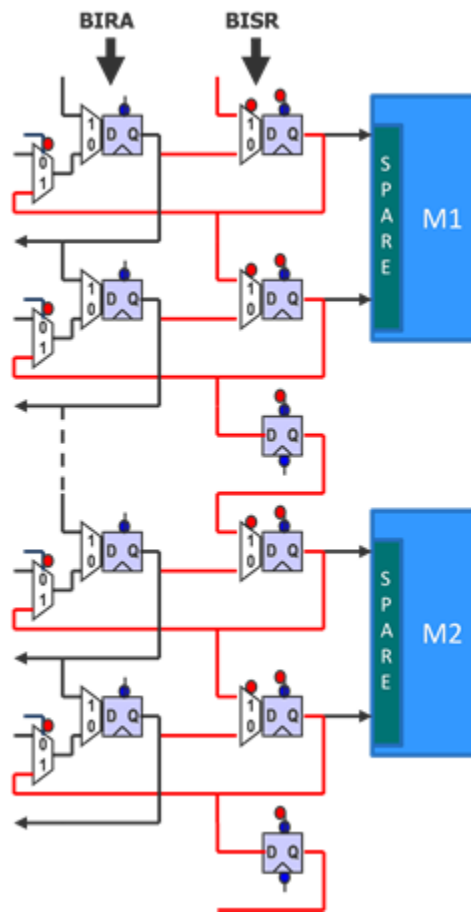
Enabling BISR Chain Tests

BISR chain tests are automatically generated when creating the PatternsSpecification if the default value for the include_repair_chain_test property is set to on. The property can be set independently for either manufacturing or signoff patterns as shown below:

```
DefaultsSpecification(<policy>) {
  PatternsSpecification {
    SignOffOptions {
      simulate_instruments_in_lower_physical_instances : off;
      MemoryBisr {
        include_repair_chain_test : on;  // Default is off
      }
    }
    ManufacturingOptions {
      MemoryBisr {
        include_repair_chain_test : on;  // Default is off
      }
    }
  }
}
```

These options can also be set prior to issuing the create_patterns_specification command as follows:

```
SETUP>set_defaults_value PatternsSpecification/SignOffOptions/MemoryBisr/
include_repair_chain_test on

SETUP>set_defaults_value PatternsSpecification/ManufacturingOptions/
MemoryBisr/include_repair_chain_test on

SETUP>set_defaults_value PatternsSpecification/SignOffOptions/
simulate_instruments_in_lower_physical_instances off
```

The simulate_instruments_in_lower_physical_instances property is off by default to reduce the time needed to complete signoff simulations. In this configuration, only the BISR registers are included in a simplified model of lower-level physical blocks. These BISR registers will capture unknown values during some of the BISR chain tests because of the absence of BIRA registers in these blocks, however, these unknown values are automatically masked. Therefore, the connections between BIRA and BISR registers can be efficiently verified hierarchically during signoff.

_____ **Note** _____

Manufacturing patterns are applied to the full design hierarchy and completely test all BISR/BIRA connections.

BISR Chain Test Description

Enabling the BISR chain tests, as described in the previous section, creates a new Patterns(BiraBisrChainTest) wrapper in the PatternsSpecification when you run create_patterns_specification. The sections that follow describe the algorithm that the new pattern implements.

The portion of the algorithm described in the first section is common to all memories, whereas the portion described in the second section only applies to memories with a serial repair interface, or to memories that are part of a shared bus.

BISR Chain Common Test Algorithm

The algorithm steps implemented by the BISR chain test Patterns wrapper that is common to all memories will be outlined. These steps provide a complete test of the BISR shift path, reset and shift enable inputs, as well as the connections to and from the BIRA registers. The connections are tested for stuck-at 0/1, as well as for shorts with immediate neighbors using a checkerboard test pattern.

Each algorithm step outlined below corresponds to a TestStep wrapper shown in Figure 5-47, which shows the first three TestSteps of the corresponding Patterns wrapper implementing the BISR chain test.

1. Clear BISR chains

2. Scan out 0s and scan in a checkerboard pattern

3. Perform a BISR to BIRA transfer

4. Clear BISR chains

5. Scan out 0s and scan in an inverse checkerboard pattern

6. Perform a BIRA to BISR transfer

7. Scan out checkerboard and scan in an inverse checkerboard pattern

8. Repeat steps 3 through 7 with inverted data

All steps, except step 3 (Tsb_BisrToBira_chckb), perform operations on the BISR chains, as indicated by the presence of the MemoryBisr wrapper.

Step 3 requires running the BIST controllers that contain BIRA registers to perform the BISR to BIRA transfer. In this example PatternsSpecification, there is one controller at the top level and two controllers in lower-level blocks. All controllers are specified in step 3 because the complete model of the lower-level blocks is used. If the simplified model of the lower-level

block was used instead, which is the default, only the controller at the top level would need to be specified. In this case, the values scanned out from the lower-level blocks are automatically masked for certain portions of the algorithm since the values are unknown.

A hardware feature is available to facilitate the BISR to BIRA transfer. The transfer can be initiated by specifying the apply_algorithm : null property in the AdvancedOptions wrapper of the PatternsSpecification. The controller performs the transfer and goes to the Done state without running any memory test algorithm. This operation is very fast since it runs in the hw_default run mode, which results in a small number of test execution cycles being used. The transfer can be performed using functional clocks or TCK, if a mechanism to inject TCK in the functional clock tree exists.

Figure 5-47. Partial Common Algorithm Implementation

```
PatternsSpecification(blka,rtl,signoff) {
…
  Patterns(BiraBisrChainTest) {
    ClockPeriods {
      clk : tester;
      clk2 : tester;
    }
    SimulationOptions {
      LowerPhysicalBlockInstances {
        core_inst1 : full;
        core_inst2 : full;
      }
    }
    TestStep(T1_ClearBisrChain) {
      MemoryBisr {
        run_mode : autonomous;
        Controller(top_rtl_tessent_mbisr_controller_inst) {
          AutonomousOptions {
            operation : clear_bisr_chain;
          }
        }
      }
    }
    TestStep(T2_r_0_w_chkb) {
      MemoryBisr {
        run_mode : bisr_chain_access;
        Controller(top_rtl_tessent_mbisr_controller_inst) {
          BisrChainAccessOptions {
            default_write_value : checkerboard;
            default_read_value : all_zero;
          }
        }
      }
    }
    TestStep(T3a_BisrToBira_chkb) {
      MemoryBist {
        run_mode : hw_default;
// Controllers perform BISR to BIRA transfer and go to Done state
        Controller(core_inst1.core_rtl_tessent_mbist_c1_controller_inst) {
          AdvancedOptions {
            apply_algorithm : null;
          }
        }
        Controller(core_inst2.core_rtl_tessent_mbist_c1_controller_inst) {
          AdvancedOptions {
            apply_algorithm : null;
          }
        }
```

Serial Repair and Shared Bus Algorithm Additions

Memories with serial repair interfaces and those in shared bus implementations require
additional steps to test the select input of the multiplexer that provides selection between the

internal and external BISR register, as well as the reset input of each bit of the internal BISR registers.

The multiplexer implementation for memories with serial repair interfaces is explained in "Built-In Self Repair (BISR)" and shown in Figure 5-2. The multiplexer implementation for shared bus memories is explained in "BIRA and BISR Generation for a Memory Cluster Module" and shown in Figure 6-13.

The additional algorithm steps outlined below are an extension of the common algorithm steps discussed in "BISR Chain Common Test Algorithm":

9) Clear BISR chains

10) Scan out 0s from internal BISR registers and scan in 1s

11) Scan out 1s from internal BISR registers and scan in 1s

12) Perform a BISR to BIRA transfer

13) Reset all BISR registers

14) Perform a BIRA to BISR transfer

15) Scan out 0s from the internal BISR registers

These steps are similar to those performed in the common algorithm except that the internal BISR registers are selected during scan-out operations. The internal BISR registers are selected by specifying the BisrChainAccessOptions/select_bisr_registers property to internal in the PatternsSpecification for steps 10 and 11, and to internal_only for step 15. The test is arranged to load different contents in the external and internal BISR registers to prove the existence of the multiplexer selecting between the two.

In step 14, the external BISR registers capture the output of the corresponding BIRA registers. The internal BISR registers might hold or shift depending on their implementation for this operation. It is assumed that all bits of the internal BISR registers have a reset input so that the expected output of step 15 is predictable. This assumption is not true for some memories which only have a reset on the repair enable bit of the internal BISR register. The test cannot be applied in this case.

Some BISR chains might be composed of a mixture of memories with parallel and serial repair interfaces, or memories that are part of a shared bus. BISR registers of memories with a parallel repair interface behave as external BISR registers. Steps 10 and 11 are tolerant of this situation, as the values scanned out from the internal or external registers are the same. However, these values differ in step 15. To ensure that only the values scanned out from the internal registers are compared, the select_bisr_registers property must be set to internal_only. The internal_only value scans out BISR registers associated to memories with either a serial or parallel memory

repair interface, however only the BISR registers associated to memories with a serial repair interface will be compared.

BISR Chain Test Limitations

The following are BISR chain test limitations that have been identified:

- Some memories with a serial repair interface only have a reset on the repair enable bit of the internal BISR register. These memories are partially reset and the additional test sequences described in "Serial Repair and Shared Bus Algorithm Additions" cannot be applied.

- Custom and pipeline BISR registers that do not have a BIRA counterpart require modifications to the pattern described in the "BISR Chain Common Test Algorithm" section. You can create custom and pipeline registers using the create_custom_bisr_register command, the pipeline keyword in the bisr_segment_order_file, or by specifying the BisrElement/Pipeline(before | after) wrapper. The pattern modification consists of changing the expect value of these registers to 0 in step 7 of the algorithm using the "read_value(*range*) : 0" property and setting in the BisrChainAccessOptions/BisrRegisterAccessOptions wrapper.

Compression Algorithm and Fuse Box Organization

Normally the information in this section is not necessary for implementing BISR but can be useful for diagnosing problems.

The compression algorithm that the BISR controller uses relies on the fact that a majority of memories do not need repair and that the values stored in the BISR registers are mostly 0's. Therefore, the compressed repair information consists of two types of words stored in the fuse box: *Zero Count* and *Repair Data*. The first bit of each word, also called *Opcode*, indicates its type. *Zero Count* words have an *Opcode* of 0, and *Repair Data* words have an *Opcode* of 1. The length of each word depends on parameters that are automatically extracted from your circuit. The length of *Zero Count* words is set to $1 + \text{Log}_2$ (*MaxChainLength*) where *MaxChainLength* is the length of the BISR chain, which can be derived from the write_memory_repair_dictionary command. The length of *Repair Data* words is set to the longest BISR register length, which is extracted automatically from the circuit and reported in the MBISR TCD file as shown in the example of Figure 5-27 and described in the "Single-Chain Case" section.

In a *Repair Data* word, the first bit is part of the repair data in addition to being an *Opcode*. *Repair Data* words do not necessarily align with BISR register boundaries because the controller does not know the location of those boundaries. For this reason, a *Repair Data* word can be incomplete when it is the last word in the BISR chain.

Figure 5-48(a) shows a typical example of repair information contained in the fuse box for a circuit requiring a single repair. The words are written to the fuse box from left to right, one bit at a time. That is, fuse box address 0 is on the left. The controller assumes that each fuse has its own address. For fuse boxes that are word oriented, the fuse box interface translates the fuse box address from the controller to a word address that is suitable for the specific fuse box and then programs a bit within that word.

The example shown is for a single power domain group. Any combination of *Zero Count* and *Repair Data* words might be used.

When multiple power domain groups are present, the tool compresses the data of each group as described above. In addition, the BISR controller stores group pointers in the fuse box to find the repair information specific to groups other than the first group. Figure 5-48(b) shows that each pointer requires a number of fuses, denoted as *NumberOfAddressBits*, equal to the number of fuse box address bits. You specify the number of fuse box address bits with the TCD FuseBoxInterface/Interface address property. The group pointer is stored MSB to LSB, meaning the MSB of the pointer is at the lower address in the fuse box. Figure 5-48(c) shows that the BISR controller inserts all pointers at the beginning of the fuse box, starting at address 0, for the default configuration of a single programming session (max_fuse_box_programming_sessions = 1).

Figure 5-48. Fuse Box Organization
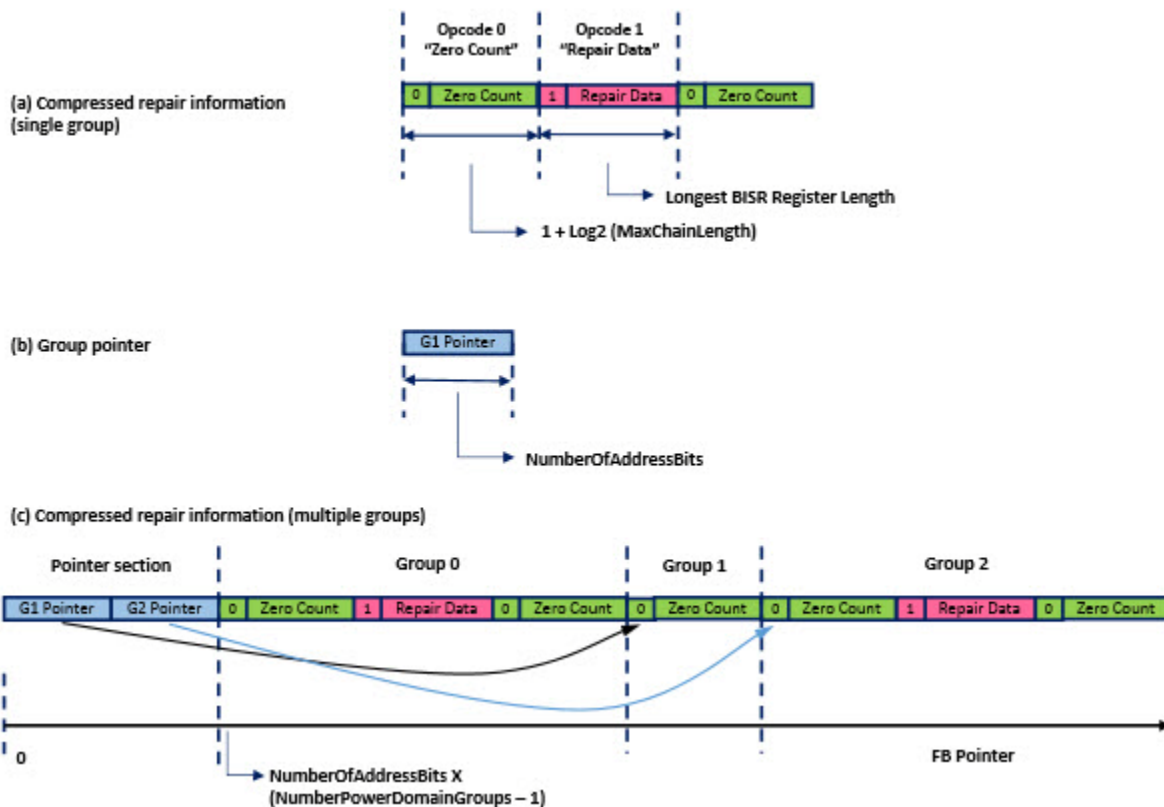
You can use the CompressBisrChain script, which is provided in the Tessent products release, to compress the content of the BISR chain using the compression algorithm and the fuse box organization that is specific to your design. For more information about how to use the *CompressBisrChain* script, see the "CompressBisrChain Script Usage" section.

When you use hard incremental repair (max_fuse_box_programming_sessions > 1), additional data is stored in the fuse box to indicate how many times the fuse box was programmed and where the repair data is located for each programming session. Figure 5-49 shows an example for max_fuse_box_programming_sessions = 4. The first four bits are session flags. A session flag is set to 1 when repair data is present for the session. The BISR controller stores the session flag for the fourth session at address 0 in the fuse box and the first session flag is stored at address 3.

The next block of fuses is a test insertion pointer section. Except for the first session, each session has one test insertion pointer that is in the fuse box immediately after the session flag section. Like the power domain group pointer, the test insertion pointer requires a number of fuses equal to the number of fuse box address bits. The fuse box stores the test insertion pointer MSB to LSB, meaning the MSB of the pointer is at the lower address in the fuse box.

The content for each programming session that follows the test insertion pointer section is organized the same way as shown in Figure 5-48. Each programming session is self-contained. In other words, none of the repair information of previous sessions is reused when using the autonomous fuse programming method.

Figure 5-49. Fuse Box Organization (max_fuse_box_programming_sessions > 1)
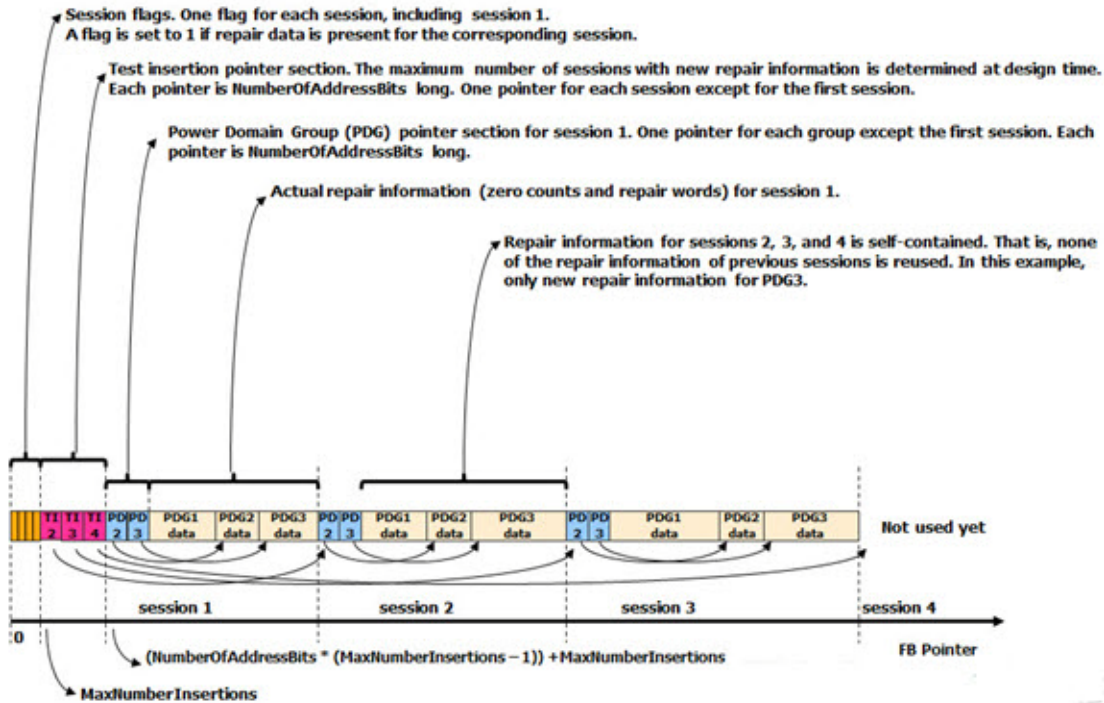
CompressBisrChain Script Usage

The *CompressBisrChain* script is primarily used for designs incorporating an eFuse that cannot be programmed on-chip using the autonomous run mode of the BISR controller, and must be programmed externally. A secondary benefit of using the *CompressBisrChain* script is that for multi-session fuse programming (hard incremental repair), optimization techniques are included that reduce the number of fuses when Power Domain Groups (PDGs) with the same BISR chain length are present. In this case, additional optimization techniques reuse fuses from previous sessions whenever possible. Using the *CompressBisrChain* script makes the manufacturing process slightly more complex in that the repair data must be extracted, processed by the script, and re-loaded in the chip for fuse programming. This process is outlined later in this section.

The *CompressBisrChain* script reduces the number of fuses by copying the pointer to PDG data of the previous programming session if no new repair is necessary for a specific PDG. The same pointer can be reused for multiple sessions. For the example implementation presented, the first

group is defined with the DftSpecification PowerDomainOptions/ power_domain_priority_order property, which presents a limitation. The pointer for that group is hardcoded in the BISR controller and cannot be modified. Therefore, the repair information for this group is not shared by different sessions. To minimize the impact of this limitation, assign the highest priority to the smallest PDG. This lessens the probability that this PDG needs to be repaired.

Figure 5-50 compares the FuseBox Access and Autonomous Self Fuse Box Program methods for the example shown in Figure 5-49. For Session 1, the two methods give the same result, assuming that all PDGs are of different lengths. In Session 2, reusing the repair data of PDG2 is possible because no new repair is necessary for that group. Therefore, the pointer for PDG2 is copied into Session 2 and points back to the repair information in Session 1. The actual repair data is not copied.

The situation is different for PDG1, however. Because of the limitation previously described, the repair data must be copied even if the data is identical. A new block of repair data is written for PDG3 because additional repairs are necessary. In Session 3, only PDG1 requires new data. The pointers for PDG2 and PDG3 are copied. PDG2 points back to Session 1 data, and PDG3 points back to Session 2 data.

All pointers to existing data are shown in red in Figure 5-50 and point backwards to the fuse box address space. In this example, only three of the four available sessions are used, leaving the possibility that fuses can be programmed in the system. Most likely you would use the autonomous mode to program the fuses because this is easier than using the *CompressBisrChain* script, but you can use either method. The difference is that the autonomous mode does not reuse any of the repair information of previous sessions.

Figure 5-50. Encoding of Repair Information Using CompressBisrChain Script
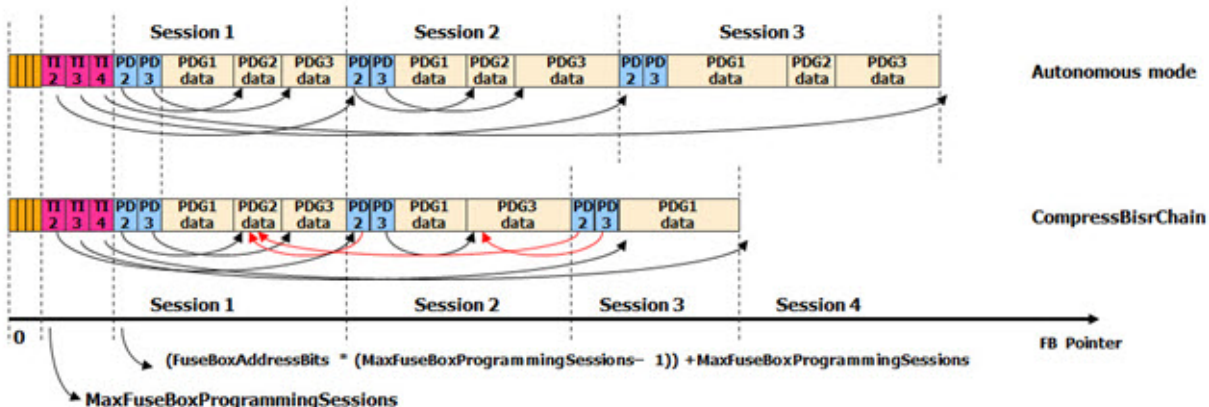
When using the *CompressBisrChain* script for the second or subsequent fuse programming session (incremental repair), one step must be added to the manufacturing flow that reads the existing fuse box content prior to writing the new fuses. This step is shown in Figure 5-52. Once repairs are determined to be necessary, the entire fuse box is read, one bit at a time, using the fuse_box_access run mode of the BISR controller. To save test time, only the fuse box portion reserved to store memory repair information should be read. You can do this by specifying the

read_address (*range*) property in the PatternsSpecification MemoryBisr controller
FuseBoxAccessOptions wrapper, as shown in the following example where the fuse box has
1024 fuses but only 512 are reserved for memory repair.

Figure 5-51. Fuse Box Access Read Fuses TestStep

```
TestStep(membisr_CheckRepairNeeded) {
  ...
}
TestStep(membisr_ExtractFuseBoxSessionInfo) {
  MemoryBisr {
    run_mode: fuse_box_access;
    Controller(FB_INST1) {
      FuseBoxAccessOptions {
        operation : read;
        // Restrict read to the first 512 locations with
        // compare to "0".
        read_address (511:0): 0 ;
      }
    }
  }
}
```

The miscompares on TDO indicate where the value 1 is programmed in the fuse box. The script
uses this information for the following:

- To find the pointer to the last session with repair information.

- To find all PDG pointers if more than one group exists.

- To find the repair information associated with each group.

- To calculate the next available fuse location to write the new repair information.

The new repair solution is then extracted as usual by performing the BIRA-to-BISR transfer and
rotating the BISR chain while comparing TDO to 0 to find which flip-flops of the BISR chain
contain the value 1. The script uses this information to compare the new compressed repair data
for each group and determine if any existing repair data can be reused.

Figure 5-52. Manufacturing Flow Using CompressBisrChain Script (Incremental Repair)

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.
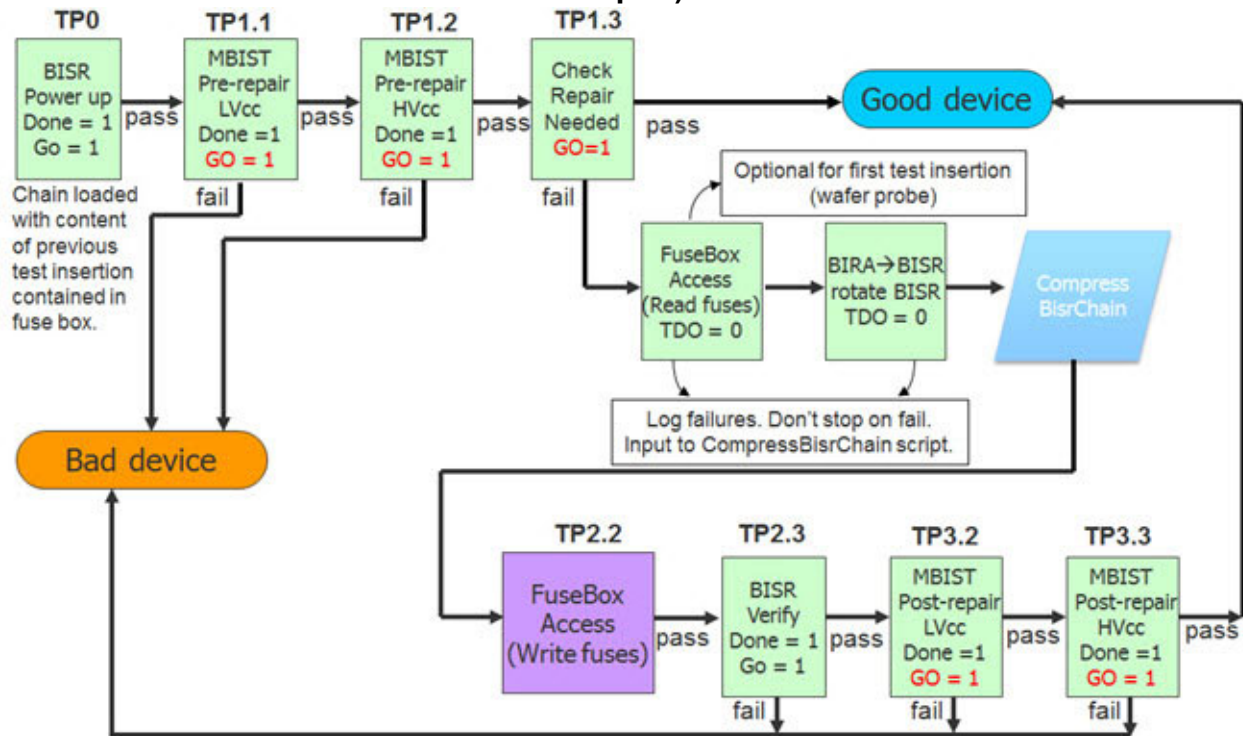
CompressBisrChain

The CompressBisrChain script, which is provided in the Tessent product release, is used to compress the content of the BISR chain using the compression algorithm and fuse box organization specific to your design.

Usage

CompressBisrChain **-simLog** *simLog_filename*
  [-configFile *config_filename*]
  [-outDir *outDir_name*]
  [-outExt *outFile_extension*]
  [-TestStepSuffix *suffix*]
  [-help]

Description

Designs with repairable memories in which a Tessent BISR controller is implemented, have two methods available for performing memory repair. The first method uses the autonomous run_mode fuse box programming method. This method performs a BISR chain compression and programming of the fuse box on-chip. The second method is done using the TAP access mode, specified by the fuse_box_access run_mode property. Fuse box programming using the TAP access mode requires some automation that is provided by the CompressBisrChain script.

Programming the fuse box using the TAP access mode requires executing the following steps:

* Extraction of the BISR chain information from a pattern output

* Compression of the BISR chain data

* Programming the fuse box using the TAP access mode

The CompressBisrChain script reads a configuration file that describes the BISR chain settings in your design as described in the -configFile switch description. The script also reads a simulation log file created from a PatternsSpecification, such as the one described in the Examples section.

The script creates four **.pattern_spec* files in the current working directory with content ready to insert into the PatternsSpecification. These files define individual pattern TestSteps that you can use to:

* Program the fuse box

* Read the fuse box data

* Program the BISR chain

* Shift out the BISR chain

Arguments

- **-simLog** *simLog_filename*

  A required switch and string pair that specifies the name of the simulation log input file. The BISR chain information is extracted from this file.

- -configFile *config_filename*

  An optional switch and string pair that specifies the location of the BISR parameter configuration file. The default setting of this option is:

  <current_working_directory>/BisrCtrlParams.tcl

  The BISR parameter configuration file is created with the write_memory_repair_dictionary command after ICL extraction has been performed on the design.

- -outDir *outDir_name*

  An optional switch and string pair that specifies the directory for saving the generated files. The default location is the current directory.

- -outExt *outFile_extension*

  An optional switch and string pair that specifies the optional suffix appended to the generated file names. The suffix can only contain alpha-numeric characters, plus "-", "_", and "." characters. By default, no suffix is appended. The default names of the generated files are: BisrChainProgram.pattern_spec, BisrChainRead.pattern_spec, FuseBoxProgram.pattern_spec, and FuseBoxRead.pattern_spec.

- -TestStepSuffix *suffix*

  An optional switch and string pair that specifies the suffix appended to the test step names. This is used to create unique test step names in the generated output files when running this script on designs with incremental repair. By default, no suffix is added.

- -help

  An optional switch that displays the usage documentation.

Examples

The following PatternSpecification examples follow the manufacturing flow using the CompressBisrChain script as outlined in Figure 5-52. Only a single pre-repair pattern (TP1.1) and post-repair pattern (TP2.3) is shown. Subsequent repair sessions include the FuseBox Access (Read fuses) flow step, which is shown in Figure 5-51 but not included in this example.

```
    Patterns(MemoryBist_PowerUp) {
      TestStep(TP0_PowerUp) {
        MemoryBisr {
          run_mode : autonomous;
          Controller(chip_rtl_tessent_mbisr_controller_inst) {
            AutonomousOptions {
              operation : power_up_emulation;
            }
          }
        }
      }
    }
    Patterns(MemoryBist_PreRepair) {
      tester_period : 7.0;
      tck_ratio   : 4;
      ClockPeriods {
        clka : tester;
      }
      AdvancedOptions {
        exclude_initial_ireset_from_patterns : on;
      }
      TestStep(TP1.1_PreRepair) {
        AdvancedOptions {
          network_end_state : reset;
        }
        MemoryBist {
          run_mode : hw_default;
          reduced_address_count : off;
          Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
            RepairOptions {
              check_repair_status : non_repairable;
            }
            DiagnosisOptions {
              compare_go : on;
            }
          }
        }
      }
    }
    Patterns(MemoryBist_CheckRepairNeeded) {
      ClockPeriods {
        clka : tester;
      }
      AdvancedOptions {
        exclude_initial_ireset_from_patterns : on;
      }
      TestStep(TP1.3_CheckRepairNeeded) {
        AdvancedOptions {
          network_end_state : reset;
        }
        MemoryBist {
          run_mode : check_repair_needed;
          Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
          }
        }
      }
    }
```

```
Patterns(BisrChainAccess) {
  TestStep(BiraToBisr) {
    MemoryBisr {
      run_mode : bisr_chain_access;
      Controller(chip_rtl_tessent_mbisr_controller_inst) {
        BisrChainAccessOptions {
          enable_rotation : on;
          enable_bira_capture : on;
          select_bisr_registers : external;
          default_read_value : all_zero;
        }
      }
    }
  }
}
```

In the MemoryBist_PreRepair TestStep (TP1.1), MemoryBIST tests all memories and determines if any memory cannot be repaired, in which case the chip is declared as bad. The BIRA controller collects failure information and computes repair solutions for repairable memories during this process. The MemoryBist_CheckRepairNeeded TestStep (TP1.3) then checks the Repair_Status[0] register of all memories for a high level to determine if a repair is needed.

If you need to repair memories, run the BisrChainAccess pattern on the tester. The BiraToBisr TestStep loads the computed BIRA repair solution into the BISR registers and rotates the registers. The tester compares the TDO output to zero during the rotation and reports the mismatches. You must then convert the reported failures into a "simulation.log" file that follows the format highlighted in red:

```
...
# 556000ns: Executing FuseBox Controller blockA_rtl_tessent_mbisr_controller_inst in BISR
Chain Access Mode
# 556000ns: Enabling BIRA capture
# 559624ns: Mismatch at pin 0 name ijtag_so, Simulated 1, Expected 0
# 559700ns: Previous scan out : pin ijtag_so , ICL register = mem1_bisr_inst.ShiftReg[8]
# 559824ns: Mismatch at pin 0 name ijtag_so, Simulated 1, Expected 0
# 559900ns: Previous scan out : pin ijtag_so , ICL register = mem1_bisr_inst.ShiftReg[10]
...
```

You run the CompressBisrChain script and specify the "simulation.log" file you created as an input to the script, as well as the BISR parameter configuration file with the -configFile switch. The script parses the "simulation.log" file, extracts the information highlighted in red and generates the four *.pattern_spec output files.

You use the TestStep within the FuseBoxProgram.pattern_spec file, created by the CompressBisrChain script, for writing the fuse box in TP2.2 of the manufacturing flow.

Example patterns for this step, and the remaining steps in the flow to verify the fuse box contents and perform post-repair testing, are as follows:

```
Patterns(FuseBoxAccess) {
  TestStep(TP2.2_FuseBoxAccess){ // Insert TestStep from the
                                 // CompressBisrChain
                                 // FuseBoxProgram.pattern_spec
                                 // output file.
  }
}
Patterns(BisrAutonomousMode) {
  TestStep(TP2.3_BISRVerify) {
    MemoryBisr {
      run_mode : autonomous;
      Controller(chip_rtl_tessent_mbisr_controller_inst) {
        AutonomousOptions {
          operation : verify_fuse_box;
        }
      }
    }
  }
}

Patterms(MemoryBist_PostRepair) {
  tester_period : 7.0;
  tck_ratio  : 4;
  ClockPeriods {
    clka : tester;
  }
  AdvancedOptions {
    exclude_initial_ireset_from_patterns : on;
  }
  TestStep(TP3.2_PostRepair) {
    MemoryBist {
      run_mode : run_time_prog;
      reduced_address_count : off;
      Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
        RepairOptions {
          check_repair_status : off;
        }
        DiagnosisOptions {
          compare_go : on;
          compare_go_id : on;
        }
      }
    }
  }
}
```

Incremental Repair

Incremental repair can be utilized to improve manufacturing yield or avoid costly system repair in the field. Incremental repair takes one of two forms: soft or hard. Specifying an option to enable soft incremental repair is not necessary as the connections that allow transferring the contents of the BISR registers to the BIRA registers are always present and enable incremental repair.

Incremental Repair Overview

You can use incremental repair to improve manufacturing yield or to avoid costly system repair in the field. Incremental repair solutions can be implemented in either a soft form or hard form.

Hard incremental repair involves programming the fuse box in more than one test insertion (for example wafer probe, package test, final test, or system test). By default, hard incremental repair is enabled for a single test insertion. Multiple insertions can be enabled by specifying the max_fuse_box_programming_sessions property in the DftSpecification/MemoryBisr/Controller wrapper with a value greater than 1, which is the default value, or by selecting the "unlimited" option.

```
DftSpecification {
  MemoryBisr {
    Controller {
      ...
      max_fuse_box_programming_sessions : <int> ;
      ...
    }
  }
}
```

When the number of programming sessions is specified with an integer value, the repair information is compressed before being stored in the fuse box, as described in the "Compression Algorithm and Fuse Box Organization" topic. The number of fuses required to store the repair information is proportional to the specified integer value, in that subsequent repair insertions are stored in the remaining unused fuse locations.

When max_fuse_box_programming_sessions is specified as "unlimited", fuse box compression is disabled and the repair solution from the BISR chains is stored uncompressed in the fuse box.

In this case, the fuse box organization is different than that shown in Figure 5-49, in that no session flags, test insertion pointers, or power domain group pointers are stored. Only the BISR register content is stored for each power domain group, including the pipeline register in the BISR controller. Subsequent repair solutions, from different power domains for example, can be written directly into the fuse box without affecting the repair solution from previous insertions. Therefore, the repair solution for each power domain can be individually programmed into the fuse box rather than all at once. This method enables an unlimited number of self_fuse_box_program autonomous mode programming sessions. The number of fuses required for this use must be equal to or greater than the total number of bits in the power domain group BISR chain plus one for the pipeline register, across all power domain groups. Disabling the fuse box compression hardware is also useful in the rare case where more than 50% of the repair registers are expected to be used. In this case, the compression algorithm becomes inefficient and may result in requiring more fuses than there are bits in the BISR chain.

Soft incremental repair consists of finding a repair solution complementing the solution already contained in the fuse box and applying the final solution for as long as the chip is powered up. This method can be used for two applications:

- The main application is to repair a chip in a system that was already repaired during manufacturing. Post-manufacturing failures are rare but can happen as the memory ages or if the chip is operated under conditions that were not tested during manufacturing. A special power supply for programming fuses is not required to perform repair. However, the disadvantage of not having one is that the repair solution must be recalculated each time the system is powered up.

- The second application is to quantify the amount of additional yield, if any, that could be gained at manufacturing time with hard incremental repair.

Figure 5-53 illustrates the flow for soft incremental repair, which is the same for both applications. To simplify Figure 5-53, TP1 and TP3 are not broken down by clock groups. This is also more representative of system applications where all clocks typically are available so that a single pattern can be used for the pre-repair and post-repair patterns.

TP2.2 is different for memories with a serial or parallel BISR interface. The BIRA to BISR transfer for memories with a parallel BISR interface can be performed quickly using load_bisr_chain mode of operation. For memories with a serial BISR interface, it is necessary to perform a full BISR chain rotation using the rotate_bisr_chain mode of operation and specifying enable_bira_capture : on.

Figure 5-53. Soft Incremental Repair Flow
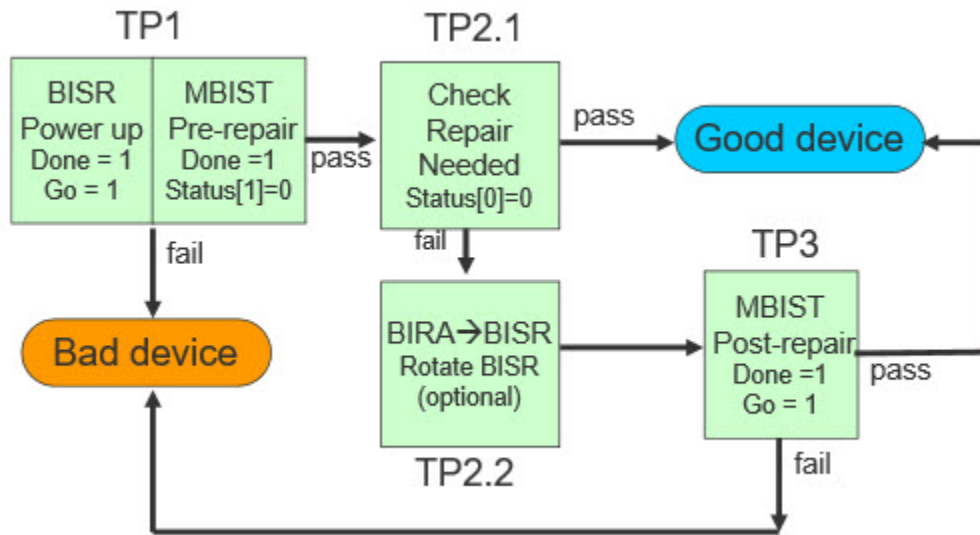
The remainder of this section explains how to implement incremental repair. Although soft incremental repair is explained first, most aspects are applicable to hard incremental repair as well. For information pertaining to hard incremental repair only, refer to the "Considerations Specific to Hard Incremental Repair" section.

BIRA Initialization

After execution of the PowerUp or PowerUpEmulation autonomous operations, the BISR chain is loaded with the current repair solution contained in the fuse box. If the chip has never been repaired, the BISR chain contains only 0s.

When memory BIST is performed and BIRA enabled, the BIRA fuse registers are loaded with the content of the corresponding BISR registers.

Figure 5-54 shows a high-level view of the interface between the BISR and BIRA registers. The connections and register enabling incremental repair are shown in red. The register PastRepair, indicates the spare was allocated in a previous test insertion.

Figure 5-54. BISR-to-BIRA Transfer Connection
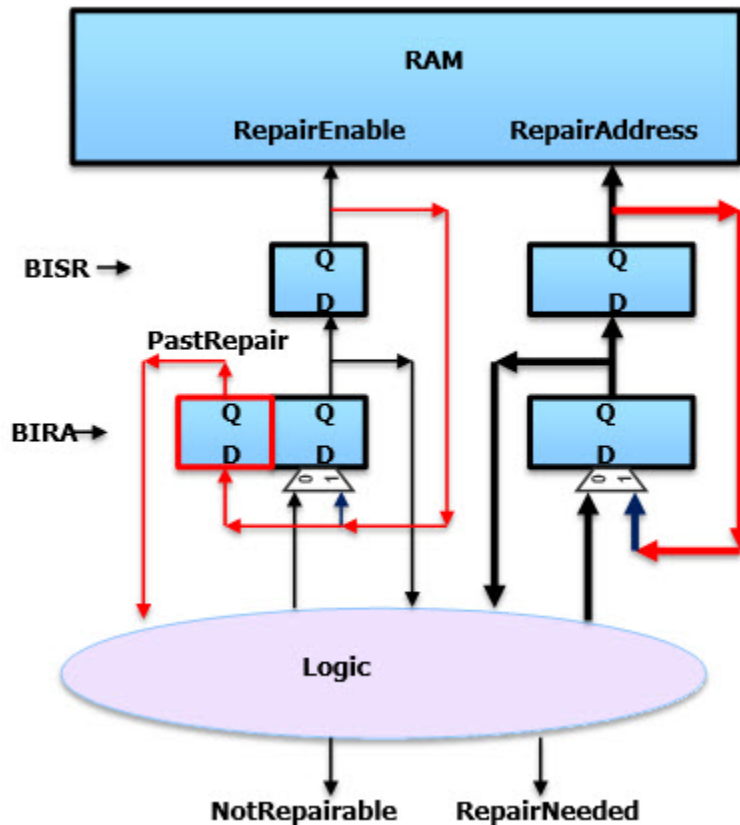
The mux at the input of the BIRA registers holding the repair enable and repair address enables loading the initial repair enable and repair address from the corresponding BISR register instead of clearing the registers. The BISR-to-BIRA transfer occurs by default. If necessary, you can disable the transfer by specifying the preserve_fuse_register_values in the PatternsSpecification/Patterns/TestStep/MemoryBist/DiagnosisOptions wrapper.

The PastRepair register is used to check whether an error that appears to be covered by a spare resource is actually in the spare itself. If this is the case, the memory must be declared as unrepairable. Otherwise, all errors appear to be covered and not need new repairs, and the MemoryBIST post-repair pattern is not performed. Faulty spares would escape under these conditions. When multiple spares can cover the same address segment (row or column), replacing the bad spare with a known good one might be possible, but spare replacement is not supported by Tessent Shell MemoryBIST. The circuit shown on Figure 5-54 is applicable to memories having a repair enable input. A slightly different circuit is used for memories that do not have such an input. Tessent Shell MemoryBIST automatically selects and generates the appropriate circuit for each memory with spare resources.

One consequence of the BISR-to-BIRA transfer capability is the additional number of wires between the two registers. The two registers should stay as close as possible to minimize wire congestion.

BIRA Repair Status Bits Checking

In the normal manufacturing flow, the BISR chain is rotated, and TDO is compared to 0 to detect whether the chip needs repair with a simple pattern that does not require extracting the repair status of individual memories. However, this mechanism does not work when using incremental repair because the BISR chain may contain 1's because of a repair in previous test insertions.

Figure 5-55 shows the implementation of the mechanism used to observe both REPAIR_STATUS bits of each memory through the controller GO output connected to the BAP. This solution only requires one additional global signal to select between the two status bits and eliminates the need to scan the controller setup chains to determine the "Non-Repairable" condition and the "Repair Needed" condition. This mechanism also can be used in the normal manufacturing flow to optimize test time.

Figure 5-55. Detection of Repair Needed Condition
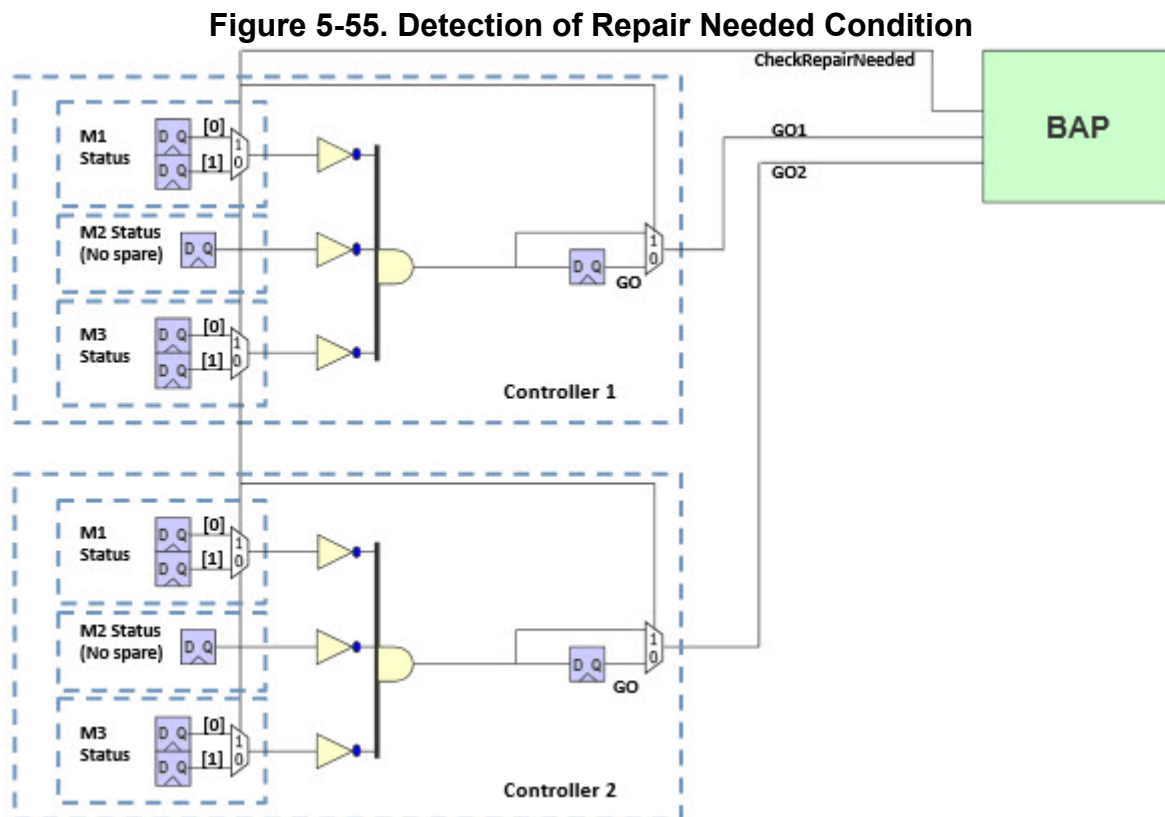
Figure 5-55 shows the hardware enabling this mechanism, including the control signal CheckRepairNeeded. The example shows two controllers. Each controller has two memories with spare resources and one memory without. All memories have a repair status register, although the memory without spare resources only has the equivalent of REPAIR_STATUS[1], which indicates a non-repairable status as shown in Table 5-7. This register is set as soon as an error is found for this memory that forces the GO output of the controller to go low. For memories with spare resources, REPAIR_STATUS[1] is set if the errors found cause the memory to be non-repairable. In both cases, the chip is bad. Note that the path from

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

REPAIR_STATUS[0] to the controller output is combinational as the GO register may not be clocked at the time the check is performed.

Table 5-7. REPAIR_STATUS Register Decodes

Bit1 Bit0	Repair Status
00	No Repair Required
01	Repair Required
1x	Not Repairable

The interpretation of the GO output throughout the BISR manufacturing flow repair stages shown in Figure 5-38, are summarized in Table 5-8 below.

Table 5-8. GO Output Interpretation by Repair Stage

Results of compare on Go (compare_go: on)	TP1 Pre-Repair Patterns (check_repair_status: on | non_repairable)	TP2 Repair Patterns (check_repair_needed: on)	TP3 Post-Repair Patterns (check_repair_status: off)
Pass	All memories are fault-free or repairable	All memories are fault-free	All memories are repaired successfully
Fail	At least one memory is non-repairable	At least one memory needs repair	At least one memory has a bad spare

Table 5-9 shows the actions performed for all combinations of compare_go, compare_go_id, compare_memory_go, check_repair_status, and extract_repair_fuse_map. The most efficient combination of options is: compare_go: On, compare_go_id: Off, check_repair_status: non_repairable. In this case, the setup chain of the controller is not accessed (Action A2 in Table 5-10) to inspect the repair status registers of all memories. This is not necessary because comparing the GO signal to 1 is sufficient to determine whether the memory is repairable.

Some combinations involving Action A3 might be problematic and are identified with a "*" in the table. Warnings are reported when scanning out GO_ID registers for which the feedback path has been disabled. This is the case for the GO_ID registers part of the shared or local comparators that are used to test memories with spare resources. The value contained in those registers is not meaningful and always matches the expected value "0".

Table 5-9. Action Table for GO_ID

compare_go	compare_memory_go	compare_go_id	extract_repair_fuse_map	check_repair_status		
				Off	On	non_repairable
Off	Off	Off	Off	A1	$A2_{00}$	$A2_{0X}$
Off	Off	On	Off	A3	$A3^* + A2_{00}$	$A3^* + A2_{0X}$
Off	On	Off	Off	$A2_{0X}$	$A2_{00}$	$A2_{0X}$
Off	On	On	Off	$A3+A2_{0X}$	$A3^*+A2_{00}$	$A3^*+A2_{0X}$
On	Off	Off	Off	A4 [1,5]	$A4 + A2_{00}$	A4 [1,4]
On	Off	On	Off	A4+A3 [3,5]	$A4+A3^*+A2_{00}$	$A4+A3^*+A2_{0X}$
On	On	Off	Off	$A4+A2_{0X}$ [2,5]	$A4+A2_{00}$	$A4+A2_{0X}$ [2,4]
On	On	On	Off	$A4+A3+A2_{0X}$ [2,3,5]	$A4+A3^*+A2_{00}$	$A4+A3^*+A2_{0X}$
Off	X	Off	On	A5	A5	A5
Off	X	On	On	$A3^*+A5$	$A3^*+A5$	$A3^*+A5$
On	X	Off	On	A4+A5	A4+A5	A4+A5
On	X	On	On	$A4+A3^*+A5$	$A4+A3^*+A5$	$A4+A3^*+A5$

Notes:

1. No Diagnosis

2. Memory level diagnosis resolution

3. IO level diagnosis (local comparators only)

4. **Yellow highlight table cells: recommended settings for MBIST pre-repair**

5. **Green highlight table cells: recommended settings for MBIST post-repair**

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Table 5-10. Action Descriptions

Action	Description
A1	Rule check: At least one comparison must be enabled.
A2$_{<S1S0>}$	Scan-out SHORT_SETUP; Compare REPAIR_STATUS[1:0] against <S1S0> Memories without spares compare their REPAIR_STATUS against 0 (<S1>).
A3 * —	Scan-out SHORT_SETUP; Compare GO_ID_REG[x:0]. Warnings: GO_ID_REGs are not "sticky" (that is, they do not accumulate test results) for some configurations and mode of operation. Only the results of the last compare are available. Refer to tables below for more information.
A4	Compare GO/DONE on BAP Status register.
A5	Scan-out BIRA_SETUP; Compare REPAIR_STATUS[1:0] and fuse map (repairable memories) AND REPAIR_STATUS[1:1] (non-repairable memories).

GO_ID Register Behavior

When the MemoryBist controller is performing a Go/NoGo test, the content of the GO_ID register accumulates the content of the comparators on each strobe. Upon completion of the algorithm, the content of the GO_ID register contains a '1' for each comparator that detected a failure at any time during the algorithm. This cumulative behavior of the GO_ID register is referred to as "sticky".

When the redundancy analysis (BIRA) mode is enabled (setting RepairOptions/ check_repair_status to "on" or "non_repairable"enables BIRA mode), the behavior of the GO_ID registers depends on the type of repair used and the location of the comparators, as shown in Table 5-11. The GO_ID registers are sticky during BIRA execution for memories using local comparators and IO repair, as well as for non-repairable memories using local comparators. For all other memories, the content of the GO_ID registers only reflect the result of the last compare of a controller run. When using shared comparators, this corresponds to the last strobe value from the last run BIST controller step. For local comparators with a repairable memory, this corresponds to the last strobe value for the current memory.

The tool issues a warning is when check_repair_status is enabled ('on' or 'non_repairable') and the MemoryBist controller has memories with non-sticky GO_ID registers. The state of the sticky GO_ID registers can be collected in the same controller run used for MBIST pre-repair. However, the disadvantage is that the simple flow shown in Figure 5-38 can't be implemented. A full controller setup is required and the tester must interpret the meaning of the data scanned out instead of using a simple pass/fail criterion for deciding the next step.

The tables below illustrate the behavior of the GO_ID registers in various modes.

Table 5-11. GO_ID Behavior in BIRA Mode

Comparator Type	Memory With Redundancy		Memory Without Redundancy
	IO Repair	Any Other Repair Method	
Shared	Non-Sticky	Non-Sticky	Non-Sticky
Local	Sticky	Non-Sticky	Sticky

Table 5-12. GO_ID Behavior in Stop-On-Error Mode

Comparator Type	Memory With Redundancy	Memory Without Redundancy
Shared	Non-Sticky	Non-Sticky
Local	Non-Sticky	Non-Sticky

Table 5-13. GO_ID Behavior in Go/NoGo Mode

Comparator Type	Memory With Redundancy	Memory Without Redundancy
Shared	Sticky	Sticky
Local	Sticky	Sticky

Table 5-14. Legend

Term	Description
Sticky	The content of the GO_ID register contains the accumulated values from the comparators from all strobes during the MemoryBIST execution. The GO_ID registers that contain 1's indicate that failures were detected on the corresponding comparators at any time during the algorithm execution.
Non-Sticky	The content of the GO_ID register contains the comparator values from the last strobe only. The GO_ID registers that contain 1's indicate the exact memory output that failed during the last strobe. This information is used for memory diagnosis (ESOE) as well as redundancy analysis (BIRA).

Examples

The following is an example configuration file section that corresponds to the flow shown in Figure 5-53 for the Pre-Repair pattern *TP1*:

```
PatternsSpecification {
  Patterns(PreRepairFlow){
    TestStep (0) {
      ...
    }
    TestStep (MbistPreRepair) {
      MemoryBist {
        Controller (<MemoryBist ICL Instance Name>) {
          RepairOptions {
            check_repair_status: non_repairable;
          }
          DiagnosisOptions {
            compare_go: on;
            compare_go_id: off;
          }
        }
      }
    }
    TestStep (1) {
      ...
    }
  }
}
```

The "Repair needed" condition is detected using a dedicated pattern, *TP2*, that consists of sampling the GO signal of all controllers. All WTAPs and TAPs are scanned twice, once to set the CheckRepairNeeded signal and once to sample the GO signals. A mis-compare indicates that the chip requires repair; otherwise, the chip is good. In all cases, the decisions are made

using a simple pass/fail criterion. The "Repair needed" condition is checked using the following wrapper properties:

```
PatternsSpecification {
  Patterns(RepairFlow){
    TestStep (0) {
      ...
    }
    TestStep (CheckRepairNeeded) {
      AdvancedOptions {
        split_patterns_file : on;
      }
      MemoryBist {
        run_mode : check_repair_needed;

      // Only controllers inside this TestStep have their
      // repair status bit checked. All others are ignored.
        Controller(<MemoryBist ICL Instance Name>) {// Repeatable
        }
      // No other properties can be specified
      }
    }
    TestStep (1) {
      ...
    }
  }
}
```

Absence of Fuse Programming Step

Once it is determined that repair is needed, only the BIRA-to-BISR transfer is performed before retesting the chips when using soft incremental repair. No fuse programming steps are performed.

Post-repair testing is necessary because newly allocated spares have not yet been tested. Also, the spares allocated in a soft incremental repair run have not been tested under all operating conditions and might fail because of small voltage and temperature variations over time.

Handling of Blocks Without Incremental Repair Capability

Blocks that do not have the incremental repair capability must be handled differently when using the incremental repair flow.

The differences are as follows:

- Blocks without incremental repair capability must be in a different Power Domain Group (PDG). PDG's can actually be sub-divisions of the same power domain. The objective is to have separate control of the BISR clock used for blocks with and without incremental repair capability.

- Blocks without incremental repair capability must run in Go/NoGo mode in the TP1 pattern. They are treated as memories without spare resources. If a failure occurs during this execution of test pattern TP1, the chip is declared non-repairable.

- Blocks without incremental repair capability cannot perform the BIRA-to-BISR transfer during the execution of the TP2.2 pattern. Otherwise, the repair solution for these blocks is lost. This means that the BIRA-to-BISR transfer of the blocks that do support incremental repair must be performed one PDG at a time.

Considerations Specific to Hard Incremental Repair

Hard incremental repair is an extension of soft incremental repair.

The differences are as follows:

- Hardware — The hardware necessary to support hard incremental repair is enabled by specifying the max_fuse_box_programming_sessions property in the DftSpecification/ MemoryBisr/Controller wrapper with "unlimited", or a value of 1 (the default value) or greater. Only the hardware of the top-level BISR controller is affected. The implementation of BISR chains is unchanged. This means that both soft and hard incremental repair can be used on existing blocks.

- Fuse box size — The number of fuses required for memory repair increases when using hard incremental repair. Refer to the description in the Incremental Repair Overview section for details on determining impacts on fuse box size.

- Fuse box organization — The way the repair data is stored in the fuse box is different, but the basic compression algorithm is unchanged when an integer value is specified for max_fuse_box_programming_sessions. Compression is disabled for hard incremental repair when "unlimited" is specified for this property. Refer to the description in the Incremental Repair Overview section for additional details.

- Repair flow — The repair flow is essentially the same as the flow without hard incremental repair. The main difference is that the flow can be repeated up to the number of times allowed with the max_fuse_box_programming_sessions property during manufacturing or in the system. Two examples are shown below. The first one is for the case where an integer is specified for max_fuse_box_programming_sessions and the second one is where the unlimited value is specified.

    In the example shown in Figure 5-56, one clock group is tested under two different sets of conditions using patterns *TP1.1 (LVcc)* and *TP1.2 (HVcc)*. During execution of *TP1.1*, the BIRA circuit is initialized with the values loaded in the BISR chain by pattern *TP0*; the PatternsSpecification/Patterns/TestStep/MemoryBist/DiagnosisOptions/ preserve_fuse_register_values configuration property is set to Off to calculate a new repair solution, taking into consideration the repair solution already programmed in the fuse box. However, preserve_fuse_register_values is set to On when executing *TP1.2* to accumulate the BIRA results. For both *TP1.1* and *TP1.2*, the Go output of all Tessent

MemoryBIST controllers is checked to determine if any non-repairable memory exists. Once all memories have been tested, pattern *TP1.3* is run to determine if any memory needs repair. Again, the Go output of all Tessent MemoryBIST controllers is checked to determine whether the fuse programming should be performed using pattern *TP2.2*. For the fuse programming, verification, and post-repair steps, the rest of the flow is identical to the flow described in "Top-Level Verification and Pattern Generation". The assumption is that the chip is not powered down at any point during the entire process.

Figure 5-56. Hard Incremental Repair Manufacturing Flow Example for max_fuse_box_programming_sessions: <integer>
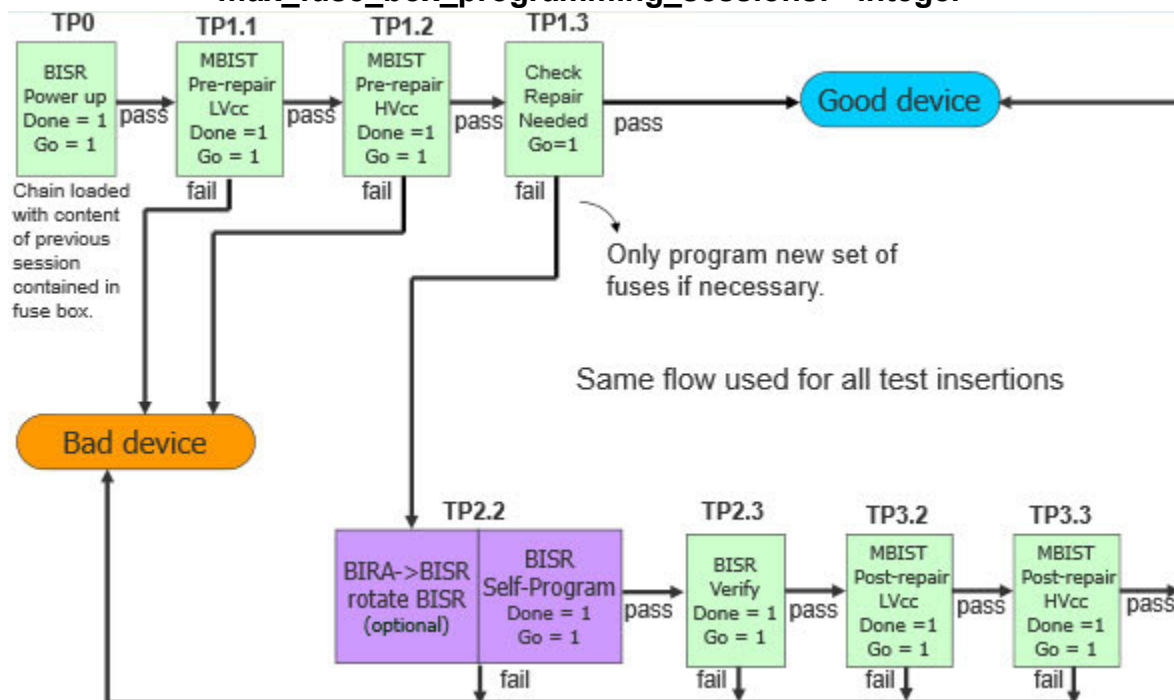
In the example shown in Figure 5-57, an additional step might be required to reset some bits of the BISR chains corresponding to the initial repair solution before proceeding with fuse programming. This depends on the type of nonvolatile memory used. Most eFuse macros do not tolerate reprogramming the same fuses, which is indicated in the eFuse datasheet. If reprogramming is not allowed, then the Verify step of TP2.2 is mandatory to avoid reprogramming fuses. The Verify autonomous mode of the BISR controller works differently when max_fuse_box_programming_sessions is set to unlimited, in that compression is disabled. The Verify step performs an XOR, determining the difference between the contents of the BISR chains and the decompressed values stored in the fuse box. The result is only the bits that were not previously programmed in the eFuse are loaded back into the BISR chains. This prevents the double programming of bits in the eFuse and avoids potential fuse box corruption. This step is expected to fail when performed before programming but must pass after programming.

BISR chains contain 0s at the end of TP2.4 if programming is successful. When a BISR controller supports a limited number of repair sessions, the verify step of TP2.3 loads the BISR chains with the correct repair data, decompressed from the fuse box. However when

max_repair_sessions is set to unlimited in the DftSpecification, the behavior of the verify step of TP2.4 is different. In this case, TP2.4 is broken into two TestSteps that perform the following operations:

- BIRA capture with BISR rotation — The BISR chains are loaded with the cumulative repair information from the BIRA modules.

- VerifyFuseBox — The cumulative repair data from the fuse box (programmed in TP2.3) is XOR'd with the BISR chain content. If the fuse box matches the BISR chain content, the resulting BISR chain content will be 0 at the end of the operation. It is therefore necessary to perform a PowerUp step before running post-repair MBIST.

_____ **Note** _____
Any 1s that remain in the BISR chain after the VerifyFuseBox TestStep of TP2.4 indicate the corresponding BISR bit failed to program in the fuse box.

Figure 5-57. Hard Incremental Repair Manufacturing Flow Example for max_fuse_box_programming_sessions: unlimited
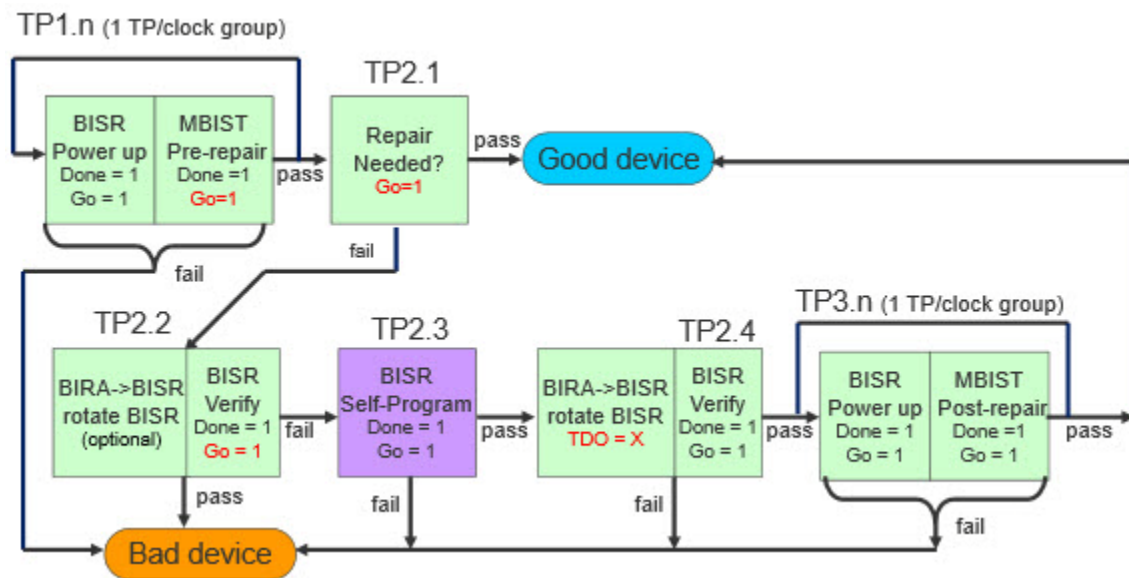
Repair Sharing

Repair sharing provides the designer with the capability to reduce the area and power-up time increase realized with memory repair implementations, while maintaining the capability of performing memory repair. The following section outlines the requirements and steps to incorporate the sharing of BIRA and BISR hardware among multiple memories to realize these benefits.

Repair Sharing Overview

Implementing Built-in Self Repair (BISR) in designs that contain a large number of memories can have a significant impact on area and the time required for power up. Tessent Shell MemoryBIST has the capability to share BISR and BIRA hardware among memories and reduce these design impacts while maintaining balance with any potential yield impacts.

Repair sharing can be implemented on memories with Row/Word-only, Column/IO-only and Row/Column repair types. Memories of different dimensions are allowed to share the same BISR/BIRA hardware. The designer also has control over the level of sharing, or grouping of memories on a BISR/BIRA circuit to maintain a proper balance between potential yield impact and improvements in area and power-up time. Yield may be impacted because a failure detected in one memory allocates the same spare element on all memory instances sharing the BISR module, and a group might become unrepairable if another failure is found in the same group.

To share the repair logic for multiple memory instances, the BIRA module must be generic to all memory instances inside the repair group. A repair group represents a number of segments that can be part of a single or multiple memories. Tessent MemoryBIST analyzes the RedundancyAnalysis wrapper for all memories inside a repair group. The ColumnSegment and RowSegment wrappers from the largest segments are used to create the BIRA module, and these can come from two different memory templates. The union of these wrappers results in a BIRA module that can accommodate all memory instances inside the repair group.

Repair Sharing Conditions

The conditions described in this section must be met to share repair logic among memories in a repair group. Tessent MemoryBIST performs rule checks to ensure the compatibility of all memories that are sharing the same BIRA and BISR modules.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

The following conditions are to be met for memories to share repair logic:

- **Memories must use the same repair type**

  Allowed repair types are Row/word-only, Column/IO-only and Row/Column.

- **Only one spare element per repair segment is utilized in memories**

  When repair sharing is enabled on memories with multiple spare elements per RowSegment or ColumnSegment wrapper, only the first spare element is connected to the shared BISR register. The additional spare elements are disabled and remain unused. For this situation, the following warning displays once for each impacted memory instance:

  ```
  //Warning:/DftSpecification(BLK,rtl)/MemoryBist/Controller(c1)/Step<0>/
  MemoryInterface(m1)
  // Read from file: BLK_rtl.dft_specification, line number: 26
  // This memory interface is assigned to repair group 'RGA'.
  // The number of spare elements defined inside the TCD description for this
  // memory is larger than 1. A maximum of 1 spare element will be implemented
  // per RowSegment and ColumnSegment wrapper when a memory interface is
  // assigned to a repair group. All other spare elements will be tied off.
  ```

  The warning shown above can be waived by setting the following environment variable:

  ```
  SETUP> setenv TESSENT_WAIVE_RS_SPARE_ELEMENT_WARNING 1
  SETUP> process_dft_specification
  ```

- **Memories must use the same spare size**

  a. Spare row block: The LSB's of the Fuse register must log the same row address bits.

  b. Spare column block: The LSB's of the Fuse register must log the same column address bits.

  c. Spare IO: Multiple IOs can use the same repair code.

- **Repair group must meet user specifications**

  a. Maximum group size: The total number of memory bits that can share a repair solution.

  b. Repair group scope: Repair sharing should be limited to segments within a physical memory, or to memories within a logical memory for Shared Bus applications.

- **Memories must have parallel memory repair interfaces**

  a. The BISR registers are shared by memories that may require a different number and arrangement of repair bits. The BISR registers contain allocation, fuse map and fuse address bits. The BISR register connections for memories with parallel repair interfaces are made directly to the corresponding memory repair bits. However, memories with serial repair interfaces would require a specific shifting order of these bits such that the shift chain in the BISR register matches the repair register internal

to each, possibly unique, memory. Because it is not possible to re-arrange the BISR registers to match the needed serial chain, memories with serial repair interfaces are not supported.

- **Memories must have compatible physical address mappings**

  The address ports specified as fuses by different memories their respective RedundancyAnalysis/RowSegment/FuseSet wrappers must use the same equations in their physical address maps. Additionally, the terms and operators must be in the same order.

  An example of an incompatible physical address mapping is shown in Figure 5-58 below. Two memories, Mem1 and Mem2 specify RowAddress[2] as Fuse[1] in their respective FuseSet wrapper. The mapping equation for this address bit is shown in their PhysicalAddressMap wrapper. The equations are equivalent, but the terms r[2] and r[0] are not in the same order. These memories would be placed in different repair groups unless the equations are modified.

Figure 5-58. Incompatible Address Mapping Example

```
//Example showing incompatible address mapping
//Two memories with spare block of 2 rows
//Mem1: 16 rows/4 columns, Mem2: 8 rows/2 columns

RedundancyAnalysis { //Mem1: 16 rows/4 columns
  RowSegment(ALL) {
    FuseSet {
            Fuse[2]: AddressPort(ADR[5]); //RowAddress[3]
            Fuse[1]: AddressPort(ADR[4]); //RowAddress[2]
            Fuse[0]: AddressPort(ADR[3]); //RowAddress[1]
    }}}
  PhysicalAddressMap { //Mem1: default mappings not shown
    RowAddress[2]:r[0] xor r[2];
  }
.
.
.
RedundancyAnalysis { //Mem2: 8 rows/2 columns
  RowSegment(ALL) {
    FuseSet {
            Fuse[1]: AddressPort(ADR[3]); //RowAddress[2]
            Fuse[0]: AddressPort(ADR[2]); //RowAddress[1]
    }}}
  PhysicalAddressMap { //Mem1: default mappings not shown
    RowAddress[2]:r[2] xor r[0];
  }
```

- **Memories must have compatible segment sizes**

  This compatibility condition is related to row or column segments for which the size is not a power of two. When sharing with other segments of larger sizes, an out-of-range repair address might be generated, and some memories may not tolerate this situation.

By default, it is assumed that memories do not allow out-of-range repair addresses, and more repair groups might be required in this case to accommodate these memories. Memories that do not tolerate their spare elements in out-of-range repair addresses should either be prevented from using repair sharing or be assigned to a repair group with identical memories. The procedure for assigning these memories to a separate repair group is as follows:

a. Disable repair sharing for incompatible memories

```
set_memory_instance_option memory_instances -repair_sharing off
```

b. Create the DftSpecification

```
create_dft_specification
```

c. Edit the DftSpecification to manually assign the memories you want from step(a) to a repair group. The repair group is specified with the controller Step/ MemoryInterface/repair_group_name property.

> **Note**
> Ensure that only compatible memories are assigned to a repair group. No validation is performed to confirm if the memories are compatible.

The following example illustrates the out-of-range repair address situation. Three memories, using row repair as outlined in Figure 5-59, have 16, 12 and 8 rows respectively. If a defect is found in row 12 to 15 of Mem1, Mem2 receives an out-of-range repair address if it is in the same repair group as Mem1. Mem3 receives a repair address that appears to be in-range and is always compatible with both Mem1 and Mem2. If Mem2 tolerates an out-of-range repair address, all three memories can be part of the same repair group. If not, two groups are required. Mem2 can be part of its own group or it can be grouped with Mem3. The solution selected depends on the memory sizes, so that the repair groups are balanced in terms of their total number of bits.

Figure 5-59. Incompatible Segment Size Example

```
//Three memories with a spare block of 2 rows
//Mem1: 16 rows/4 columns, Mem2: 8 rows/2 columns

RedundancyAnalysis { //Mem1: 16 rows/4 columns
  RowSegment(ALL) {
    FuseSet {
            Fuse[2]: AddressPort(ADR[5]); //RowAddress[3]
            Fuse[1]: AddressPort(ADR[4]); //RowAddress[2]
            Fuse[0]: AddressPort(ADR[3]); //RowAddress[1]
    }}}
.
.
.
RedundancyAnalysis { //Mem2: 12 rows/4 columns
  RowSegment(ALL) {
    FuseSet {
            Fuse[2]: AddressPort(ADR[5]); //RowAddress[3]
            Fuse[1]: AddressPort(ADR[4]); //RowAddress[2]
            Fuse[0]: AddressPort(ADR[3]); //RowAddress[1]
    }}}
.
.
.
RedundancyAnalysis { //Mem3: 8 rows/4 columns
  RowSegment(ALL) {
    FuseSet {
            Fuse[1]: AddressPort(ADR[4]); //RowAddress[2]
            Fuse[0]: AddressPort(ADR[3]); //RowAddress[1]
    }}}
```

Implementing Repair Sharing

Repair sharing is disabled by default for backward compatibility. Once a design is elaborated, repair sharing can be enabled for the memory instances you want by following the methods outlined in this section.

Creating and Modifying Repair Share Groups

Once a design has been elaborated by executing the set_current_design command, the memories that are to have repair sharing need to be enabled prior to the generation of the DftSpecification. This operation is performed while in the DFT context and the setup system mode. After the DftSpecification is created, the groupings can be further modified.

The command set_memory_instance_options is used to set the memory instance option -repair_sharing, which is used to control repair sharing for the specified memory instances.

```
set_memory_instance_options memory_inst -repair_sharing off | on | (auto)
```

The *memory_inst* argument enables wildcards and can be a list or collection of one or more memory instances. For Shared Bus applications, the instance name can be the name of a memory cluster.

The -repair_sharing option defaults to "auto", which defers repair sharing control to the global DefaultsSpecification RepairOptions/repair_sharing property, as shown in Figure 5-60. Because this has the default setting of "off", repair sharing is disabled by default for all memories. Setting the instance option -repair_sharing to "on" or "off" overrides the DefaultsSpecification RepairOptions repair_sharing setting for the specified memories and memory clusters.

Figure 5-60. DefaultsSpecification Repair Sharing Properties

```
DefaultsSpecification {
  DftSpecification {
    MemoryBist {
      RepairOptions {
        max_repair_group_size :(unlimited) | {int[(kilobits) | megabits]};
         repair_sharing        : on | (off);
      }
      MemoryInterfaceOptions {
        repair_group_scope    : physical_memory | (controller);
      }
      MemoryClusterOptions {
        repair_group_scope    : (controller) | logical_memory
                                | physical_memory ;
      }
    }
  }
}
```

The example dofile shown in Figure 5-61 shows how repair sharing can be enabled on all memory instances in a design. Note that this can also be achieved by setting the DefaultsSpecification RepairOptions/repair_sharing property to "on", which enables repair sharing for all memories by default.

Figure 5-61. Repair Sharing on All Memory Instances Example

```
set_context dft -rtl
read_cell_library my_library.lib
read_verilog my_chip.vb
set_current_design my_chip

#enable memory sharing on all memory instances
set_memory_instance_options [get_memory_instances] -repair_sharing on

set_design_level chip
set_dft_specification_requirements -memory_test on
add clocks clka 12ns

check_design_rules
set my_spec [create_dft_specification]
report_config_data $my_spec

process_dft_specification
```

The property settings in the DefaultsSpecification, along with those specified with set_memory_instance_options, are used by the create_dft_specification command to construct repair groups for non shared-bus memories and generate the DftSpecification. Repair groups for Shared Bus memories cannot be resolved during create_dft_specification because the expansion level (logical or physical) is not known. For memory clusters, the settings in the DefaultsSpecification RepairOptions and MemoryClusterOptions wrappers are copied into the DftSpecification MemoryCluster wrapper for each cluster. If repair sharing has been enabled

for the memory cluster, either through the default setting or with set_memory_instance_options, then the repair_sharing property in the MemoryCluster wrapper is set to "on". The repair groups for memory clusters are then generated during process_dft_specification.

After create_dft_specification is run, a DftSpecification similar to the one shown in Figure 5-62 is created that shows repair group assignments for non shared-bus memories, as well as MemoryCluster wrappers for shared-bus memories. The DftSpecification can be modified by the user to change repair group assignments for non shared-bus memories. Modifications can also be made on memory clusters to enable or disable repair sharing, change repair_group_scope settings or change max_repair_group_size settings. After the wanted changes are made, process_dft_specification can be run.

Figure 5-62. Example DftSpecification Properties for Repair Sharing

```
DftSpecification(top,rtl) {
  MemoryBisr {
  }
  MemoryBist {
    ijtag_host_interface : Sib(mbist);
    Controller(c1) {
      clock_domain_label : clka;
      Step {
        comparator_location : shared_in_controller;
        MemoryInterface(m1) {
          instance_name : core_inst1/blockA_clka_i1/mem1;
          repair_group_name : (none) | <repair group name>;
        }
        ReusedMemoryInterface(m2) {
          reused_interface_id : m1;
          instance_name : core_inst1/blockA_clka_i2/mem1;
          repair_group_name : (none) | <repair group name>;
        }
      }
      Step {
        comparator_location : shared_in_controller;
        MemoryInterface(m2) {
          instance_name : core_inst1/blockA_clka_i1/mem4;
          repair_group_name : bira_g1;
        }
        MemoryInterface(m3) {
          instance_name : core_inst1/blockA_clka_i1/mem5;
          repair_group_name : bira_g2;
        }
        MemoryInterface(m4) {
          instance_name : core_inst2/blockA_clka_i1/mem5;
          repair_group_name : bira_g1;
        }
      }
    }
  }
  Controller(c2) {
    clock_domain_label : clka;
    MemoryCluster(c1) {
      instance_name : cluster_instance;
      repair_sharing : (off) | on;
      max_repair_group_size : (unlimited)
                              | int [(kilobits)|megabits];
      repair_group_scope : (controller) | logical_memory
                              | physical_memory;
    }
  }
}
```

BISR Segment Order File

There are no modifications to the structure or content of the BISR segment order file when using repair sharing. The paths to the physical memories described in the BISR segment order file are used to determine the final order of the BISR chain when repair sharing is enabled.

When repair groups are used, the first memory that is part of a repair group defines the location of the shared BISR register in the chain. Subsequent memories that belong to that repair group do not affect the final BISR chain order.

Figure 5-63 below shows the order of the BISR registers as described in the BISR segment order file on the left hand side and the final order when repair sharing is enabled on the right hand side. The BISR register of repair group 1 is first in the final chain because M1 the first memory of that repair group and its position in the original chain order is before M3, which is the first memory of repair group 2.

Figure 5-63. BISR Chain Ordering With Repair Sharing
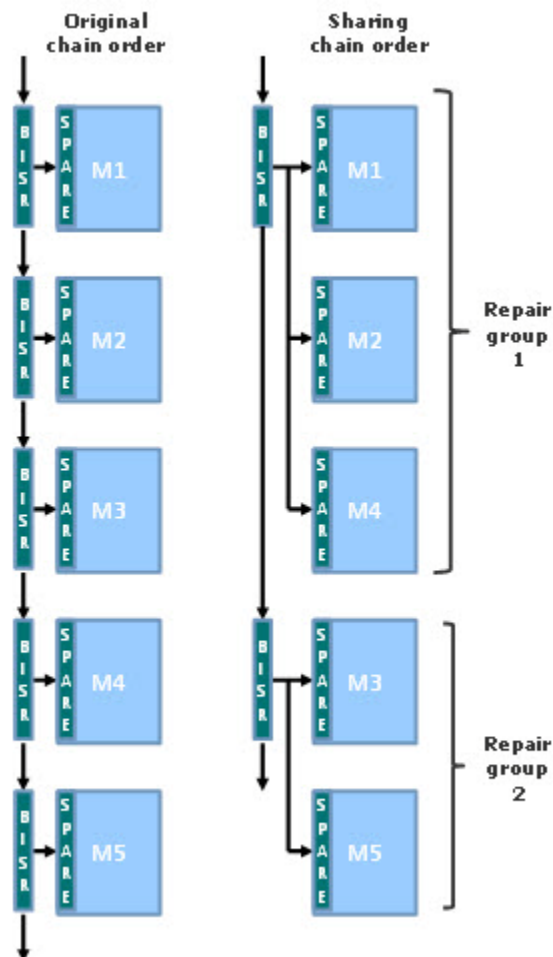
The process_dft_specification command creates an annotated BISR segment order file with a
".annotated" suffix in the current directory. The annotated BISR segment order file for the
circuit on the left in the figure above is shown below:

```
//-------------------------------------------------------
//   File created by: Tessent Shell
//           Version:
//         Created on:
//-------------------------------------------------------

BisrSegmentOrderSpecification {
  PowerDomainGroup(-) {
    // bisr_si --> {OrderedElement ...} --> bisr_so
    OrderedElements {
      mem_m1; // RepairGroup:None BISRLength:18
      mem_m2; // RepairGroup:None BISRLength:18
      mem_m3; // RepairGroup:None BISRLength:18
      mem_m4; // RepairGroup:None BISRLength:18
      mem_m5; // RepairGroup:None BISRLength:18
    }
  }
}
```

The comment after each OrderedElements entry of "RepairGroup:None" confirms that repair
sharing is not enabled, and that each memory has its own BISR register 18 bits in length. To
contrast, the annotated BISR segment order file for the circuit on the right in Figure 5-63 is
shown below:

```
//-------------------------------------------------------
//   File created by: Tessent Shell
//           Version:
//         Created on:
//-------------------------------------------------------

BisrSegmentOrderSpecification {
  PowerDomainGroup(-) {
    // bisr_si --> {OrderedElement ...} --> bisr_so
    OrderedElements {
      mem_m1; // RepairGroup:c1_RG1 BISRLength:18
      mem_m2; // RepairGroup:c1_RG1
      mem_m4; // RepairGroup:c1_RG1
      mem_m3; // RepairGroup:c1_RG2 BISRLength:18
      mem_m5; // RepairGroup:c1_RG2
    }
  }
}
```

The "RepairGroup:c<n>_RG<m>" entry in the comment after each OrderedElements entry
confirms the repair sharing feature is enabled. The c<n> part of the string shows the controller
that tests the memory and is the same identifier (*id*) specified in the MemoryBist/Controller(*id*)
of the DftSpecification. In the example above, all memories are tested by the "c1" controller.
The RG<m> part of the string shows the repair group that the memory belongs to, which is
specified by the repair_group_name property of the DftSpecification.

When you use repair sharing, only one OrderedElements entry per repair group has a BISRLength comment, which shows the length of the BISR register associated with that repair group. The file lists all members of a repair group on consecutive lines in the OrderedElements wrapper.

If you want to manually reorder the BISR chain, Siemens recommends moving all elements of a repair group together. For example, if you want to reverse the BISR chain order shown in the repair sharing example above, you will change it as follows:

```
//-----------------------------------------------------
//  File created by: Tessent Shell
//         Version:
//       Created on:
//-----------------------------------------------------

BisrSegmentOrderSpecification {
  PowerDomainGroup(-) {
    // bisr_si --> {OrderedElement ...} --> bisr_so
    OrderedElements {
      mem_m3; // RepairGroup:c1_RG2 BISRLength:18
      mem_m5; // RepairGroup:c1_RG2
      mem_m1; // RepairGroup:c1_RG1 BISRLength:18
      mem_m2; // RepairGroup:c1_RG1
      mem_m4; // RepairGroup:c1_RG1
    }
  }
}
```

BISR Instance Location

The BISR registers are inserted at different hierarchical levels in the design depending on the composition of the repair groups. For all repair_group_scope settings, the common ancestor or parent module of all memory instances inside a repair group is used to determine the BISR register instance location.

In the example shown in Figure 5-64, all the memories use local comparators and have two spares that share a common BISR register. For this case, the BISR register location is simply the immediate parent of each memory and is the same as the case where there is no repair sharing. This is because for repair sharing, all memories must share comparators that are located in the controller.

Figure 5-64. BISR Register Location With Local Comparators
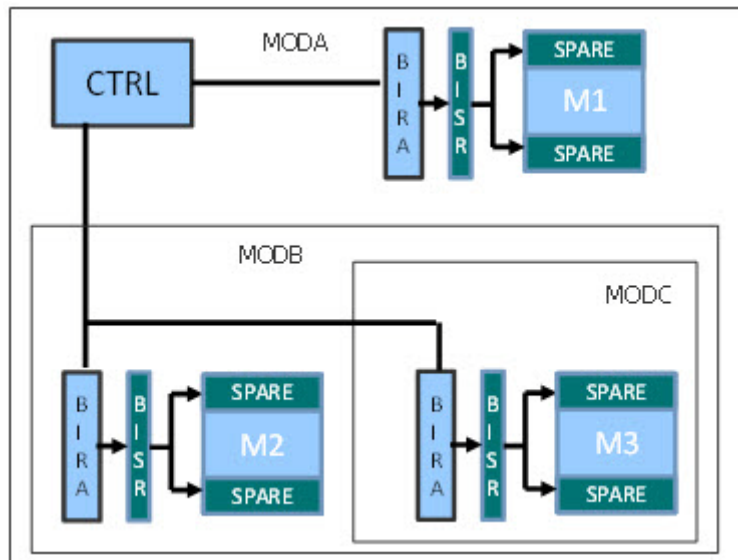
Note that the same BISR register location result would occur if the comparators and BIRA logic were moved to the controller, but repair groups were limited to physical memory boundaries by specifying repair_group_scope : physical_memory.

In the example shown in Figure 5-65, M2 and M3 share comparators that are located in the controller along with the BIRA logic. The shared BISR register is located in MODB, which is the common ancestor of the two memories. For this example, M1 has its own repair group or is excluded from repair sharing because of having local comparators.

Figure 5-65. BISR Register Location Example With Two Repair Groups
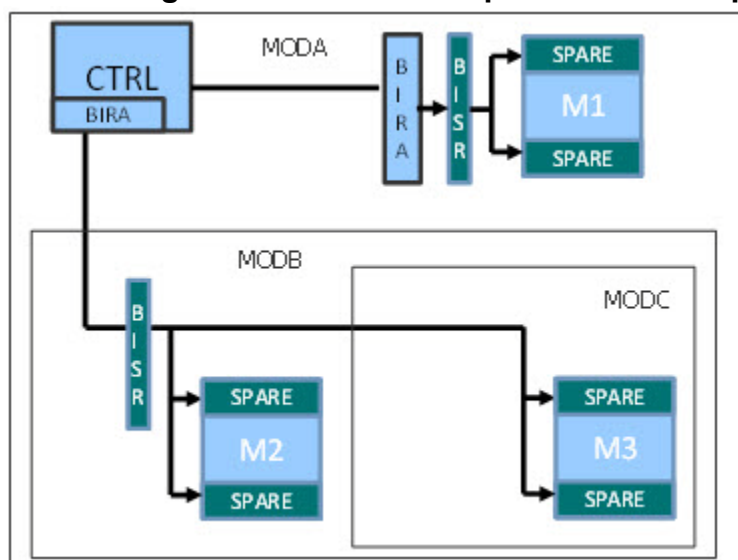
The example shown in Figure 5-66 shows the case where all memories have comparators located in the controller and are part of the same repair group. In this case, the BIRA logic is

located in the controller with the comparators and the shared BISR register for all three memories is located in the common ancestor module MODA.

Figure 5-66. BISR Register Location Example With One Repair Group
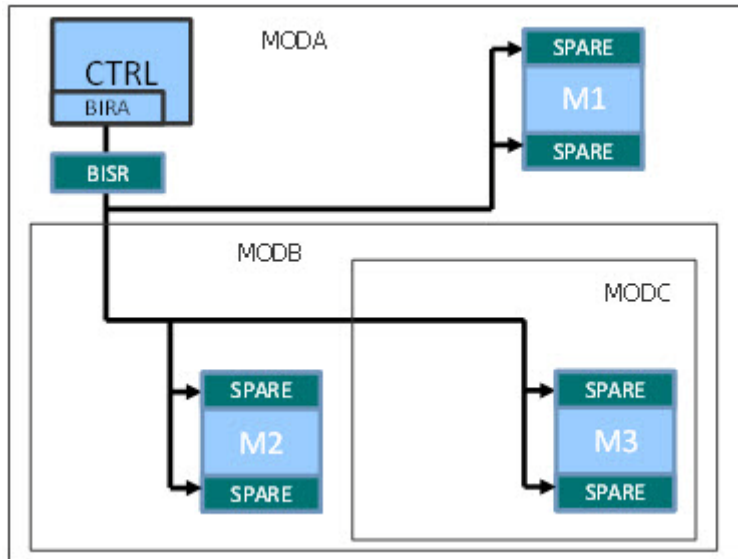
Fast BISR Loading

Fast BISR loading is a BISR DFT hardware modification that can be implemented to reduce the time needed for loading the BISR registers during power-up. The material presented in this section describes the architectural modifications created as well as how to implement fast BISR loading.

Fast BISR Loading Overview

Fast BISR loading is a BISR DFT hardware modification that can be implemented to reduce the time needed for loading the BISR registers during power-up. The repair data is not compressed for this implementation and is stored directly in the fuse box. The BISR hardware drives the repair information from the fuse box directly to the memory parallel repair ports, which enables a significant speed improvement when powering up a device or power region. You can only use repairable memories with parallel repair interfaces with fast BISR loading implementations. Repairable memories with serial repair interfaces are not allowed.

The fuse box must be large enough to store the repair information for the entire chip. It is not necessary to further increase the fuse box size if Incremental Repair is used. Because the entire fuse box content is driven from fuse box interface ports, a compatible interface defined by the eFuse macro must be used. Siemens EDA provides compatible fuse box interfaces for TSMC technologies. Others must be provided by the user.

Chips that incorporate large numbers of memories may require implementing Repair Sharing to reduce the number of unique memory repair inputs and reduce routing congestion.

Fast BISR Architecture

An overall view of the fast BISR loading architecture is shown in the figure below. In this overview, the existing serial BISR interfaces are shown in orange, with one BISR chain for the repairable memories within each of the three power domains. When fast BISR loading is implemented, each BISR register is directly connected to the read buffer on fuse box interface using a parallel bus. Each BISR register is connected to dedicated bits on the parallel repair bus, as indicated by the purple paths in the figure.

After implementation, both parallel and serial repair buses are in place, with the parallel bus used to quickly load the BISR registers during power-up and the serial interface used for Autonomous mode BISR chain rotation operations.

Figure 5-67. Overall Fast BISR Loading Architecture
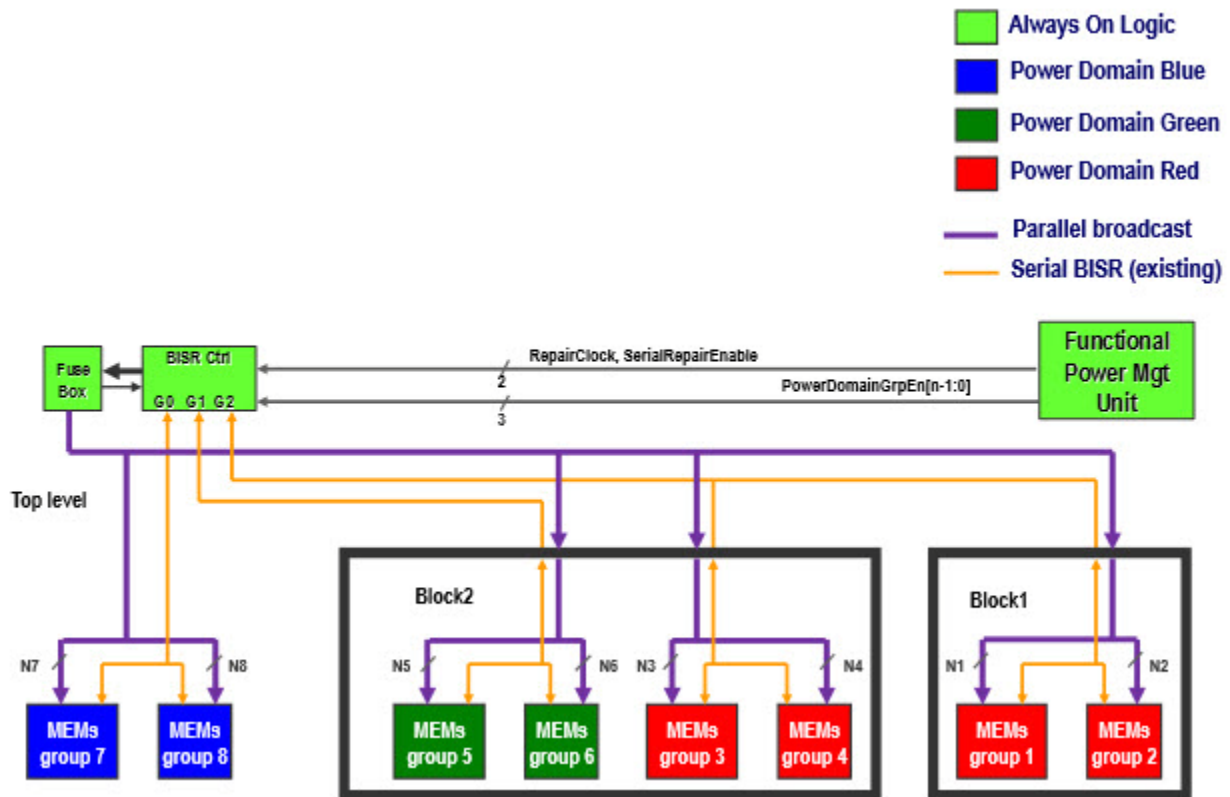
When fast BISR loading is enabled in the DftSpecification, multiplexers are added between the BISR register and the memory repair inputs as shown in Figure 5-68. These multiplexers enable the repair solution contained in the read buffer of the fuse box to be applied in parallel to the memory inputs. The parallel load access is enabled by setting the SerialRepairEnable input pin to "0". This input is reset to "0" when the TAP is reset and it is not necessary to control this input during the functional power up mode. This input is set to "1" for all other modes, unless explicitly set by the user during patterns generation.

Figure 5-68. Parallel Load Multiplexers
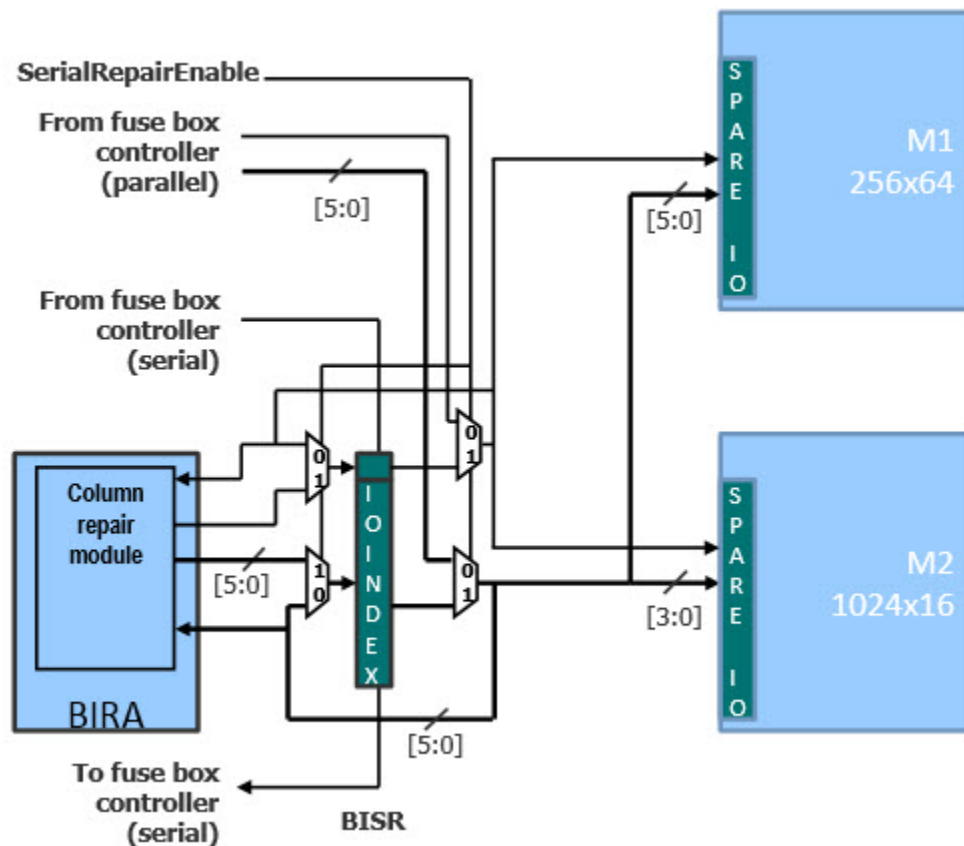
The second set of multiplexers, located between the BIRA module and the BISR register, enables repair data to be captured by the BISR register for verification purposes when SerialRepairEnable is set to "0".

During DFT Insertion, the SerialRepairEnable signal is connected to the BISR controller if it is present. If the BISR controller is not yet present in the design, this signal is connected to a port on the physical block.

The parallel repair inputs connecting through muxes to the memory repair inputs and BISR register inputs are connected to the read buffer port on the fuse box interface. The fuse box is not compressed when fast BISR loading is implemented and the memory repair information is stored directly, which requires a larger fuse box. Each parallel repair input bit is automatically connected to its corresponding fuse box bit.

Fast BISR Integration on Lower Level Blocks

When integrating *sub_block* or *physical_block* child designs, how BISR is implemented within each block must be considered to successfully implement fast BISR loading. The combinations listed in Table 5-15 identify both supported and unsupported cases, as well as options available to the user for design variations.

Table 5-15. Fast BISR Loading Support

Parent Block	Child Block	Action
Serial	Serial	This is the normal usage case. The BISR chains in both the parent and child blocks are loaded serially. If needed, any child BISR chains can be disabled by connecting the chain inputs to LogicLow and removing the chain SI pin from the BISR segment order file.
Parallel Serial	Serial	Unsupported case. The user has three options: • Support fast BISR by enabling SerialRepairEnable on the child block through setting MemoryBisr/ memory_repair_loading_method to "from_read_buffer" and re-processing DFT insertion. • Remove fast BISR entirely by disabling the SerialRepairEnable property for the parent block in the DftSpecification. This is done by setting MemoryBisr/memory_repair_loading_method to "serial". • Retain fast BISR in the parent block and disable memory repair in the child blocks by removing the child BISR chains from the BISR segment order file and connecting the SI pin to LogicLow.
Parallel Serial	Parallel Serial	This is a supported usage case. If needed, the user has the option to disable both the parallel and serial inputs of the child block.
Serial	Parallel Serial	Unsupported case. The user has two options: • Support fast BISR by enabling SerialRepairEnable on the parent block through setting MemoryBisr/ memory_repair_loading_method to "from_read_buffer" and re-processing DFT insertion. • Remove fast BISR entirely by disabling the parallel port of the child block. This is done by forcing the SerialRepairEnable input port to a LogicHigh.

The descriptions for Serial and Parallel memory repair loading types used in Table 5-15 are provided in Table 5-16 below.

Table 5-16. BISR Loading Terminology

Type	Description
Serial	All BISR registers in this design level are initialized by shifting the repair information from the fuse box controller. The fuse box content is decompressed and loaded serially inside the BISR registers. The memory repair ports are driven by the BISR register outputs. The memories must wait until the BISR power domain group has completed its initialization before functional operations can begin.

Table 5-16. BISR Loading Terminology (cont.)

Type	Description
Parallel	All BISR registers in this design level have a multiplexer that is used to select the source of the repair information driven to the memory repair ports. The BISR registers have an extra input port that is connected to the fuse box read buffer. This input can be selected as the repair information driven to the memory repair ports. These registers also support the serial BISR loading method.

Implementing Fast BISR Loading

The following sections describe various implementation details for including fast BISR loading in your design.

Fuse Box Interface

For Fast BISR loading implementations, the repair data in the fuse box is not compressed and is driven directly to the memory repair ports using parallel buses. The number_of_fuses property must be large enough to enable storing the repair information for the entire chip, and the entire contents of the fuse box must be available on the fuse box interface ports. This requires the fuse box interface to have a read buffer that is initialized with the fuse box contents and drives the repair values on the fuse box interface ports.

The Tessent Core Description (TCD) FuseBoxInterface provides the declaration of the fuse box size, read buffer output port name and the name of the read_buffer_select input port on the fuse box interface, as shown in the example below.

```
Core(module_name)
  FuseBoxInterface {
    number_of_fuses : 2000 ;
    Interface {
      // inputs
      read_buffer_select : FBSelect ;
      // outputs
      read_buffer_output : FBData[63:0] ;
    }
  }
}
```

The read_buffer_select and read_buffer_output properties are mandatory if the MemoryBisr/ memory_repair_loading_method parameter is set to "from_read_buffer", otherwise they are optional. If they are specified, the port names and bus range are validated against the design module. Furthermore, the bus range for read_buffer_output must be specified.

If you do not have a FuseBoxInterface wrapper associated with a fuse_box_module_interface property, or if the connections need to be made to any place other than the ports of the fuse box interface module, the ExternalFuseBoxOptions/ConnectionOverrides wrapper can be used to specify the read_buffer_select and read_buffer_output connections.

DFT Insertion

Three properties in the DftSpecification are used to insert fast BISR loading into a design. One property is used to enable the insertion and two describe the interface port names.

To enable fast BISR loading, the memory_repair_loading_method property shown below, is set to "from_read_buffer". The default BISR implementation is "serial", which is described in Table 5-16.

```
DftSpecification(module_name,id) {
  MemoryBisr {
    memory_repair_loading_method : (serial) | from_read_buffer ;
    . . .
    Interface {
      serial_repair_enable : port_naming ; // %s_bisr_serial_repair_enable
      parallel_in          : port_naming ; // %s_bisr_parallel_in
    }
  }
}
```

The serial_repair_enable and parallel_in properties specify the port names to use for the sub block or physical block interface. These ports are only created if the block contains one or more memories with parallel repair inputs. The port name specified with serial_repair_enable corresponds to SerialRepairEnable and the port name specified with parallel_in corresponds to the parallel input bus, as illustrated in Figure 5-68. These two properties are ignored unless memory_repair_loading_method is set to from_read_buffer. If they are not specified when fast BISR loading is enabled, the port names default to the names specified in the DefaultsSpecification wrapper shown below.

```
DefaultsSpecification {
  DftSpecification {
    MemoryBisr {
      ChainInterface {
        serial_repair_enable : port_naming;//%s_bisr_serial_repair_enable
        parallel_in          : port_naming;//%s_bisr_parallel_in
      }
    }
  }
}
```

During DFT insertion with fast BISR loading functionality enabled, the repair data compression logic is automatically removed from the BISR controller and the raw uncompressed repair data is stored inside the fuse box. A test data register (TDR) is also added in the BISR controller to drive the SerialRepairEnable output port. Child blocks are processed as outlined in Table 5-15 and parallel load multiplexers are added to compatible blocks that contain one or more repairable memories with parallel repair inputs.

Tessent MemoryBIST validates the connections bit-by-bit between each memory parallel BISR input and the proper bus index of the fuse box read_buffer_output port (when the BISR

controller is present at the top level), or the parallel_in ports of the current block-level design, during ICL extraction and ICL elaboration. This automated connection verification is significantly faster and more reliable than simulation-based verification. The parallel BISR inputs of memory instances can be tied off if they are not used for parallel repair.

Pattern Generation

The properties shown below are available to control fast BISR loading pattern generation for both autonomous and bisr_chain_access modes of operation for the BISR controller.

```
PatternsSpecification(module_name,id,id) {
  Patterns(pattern_name) {
    TestStep(name) {
      MemoryBisr {
        Controller(name) {
          AutonomousOptions {
            enable_bira_capture : on | (off) ;
            select_read_buffer  : on | off |(auto) ;
          }
          BisrChainAccessOptions {
            enable_bira_capture : on | (off) ;
            select_read_buffer  : on | (off) ;
          }
        }
      }
    }
  }
}
```

When the select_read_buffer property is set to on, the SerialRepairEnable signal driving the multiplexers shown in Figure 5-68 is set to "0". This enables the module's parallel inputs, that are driven by the fuse box read buffer outputs, to reach the memory repair ports. If the enable_bira_capture property is also enabled, the parallel repair inputs are also be captured inside the BISR registers at the beginning of the shift cycle and scanned out.

When the select_read_buffer property is set to off, the SerialRepairEnable signal driving the multiplexers is set to "1". This enables the repair values from the BISR register outputs to drive the memory repair ports. If the enable_bira_capture property is enabled in this case, the values from the BIRA engine are also captured inside the BIRA registers at the beginning of the shift cycle.

Generally, there is no BISR controller or fuse box present when the design level is sub_block or physical_block. These are typically inserted at the chip design level. For sub_block and physical_block design levels, the parallel repair input ports of the BISR registers are connected to a top-level port. When generating patterns for these blocks when select_read_buffer set to on, an iProc is needed that provides iForcePort statements on the primary inputs of the block to

emulate the read buffer values and avoid unknown values being captured into the BISR registers. The iProc shown below provides an example implementation:

```
iTopProc ReadBuffer{} {
  iForcePort ReadBufferParallelPortName 0
  iApply
}
```

Note that the BisrChainAccessOptions/default_write_value and BisrRegisterAccessOptions/write_value properties can only be used to initialize BISR register values with the serial repair method.

For pattern generation at the chip level with the fuse box and controller present in the design, the fast BISR loading process is initiated by loading the fuse box content into the read buffer. The methodology used for the transfer of data from the fuse box to the read buffer is dependent on the fuse box interface logic and is not documented here. This step is done manually by the end-user and consists of adding extra TestSteps that typically involve activating a FuseBoxAccess to initiate the fuse box to read buffer transfer. For further information, refer to Figure 5-57 in "Considerations Specific to Hard Incremental Repair".

External Repair

External repair gives you the capability of implementing memory repair on memories that do not have dedicated repair logic. This feature utilizes an extra memory I/O as the spare element and adds the necessary logic in the memory interface to operate it.

The following topics describe the external repair feature and how you implement it into the design flow for Tessent Shell MemoryBIST.

External Repair Overview

You typically implement memory repair with memories that contain spare column or row elements. These types of memories embed decode and multiplexing logic to access the spare elements when needed. You can implement external memory repair in situations where memories with embedded repair logic are not available, or complete testing of the repair logic is required for reliability reasons.

External repair uses one extra memory I/O as the spare element and adds the necessary logic in the memory interface to operate it. For situations where incremental repair is used, you can increase design quality and reliability through full testability of the spare I/O and external repair logic. You can implement Incremental Repair during the manufacturing process or in high-reliability systems.

Figure 5-69 shows a memory that implements external repair. The memory has five IOs and functionally, only four are used. The fifth I/O is a spare, with its input tied to 0 and the output not connected to functional logic.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Figure 5-69. Memory With External Repair
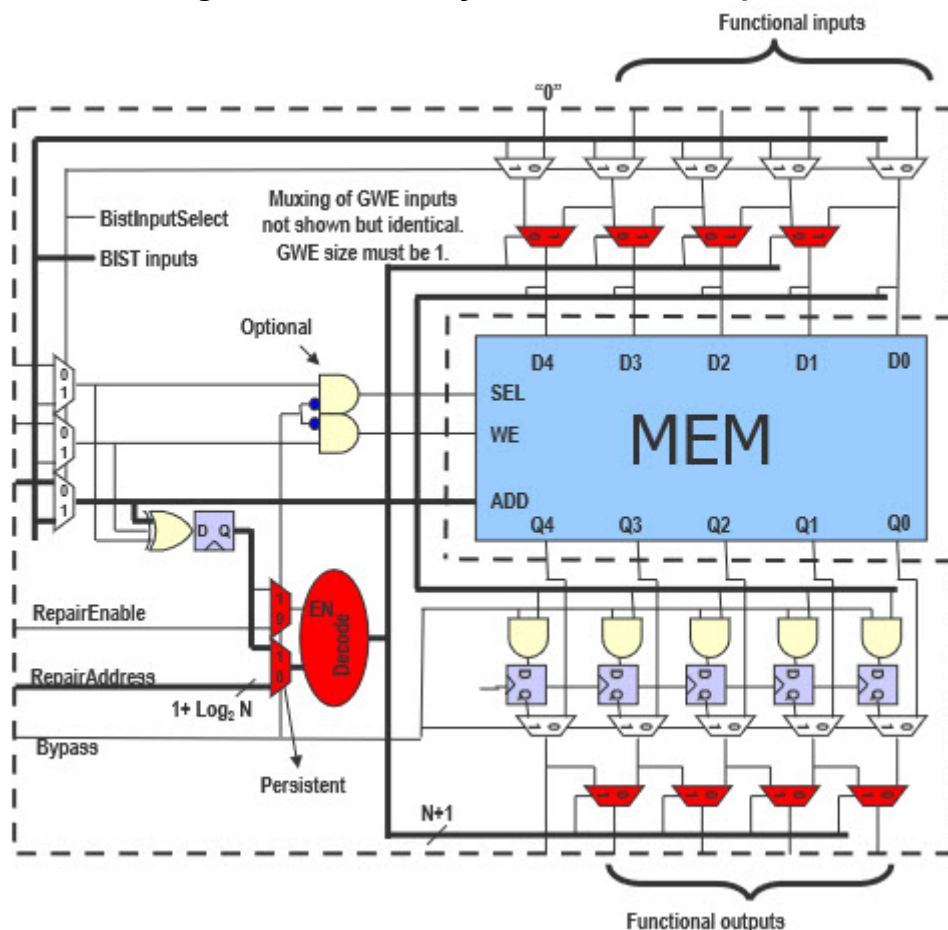
The BIST interface includes the typical logic to intercept the memory inputs for BIST, as well as the logic for observation and control to test logic in the fan-in and fan-out of the memory. The tool adds the red-highlighted logic for external repair and includes muxes for shifting the inputs and outputs according to the repair address provided by the BISR register (not shown in the figure). Also not shown, is the structure of the bit write enable muxes that intercept the bit write enable inputs are identical to that used for the data inputs.

The tool implements the repair logic so it can be fully tested using scan ATPG or logic BIST when the memory is in bypass mode. Bypass mode is configured when you specify both of the following properties as:

- scan_bypass_logic — set or inferred to sync_mux (recommended) or async_mux

- observation_xor_size — set or inferred to auto

The tool only marks the RepairEnable and RepairAddress inputs of the interface as Detected by Implication (DI). It is then possible to guarantee that any I/O can be replaced later. This might be important if the system performs memory repair. When using memories with embedded

repair logic, it is not practical to exhaustively verify the decoder and muxing logic because of their limited access.

The repair logic implementation also enables you to apply multiple load or RAM sequential patterns on a repaired memory. In that mode, while the memory has the repair solution applied, a shift at the inputs to the bypass registers could occur making the values captured by the bypass register during ATPG test unpredictable. The input of the bypass register is gated to logic 0 for this reason. For the same reason, the output of the bypass register is not observable as some of the bits might be shifted before going out of the memory interface and the values captured by registers in the memory fan-out would become unpredictable. The most significant bit of the memory output also has an unpredictable value when applying multiple load patterns to a repaired memory. However, this output only fans out to the MBIST logic, and all memory outputs are prevented from being captured when multiple load patterns are applied so that only functional paths are exercised.

Repair analysis fully tests the spare I/O at the same time as the rest of the memory, under all conditions of voltage and temperature, and incremental repair can reliably use them as needed. When using memories with embedded repair logic, extra test steps might be required to test the spare I/O. Another advantage is that if repair analysis detects a failure in both the spare and the rest of the memory, the analysis immediately declares the memory as non-repairable. This eliminates the need for running the post-repair test step, as required when implementing embedded repair logic. It is important to note that if only the spare is faulty, the analysis declares the memory repairable and programs fuses to indicate that the spare is faulty; however the memory is still usable. A subsequent analysis will declare the memory as non-repairable if it finds a fault in the rest of the memory.

Implementing External Repair

This section describes how you implement external memory repair in the Tessent Shell flow. There are very few differences from the steps described earlier in this chapter for memories with embedded repair logic; therefore you will see only the differences from the normal design flow.

Design Preparation

The first step is to generate the memories required for the functional design with one additional I/O. You then insert the memories in the design as usual, except you tie the MSB of the data input to 0 and leave the MSB of the data output open. This method enables the use of the same memory module in two different ways, which is with or without external repair. For example, you can use a 9-bit memory module as an 8-bit memory with external I/O repair, or a 9-bit memory without repair.

The Tessent Core Description (TCD) of a memory using external repair must not include a RedundancyAnalysis wrapper, whether you created the TCD manually or automatically generated it by using a memory compiler. The presence of this wrapper indicates the memory already includes repair logic.

> _____ **Note** _____
> Tessent MemoryBIST does not substitute memories in the design with memories containing an additional I/O. It is your responsibility to generate and insert the memories with the correct number of I/Os.

You can use the -external_repair_ready, -repairable, -non_repairable, and -type *memory_type* arguments for the get_memory_instances command to find and create a collection of memories that are suitable for external repair.

DFT Insertion for External Repair

When you have completed the design preparation, you next configure the memory instances and DftSpecification for memory BIST insertion. This section discusses the portion specific to configuring external memory repair.

Prerequisites

- Complete the "Design Preparation" items.

Procedure

1. During the "Adding Constraints Before Design Rule Checking" step of the Tessent MemoryBIST insertion flow, you specify the following command to enable external repair for the specified memory instances:

   ```
   set_memory_instance_options mem_inst -generate_external_repair_logic on
   ```

   MemoryBIST inserts external repair logic for the memory instances specified by the *mem_inst* field. You can specify wildcard entries, which makes it easy to specify a large number of memories.

2. Run check_design_rules to transition from setup to analysis mode:

   ```
   check_design_rules
   ```

   The bisr_segment_order_file is created and automatically populated with a BISR segment for each memory that has the generate_external_repair_logic property specified to on. For example:

   ```
   BisrSegmentOrderSpecification {
     PowerDomainGroup(-) {
       OrderedElements {
         mem_container_inst/mem_inst1;
       }
     }
   }
   ```

3. Create the DftSpecification

   ```
   create_dft_specification
   ```

   MemoryBIST automatically populates the MemoryInterface wrapper of the memories you specified for external repair with the "generate_external_repair_logic : on" property. For example:

   ```
   DftSpecification(block1,rtl) {
     MemoryBist {
       Controller(c1) {
         Step {
           MemoryInterface(m1) {
             generate_external_repair_logic : on;
             instance_name : mem_container_inst/mem_inst1;
           }
         }
       }
     }
   }
   ```

Results

MemoryBIST generates external repair logic for each memory you specified in Step 1. By default, both a repair analysis (BIRA) module and a BISR module is generated for each memory specified by the generate_external_repair_logic option. However, methods outlined elsewhere

in this chapter are available for you to exclude the BIRA module or the BIRA module. This may be necessary in custom repair flows. You can exclude the BIRA module by setting the Controller/Step/MemoryInterface/repair_analysis_present property to off in the DftSpecification. You can exclude the BISR segment by removing the memory from the bisr_segment_order_file.

The sharing of BIRA/BISR logic, as described in "Repair Sharing", is also supported. However, the external repair logic, highlighted as red in Figure 5-69, remains dedicated to each memory when you enable sharing.

External Repair Assumptions and Limitations

The following assumptions and limitations apply when implementing external memory repair. Workarounds are suggested where applicable.

1. Repair is limited to a single IO. For very wide memories (such as those with more than 64 IOs), this may limit the maximum frequency at which the BIRA module can operate. You can use the DftSpecification RepairOptions/enable_multicycle_operation property to address this limitation.

2. Memories with an additional IO must be instantiated by the user. Tessent MemoryBIST does not insert or substitute memories.

3. Ports with the function GroupWriteEnable must have the same width as the data input port.

4. Memories behind a shared bus are not supported. Most applications using a shared bus only provide access to the functional inputs and outputs. Most benefits of using external repair would be lost, even if the repair logic could be decoupled from the memory interface and placed around the physical memories.

5. Memories having scalar ports are not supported.

6. Embedded scan chains present in the design cannot capture the value of data inputs and outputs when you apply multiple load patterns. Tessent MemoryBIST does not check this since it is implementation specific.

7. Embedded test logic is not supported for memory data ports. Therefore, the TestInput and TestOutput properties are not allowed for data ports in the memory TCD.

8. When present, you must tie memory inputs with port function OutputEnable to their active value by setting the Step/MemoryInterface/output_enable_control property to "always_on" in the DftSpecification.

Chapter 6
Implementing MemoryBIST With Memory Shared Bus Interface

This chapter describes the concepts, flow, architecture, and configuration files needed to generate, insert, and verify BIST hardware for testing memories that use Shared Bus interfaces. A Shared Bus interface provides a common access port to a number of memories. This architecture enables scalability when adding memories inside a module and at the same time preserves a fixed footprint at the module boundary for memory BIST access. A typical application for Shared Bus interfaces would be testing memories that are inside processor core modules.

The terms *Shared Bus memory cluster* or *Shared Bus cluster* refer to a module that provides access to multiple memories using a common Shared Bus interface. The memories accessed through the Shared Bus interface are called *logical memories*. A logical memory is an address space that is composed of one or more *physical memories*. Library files provide descriptions of the Shared Bus memory cluster module, shared interface ports and information about the logical and physical memories. This chapter describes the steps required to create these library files and explains the tool flow that performs the generation, insertion, and verification of embedded test hardware.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Limitations

When implementing Tessent MemoryBIST with Shared Bus memory interfaces, the following limitations apply:

- Shared Bus interfaces cannot be driven in parallel by the same memory BIST controller, even if the Shared Bus interfaces are identical in terms of function.

- Shared Bus memory cluster modules with logical memories that are accessed concurrently through the Shared Bus must meet one of the following criteria:

  o logical memory data inputs do not overlap on the data inputs of the cluster module interface.

  o logical memory data input connections use identical (fully overlap) data inputs of the cluster module interface.

- When the memory access level is logical, PhysicalAddressMap and PhysicalDataMap in the physical memory libraries is not supported. The workaround is to manually code this information into the logical memory core library files.

- Because the memory BIST circuits reside outside the Shared Bus memory cluster, any memory bypass or scan observation logic required for scan test must be integrated directly inside the Shared Bus memory cluster.

- For hardware optimization reasons, only shared comparators are supported for Shared Bus memory clusters. The comparator circuit is instantiated in the memory BIST controller so that it can be shared among logical memories.

- All memories within a Shared Bus memory cluster module must belong to the same power domain group.

Applying Memory BIST to a MemoryCluster

The example design shown below is used throughout this chapter to demonstrate the steps required to generate, insert, and verify BIST hardware for testing memories via the Shared Bus interface.

The syntax of the MemoryCluster core library file is shown in Figure 6-5.

Figure 6-1. Example of CHIP With a Shared Bus Memory Cluster Module
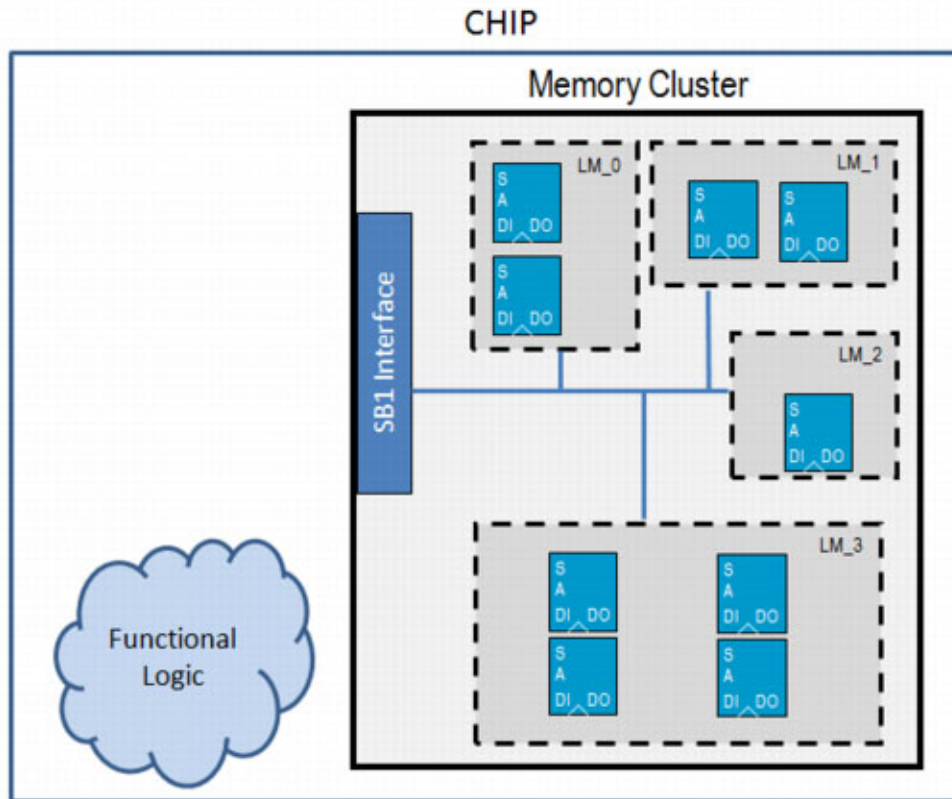
The design in Figure 6-1 has one Shared Bus interface named SB1. Four logical memories named LM_0 through LM_3 are accessible using the common Shared Bus interface. The logical memories represent an address space that is accessible from the external Shared bus interface SB1 and may be composed of one or more physical memories, which are represented by the blue boxes in Figure 6-1.

Although this example uses a Shared Bus memory cluster module with a single Shared Bus interface, a Shared Bus memory cluster module can have more than one Shared Bus interface. Each Shared Bus interface provides access to the memory data, control, and clock ports as well as other control ports required to address specific memories inside the Shared Bus memory cluster module.

The design in Figure 6-1 shows a configuration where all logical memories are located within the Shared Bus memory cluster. Depending on the IP implementation, logical memories may be instantiated outside the Shared Bus memory cluster at a different design hierarchy.

A single logical memory can be accessed at any time per Shared Bus interface. Each logical memory is enabled by specifying its corresponding selection code on the array select port of the Shared Bus interface. Once an array access code is specified on the Shared Bus interface, the corresponding logical memory can be accessed externally through the clock, data, and control ports of the Shared Bus interface.

In turn, a logical memory may be composed of one or more physical memories. This process of selecting and instantiating the physical memories in the design is called physical RAM integration. The documentation of your specific IP provides the requirements for the physical memories and guidelines for design modifications. After the RAM integration process, the Tessent libraries must be adjusted to match the physical memory implementation in the design. Refer to the "Library Requirements" section for more information.

The hierarchical nature of the logical and physical memories has several advantages:

- Early verification of the Shared Bus interface can be performed at the logical memory level using a behavioral model without physical memories.

- The user can decide on the implementation of the physical memories without affecting the overall Shared Bus access. The user can select the RAMs and stacking configuration based on their design requirements. The changes are localized within the logical level.

Figure 6-2. Composition of Logical Memory LM_1
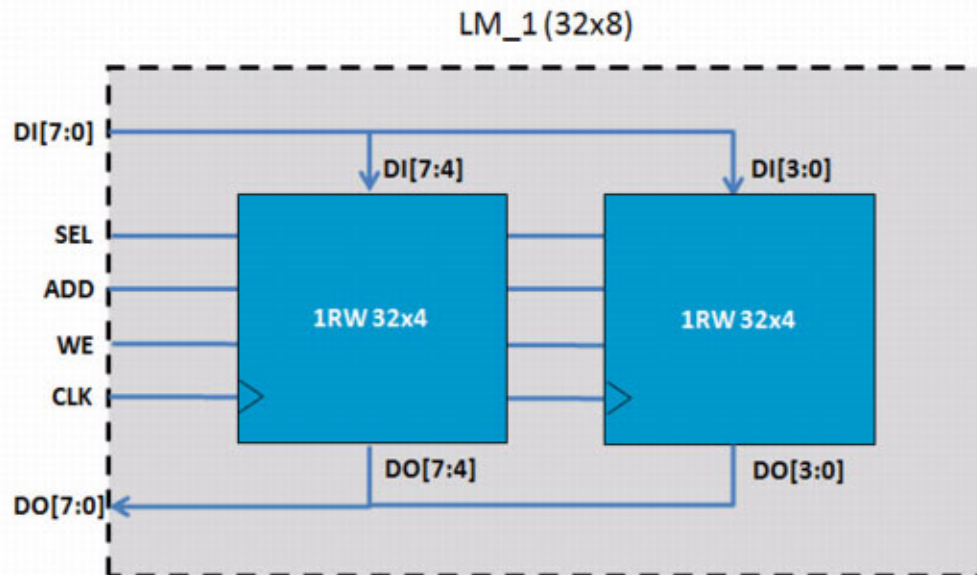
Figure 6-2 shows the composition of logical memory LM_1. The logical memory receives and drives 8 bits of the data channel on the Shared Bus interface. Two physical memories are instantiated in a horizontal stacking configuration. The 4-bit data ports of the physical memories combine to form the 8-bit data path of LM_1.

During memory BIST planning phase, one dedicated memory BIST controller is assigned per Shared Bus memory cluster module. The memory BIST controller and interface logic are instantiated at the same level as the Shared Bus memory cluster module as illustrated in Figure 6-3.

Figure 6-3. Shared Bus Memory Cluster Module After Embedded Test Insertion
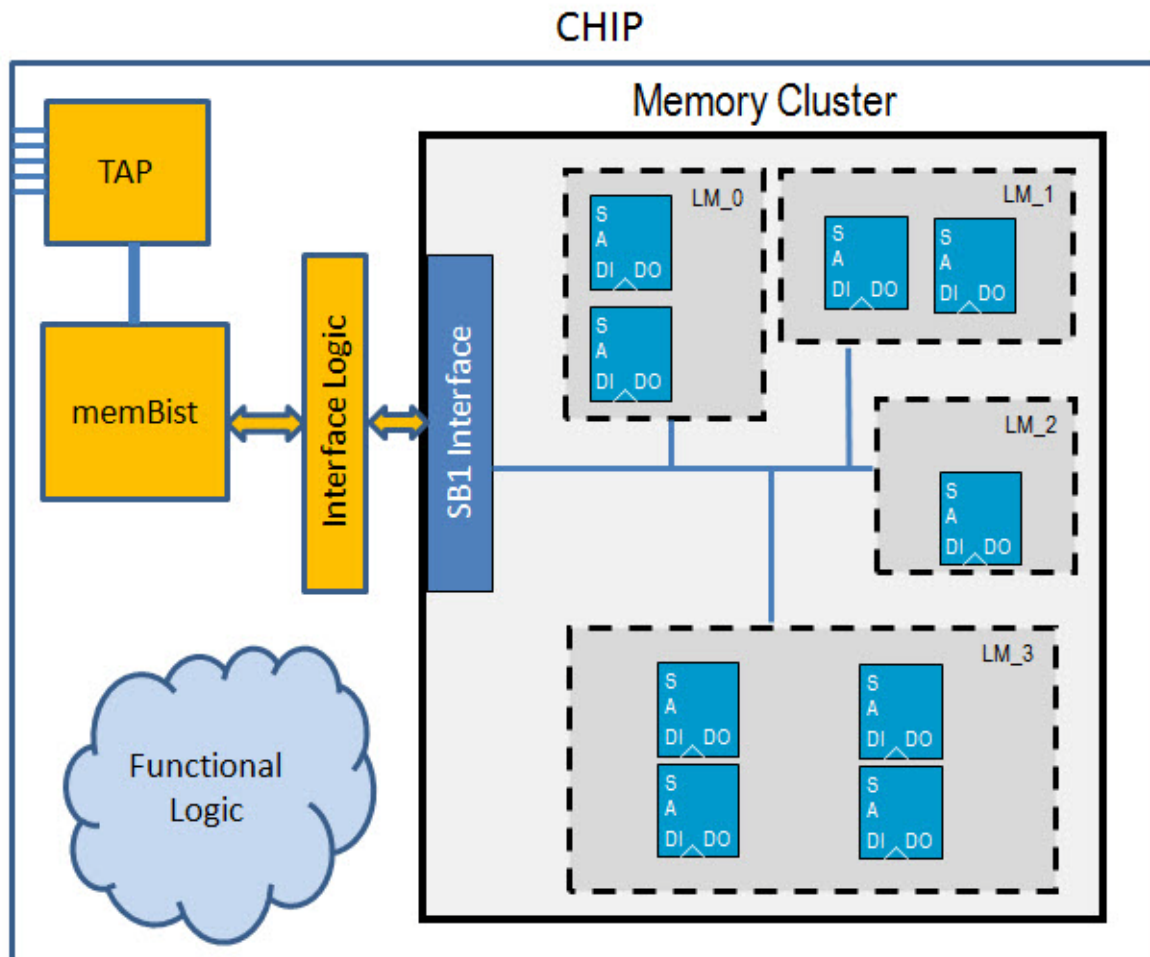
If the design contains standard memories, a different memory BIST controller is assigned to test these memories. A memory BIST controller that is assigned to a Shared Bus memory cluster module cannot be used to test memories outside the Shared Bus memory cluster. Multiple Shared Bus memory clusters can be instantiated inside the design.

Shared Bus Support Features

The embedded test hardware generated for Shared Bus interfaces is very similar to the hardware generated for standard memories and is fully supported in the hierarchical flow. A memory BIST controller and memory interfaces are generated and instantiated as usual.

Extra modules, such as memory emulation logic and multiplexing logic, also are generated and connected between the memory BIST interfaces and the memory cluster module as shown in .

Figure 6-4. Memory BIST Shared Bus Hardware Overview
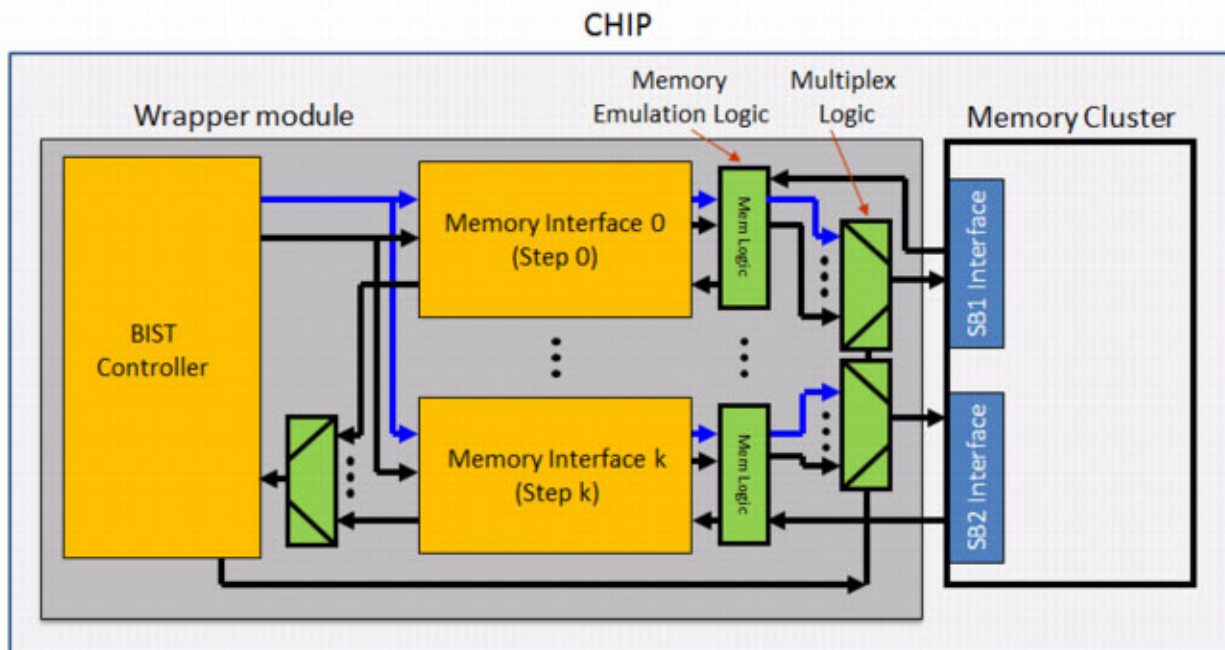
The memory emulation logic blocks shown in green correspond to the logical or physical memories inside the Shared Bus memory cluster module. One memory emulation logic module is generated for each logical or physical memory. The multiplexing logic, also shown in green, handles the control and access logic between the Shared Bus interface ports, the memory BIST controller, and the memory emulation modules. Together, the memory emulation modules and the multiplexing logic provide a virtual access to all the logical or physical memories. This enables the memory BIST controller to run the BIST algorithms and perform standard operations on all logical or physical memories.

The BIST controller, memory interface modules, memory emulation modules, and multiplexing logic are grouped inside a Shared Bus assembly module. Wrapping the BIST logic enables cross-boundary area optimization during synthesis and reduces the loose logic in the design after synthesis. The benefits of using the wrapper module are improved logic optimization and significant area reduction.

Shared Bus Interface MemoryBIST Implementation Flow

This section explains the three required memory library files and provides an overview of the various steps in the flow to test memories behind a Shared Bus interface.To a large extent, the flow is the same as the implementation for standard memory BIST with variations in terms of memory BIST logic implementation and testing.

Library Requirements

Three memory library files are required:

- Memory Cluster Tessent Core Description

  o Associated with each Shared Bus memory cluster module

  o Describes Shared Bus interface pins

  o Lists all logical memories contained within the Shared Bus memory cluster module

  o Describes pin mappings between logical memories and the Shared Bus interface

- Logical Memory Tessent Core Description

  o Associated with each logical memory in the Shared Bus memory cluster module

  o Created or modified by the user based on physical memories used

  o References library files of individual physical memories making up the logical memory

- Physical Memory Tessent Core Description

  o Associated with each physical memory

  o Used in the standard Tessent MemoryBIST flow

  o Does not need modifications if generated by a memory compiler

The Shared Bus memory cluster and logical memory TCD are usually created manually. The TCD can be automated if your IP is delivered in an electronic format describing the memory organization accessed through the Shared Bus interface.

The MBIST Information File (MBIF) format may be converted into Tessent libraries using the memlibGenerate utility. This utility is a standalone executable in the Tessent software installation tree.

For more information about the availability of the MBIF format, please contact your IP provider.

For more information about the TCD automation, refer to the application note "Creating Memory Cluster and Logical Memory Libraries Using memlibGenerate".

Memory Cluster Tessent Core Description

The TCD syntax for Shared Bus memory cluster modules is shown in the figure below.

Refer to the Core/MemoryCluster TCD description for more information.

Figure 6-5. Memory Cluster TCD Syntax

```
Core(core_name) {
  MemoryCluster {
    Port (portName) {
      Direction: InOut | (Input) | Output;
      Function: portFunction;
      SafeValue: (X) | 0 | 1;
    }
    MemoryBistInterface (interfaceName) { // Repeatable
      Port (port_name) {
        Direction: InOut | (Input) | Output;
        Polarity: (ActiveHigh) | ActiveLow;
        Function: portFunction;
        LogicalPort: port_id;
      }
      MemoryGroupAddressDecoding (GroupAddress | Address [x:y]) {
        Code (binaryValue): logical_memory_id[, logical_memory_id...];
                                        //Property is repeatable
      }
      LogicalMemoryToInterfaceMapping (logical_memory_id) {
        MemoryTemplate: logicalMemoryCoreName;
        ConfigurationData: binaryValue;
        PipelineDepth: integer;
        MemoryInstanceName: instance_path_to_logical_memory;
        PinMappings {
        // wrapper is repeatable for multi-port logical memories
          TestPortSelect: binary;
          LogicalMemoryLogicalPort(lm_logical_port_id) :
              InterfaceLogicalPort(cluster_interface_logical_port_id);
          LogicalMemoryDataInput[indexList]:
              InterfaceDataInput[indexList];
          LogicalMemoryDataOutput[indexList]:
              InterfaceDataOutput[indexList];
          LogicalMemoryAddress[indexList]:
              InterfaceAddress[indexList];
          LogicalMemoryWriteAddress[indexList]:
              InterfaceWriteAddress[indexList];
          LogicalMemoryReadAddress[indexList]:
              InterfaceReadAddress[indexList];
          LogicalMemoryGroupWriteEnable[indexList]:
              InterfaceGroupWriteEnable[indexList];
        }
      }
    }
  }
}
```

Note

When you have both 1R1W and 1RW memories on the same Shared Bus memory cluster and there are Address and WriteAddress ports functions defined on the cluster MemoryBistInterface for driving the address ports of the 1R1W memory, the 1RW memories should connect to the address port with function Address and use *LogicalMemoryAddress[]: InterfaceAddress[]* property to define the address port mapping.

Logical Memory Tessent Core Description

The logical memory library file uses a superset of the memory core library syntax, specifically the addition of MemoryGroupAddressDecoding and PhysicalToLogicalMapping wrappers.

The logical memory TCD is an extension of the memory TCD. Figure 6-6 shows the syntax applicable to logical memories. Refer to the Core/Memory TCD description for more information.

Figure 6-6. Logical Memory TCD Syntax

```
Core(core_name) {
Memory {
  Algorithm: algo_name;
  Port(port_name) {
    Direction: InOut | (Input) | Output;
    Polarity: (ActiveHigh) | ActiveLow;
    Function: function_type;
    LogicalPort: lm_logical_port_id;
  }
  AddressCounter {
    Function (Address) {
      LogicalAddressMap {
        ColumnAddress[x:y]: Address[a:b];
        RowAddress[x:y]: Address[a:b];
        BankAddress[x:y]: Address[a:b];
      }
    }
    Function (ColumnAddress | RowAddress | BankAddress) {
      CountRange [lowRange:highRange];
    }
  }

  MemoryGroupAddressDecoding (Address [a:b]) {
    Code (binaryValue): physical_memory_id[, physical_memory_id...];
                                        //Property is repeatable
  }
```

```
PhysicalToLogicalMapping(physical_memory_id) {
  MemoryTemplate: physicalTemplateName;
  PinMappings {
  // wrapper is repeatable for multi-port logical memories
    PhysicalMemoryLogicalPort(pm_logical_port_id):
        LogicalMemoryLogicalPort(lm_logical_port_id);
    PhysicalMemoryDataInput[<indexList>]:
        LogicalMemoryDataInput[<indexList>];
    PhysicalMemoryDataOutput[<indexList>]:
        LogicalMemoryDataOutput[<indexList>];
    PhysicalMemoryAddress[<indexList>]:
        LogicalMemoryAddress[<indexList>];
    PhysicalMemoryWriteAddress[<indexList>]:
        LogicalMemoryWriteAddress[<indexList>];
    PhysicalMemoryReadAddress[<indexList>]:
        LogicalMemoryReadAddress[<indexList>];
    PhysicalMemoryGroupWriteEnable[<indexList>]:
        LogicalMemoryGroupWriteEnable[<indexList>];
  }
 }
}
}
```

During the RAM integration process, the user is free to implement the logical memory address space using any available memory. Once the process is completed, the logical memory TCD must be edited to match the implementation.

The TCD modifications include adding the MemoryGroupAddressDecoding wrapper to indicate how the physical memories are activated, and PhysicalToLogicalMapping wrappers to indicate how the physical memories are connected to the ports of the logical memory.

Figure 6-7. Example Logical Memory With Four Physical Memories
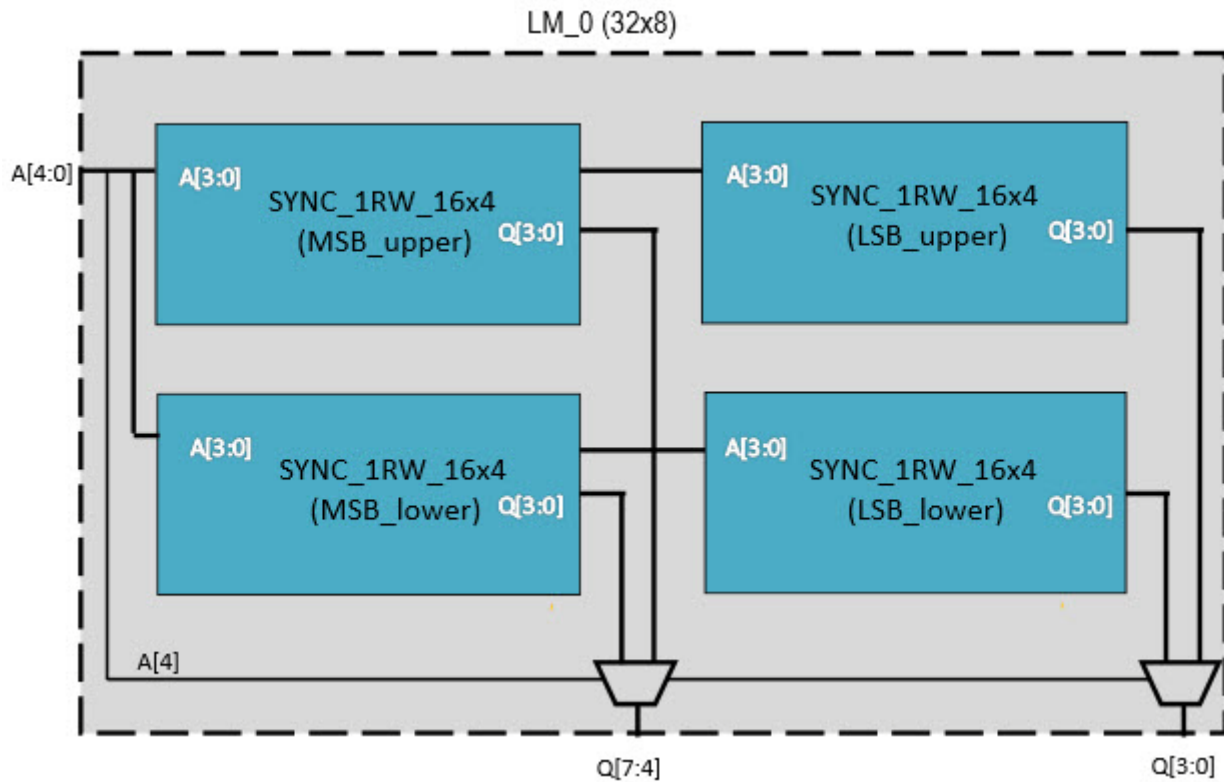
Figure 6-7 shows logical memory LM_0 implemented with 4 physical memories. The logical memory size is 32 words. It receives and drives 8 bits of the data channel on the Shared Bus interface.

In the logical memory TCD, the data input, data output, address port and the address segmentation are defined as shown below:

```
Port(D[7:0]) {
    Function: Data;
    Direction: Input;
}
Port(Q[7:0]) {
    Function: Data;
    Direction: Output;
}
Port(A[4:0]) {
    Function: Address;
    Direction: Input;
}
AddressCounter {
    Function(Address) {
        ColumnAddress[0:0] : Address[0:0];
        RowAddress[3:0]    :  Address[4:1];
    }
}
```

Four identical physical memories are instantiated in a 2 by 2 stacking configuration. The 4-bit data ports of the physical memories combine to form the 8-bit data path of LM_0. Logical memory address A[4] selects the upper or lower pairs of memories. The remaining address bits A[3:0] control the address inputs of the physical memories. Assume that the logical memory clock and control signals propagate to all physical memories. When A[4] is 0, memories MSB_lower and LSB_lower are enabled. When A[4] is 1, memories MSB_upper and LSB_upper are enabled. As a result, MSB_lower and LSB_lower are tested concurrently, followed by MSB_upper and LSB_upper.

The MemoryGroupAddressDecoding wrapper indicates the address signal used to activate the physical memories and defines the selection codes:

```
MemoryGroupAddressDecoding(Address[4]) {
    Code(1'b0): MSB_lower, LSB_lower;
    Code(1'b1): MSB_upper, LSB_upper;
}
```

The PhysicalToLogicalMapping wrapper associates the data, address and group write enable ports of the physical memory to the ports of the logical memory. Clock and other control signals, such as write and read enables, are assumed to be broadcast to all physical memories. The tool implicitly maps the physical memory port to the logical memory port if they are defined with the same port function. One wrapper is required per physical memory:

```
PhysicalToLogicalMapping(MSB_lower) {
    MemoryTemplate: SYNC_1RW_16x4;
    PinMappings {
        PhysicalMemoryDataInput[3:0]  : LogicalMemoryDataInput[7:4];
        PhysicalMemoryDataOutput[3:0] : LogicalMemoryDataOutput[7:4];
        PhysicalMemoryAddress[3:0]    : LogicalMemoryAddress[3:0];
    }
}

PhysicalToLogicalMapping(LSB_lower) {
    MemoryTemplate: SYNC_1RW_16x4;
    PinMappings {
        PhysicalMemoryDataInput[3:0]  : LogicalMemoryDataInput[3:0];
        PhysicalMemoryDataOutput[3:0] : LogicalMemoryDataOutput[3:0];
        PhysicalMemoryAddress[3:0]    : LogicalMemoryAddress[3:0];
    }
}

PhysicalToLogicalMapping(MSB_upper) {
    MemoryTemplate: SYNC_1RW_16x4;
    PinMappings {
        PhysicalMemoryDataInput[3:0]  : LogicalMemoryDataInput[7:4];
        PhysicalMemoryDataOutput[3:0] : LogicalMemoryDataOutput[7:4];
        PhysicalMemoryAddress[3:0]    : LogicalMemoryAddress[3:0];
    }
}
```

```
PhysicalToLogicalMapping(LSB_upper) {
    MemoryTemplate: SYNC_1RW_16x4;
    PinMappings {
        PhysicalMemoryDataInput[3:0]  : LogicalMemoryDataInput[3:0];
        PhysicalMemoryDataOutput[3:0] : LogicalMemoryDataOutput[3:0];
        PhysicalMemoryAddress[3:0]    : LogicalMemoryAddress[3:0];
    }
}
```

If the physical memories implement redundancy, then their hierarchical instance path must be specified in order for Tessent MemoryBIST to insert the BIRA and BISR circuit. Refer to the "Handling MemoryCluster Modules With Repairable Memories" section for more information.

Pseudo-Vertical Stacking of Physical Memory

The pseudo-vertical stacking configuration is where you implement a W x D physical memory to comprise a logical memory interface dimension of $(n*W)$ x (D/n).

The variable W represents the number of physical memory words, D is the physical memory data path width, and n is typically a power of two value which corresponds to the number of partitions you split the physical memory data path into. The logical memory can contain many physical memories using the pseudo-vertical stacking configuration.

Figure 6-8. Pseudo-Vertical Stacking of Physical Memory
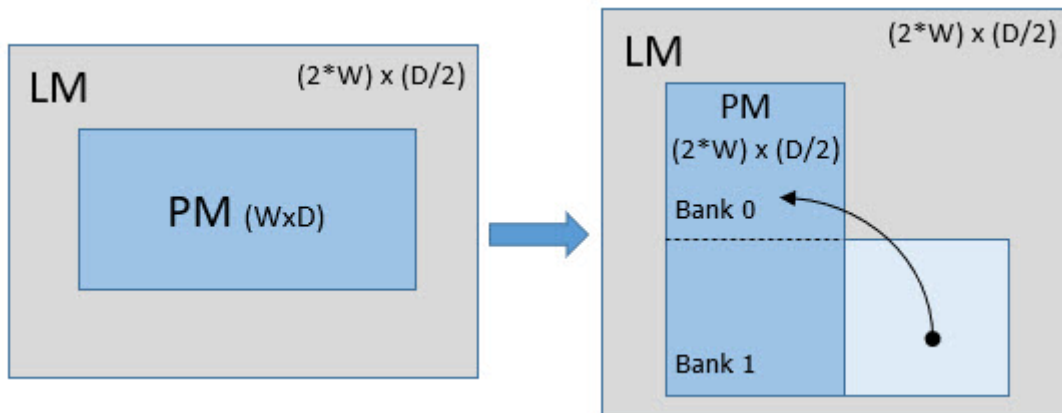
Figure 6-9 shows an example logical memory LM_64x8 with a single 32x16 physical memory. In this example implementation, the lower data bits serve as the logical memory's lower bank, and the higher data bits serve as the upper bank.

Figure 6-9. Example Logical Memory With Pseudo-Stacked Physical Memory
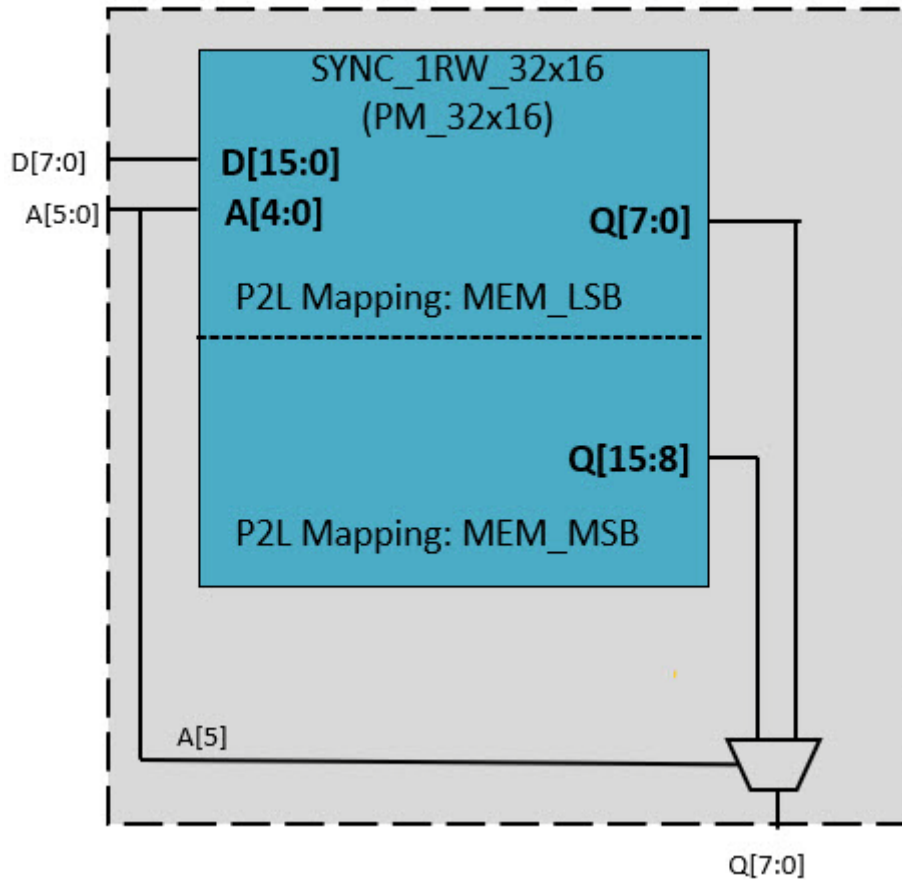
The logical memory TCD defines the data input, data output, address port, and the address segmentation as shown below:

```
Port(A[5:0]) {
   Function  : Address;
   Direction : Input;
}
Port(D[7:0]) {
   Function  : Data;
   Direction : Input;
}
Port(Q[7:0]) {
   Function  : Data;
   Direction : Output;
}
```

```
AddressCounter {
    Function(ColumnAddress) {
      LogicalAddressMap {
        ColumnAddress[0] : Address[0];
        ColumnAddress[1] : Address[1];
        ColumnAddress[2] : Address[5];
      }
      CountRange [0:7];
    }
    Function(RowAddress) {
      LogicalAddressMap {
        RowAddress[0] : Address[2];
        RowAddress[1] : Address[3];
        RowAddress[2] : Address[4];
      }
      CountRange [0:7];
    }
}
```

Logical memory address A[5] splits the physical memory data path in two parts by selecting the upper or lower portion, or bank, of the physical memory data path bits. The remaining logical memory address A[4:0] bits control the address inputs of the physical memory.

The MemoryGroupAddressDecoding wrapper indicates the address signal used to activate the wanted physical memory data path split and defines the selection code:

```
MemoryGroupAddressDecoding (Address[5]) {
    Code(1'b0) : MEM_LSB;
    Code(1'b1) : MEM_MSB;
}
```

The PhysicalToLogicalMapping wrapper associates the data, address, and group write enable ports of the physical memory to the ports of the logical memory. From the perspective of the logical memory TCD, pseudo-vertical stacking of a physical memory is implemented by specifying multiple PhysicalToLogicalMapping wrappers, where each references the same physical memory instance with the MemoryInstanceName property. The PhysicalToLogicalMapping wrappers for the example outlined in this section are shown below. Note how the logical memory data input ports are mapped to the physical memory data input ports, where each logical memory data bit is broadcast to two physical memory data bits.

```
PhysicalToLogicalMapping(MEM_LSB) {
  MemoryTemplate : SYNC_1RW_32x16_RC_BISR;
  MemoryInstanceName : PM_32x16;
  PinMappings {
    PhysicalMemoryDataInput[7:0]   : LogicalMemoryDataInput[7:0];
    PhysicalMemoryDataOutput[7:0] : LogicalMemoryDataOutput[7:0];
    PhysicalMemoryAddress[4:0]     : LogicalMemoryAddress[4:0];
  }
}
PhysicalToLogicalMapping(MEM_MSB) {
  MemoryTemplate : SYNC_1RW_32x16_RC_BISR;
  MemoryInstanceName : PM_32x16;
  PinMappings {
    PhysicalMemoryDataInput[15:8]  : LogicalMemoryDataInput[7:0];
    PhysicalMemoryDataOutput[15:8] : LogicalMemoryDataOutput[7:0];
    PhysicalMemoryAddress[4:0]     : LogicalMemoryAddress[4:0];
  }
}
```

If the physical memories implement redundancy, then their hierarchical instance path must be specified in order for Tessent MemoryBIST to insert the BIRA and BISR circuit. Refer to the "Handling MemoryCluster Modules With Repairable Memories" section for more information.

Requirements and Limitations

Requirements

- If the TCD for a physical memory used for pseudo-vertical stacking contains a PhysicalDataMap wrapper, it is required that the mapping between the data input ports and the internal data lines in the memory are symmetrical among all parts of the physical memory into which the data path is split. The tool will automatically ignore the PhysicalDataMap if it is of the style described in the "Use the Physical Data Map Correctly" sub-topic of "Optimization Recommendations". Otherwise, a non-symmetrical PhysicalDataMap cannot be ignored and the tool will switch to the logical access level.

- Any unused bits of the physical memory must be MSBs of each memory partition defined in the corresponding PhysicalToLogicalMapping wrapper. Otherwise, the physical memory access level is not supported for the logical memory.

  The PhysicalToLogicalMapping wrappers shown below illustrate the proper configuration of the example PM_32x16 physical memory in implementing a 64x7 logical memory. There is a single unused physical memory bit (MSB) in each wrapper.

```
PhysicalToLogicalMapping(MEM_LSB) {
  MemoryTemplate : SYNC_1RW_32x16_RC_BISR;
  MemoryInstanceName : PM_32x16;
  PinMappings{
    PhysicalMemoryDataInput[6:0] : LogicalMemoryDataInput[6:0];
    PhysicalMemoryDataOutput[6:0]: LogicalMemoryDataOutput[6:0];
    PhysicalMemoryAddress[4:0]   : LogicalMemoryAddress[4:0];
  }
}
```

```
PhysicalToLogicalMapping(MEM_MSB) {
  MemoryTemplate : SYNC_1RW_32x16_RC_BISR;
  MemoryInstanceName : PM_32x16;
    PinMappings {
      PhysicalMemoryDataInput[14:8] : LogicalMemoryDataInput[6:0];
      PhysicalMemoryDataOutput[14:8]: LogicalMemoryDataOutput[6:0];
      PhysicalMemoryAddress[4:0]    : LogicalMemoryAddress[4:0];
    }
  }
```

An unsupported configuration would be realized if a PhysicalToLogicalMapping wrapper was configured to leave an unused physical memory bit in a LSB position, as illustrated by the example below:

```
PhysicalToLogicalMapping(MEM_LSB) {
  MemoryTemplate : SYNC_1RW_32x16_RC_BISR;
  MemoryInstanceName : PM_32x16;
  PinMappings{
    PhysicalMemoryDataInput[7:1] : LogicalMemoryDataInput[6:0];
    PhysicalMemoryDataOutput[7:1]: LogicalMemoryDataOutput[6:0];
    PhysicalMemoryAddress[4:0]    : LogicalMemoryAddress[4:0];
  }
}
PhysicalToLogicalMapping(MEM_MSB) {
  MemoryTemplate : SYNC_1RW_32x16_RC_BISR;
  MemoryInstanceName : PM_32x16;
    PinMappings {
      PhysicalMemoryDataInput[14:8] : LogicalMemoryDataInput[6:0];
      PhysicalMemoryDataOutput[14:8]: LogicalMemoryDataOutput[6:0];
      PhysicalMemoryAddress[4:0]    : LogicalMemoryAddress[4:0];
    }
  }
```

Limitations

- Use of the report_memory_cluster_configuration command prior to executing the check_design_rules command results in the logical memory access level reported for the logical memory implementing pseudo-vertical stacking. The report_memory_cluster_configuration command should be run after check_design_rules to verify the physical access level is reported.

- The memory TCD syntax enables you to specify the RedundancyAnalysis/ColumnSegment/ShiftedIORange property in the form of a bused port or as scalar ports separated by a comma. Specifying this property as scalar ports for the pseudo-vertical stacking configuration is not supported and is rule-checked by the tool.

Physical Memory Tessent Core Description

The physical memory library file is the same as the standard memory library file used for memory BIST without Shared Bus.

Please refer to the Tessent Core Description (Core/Memory) for further details on the physical memory core library.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Shared Bus Learning

The organization and access mechanism of the memories attached to the shared bus interface must be described to Tessent MemoryBIST in the form of a Tessent Core Description (TCD) within the memory library files. The process of creating the TCDs and validating them against the memory cluster core in the design has typically been a manual one, but much of it can be automated with the shared bus learning flow.

An overview of the shared bus learning flow is shown in Figure 6-10. It is recommended that you implement the flow on a single, stand-alone shared bus memory cluster core prior to DFT insertion. Automation is available for Steps 1, 4 and 5 in the flow and you intercept the normal DFT insertion flow at Step 6.

Figure 6-10. Shared Bus Learning Flow
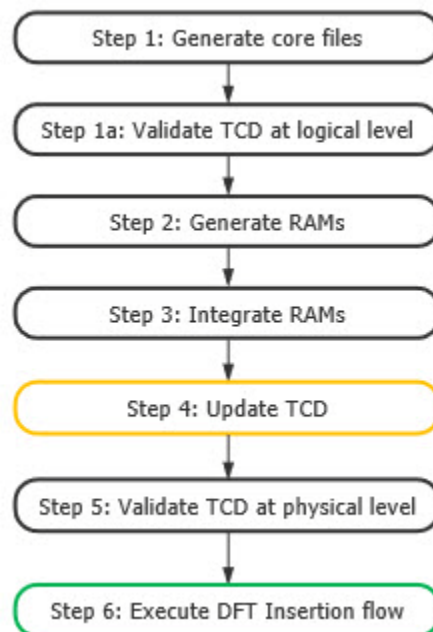
Step 1 is where you generate the memory cluster core RTL, instantiate the core into the design, and create the memory cluster and logical memory TCD. As outlined in "Library Requirements", the memory cluster TCD and logical memory TCD may be created manually, which is usually the case for users designing their own memory clusters. For commercial cores, your IP provider may deliver the IP in an MBIST Information File (MBIF) electronic format that describes the memory organization accessed through the shared bus interface. The generation of the TCD can be automated if your IP is available in MBIF format through the use of the *memlibGenerate* utility, located in the Tessent software installation tree.

Steps 2 and 3 is where physical RAM integration is done. In this process, you select and instantiate physical memories into each logical memory. Refer to the documentation from your IP provider for specific requirements and design guidelines for integrating the physical memory.

Step 4 updates the logical memory TCD to match the physical memory implementation after RAM integration is complete. The set_memory_cluster_library_generation_options command automates this learning process, which is described in "Physical-to-Logical (P2L) Mapping Automation".

Step 5 is where a tracing-based validation is performed to verify the memory cluster and logical memory TCD with the memory cluster core RTL. The set_memory_cluster_validation_options command automates this process, which is described in "Library Validation".

Step 6 is where you perform the normal DFT flow on your design containing the fully populated memory cluster core. This step generates the DFT IPs, inserts them into your design, and verifies their operation.

The file usage and data processing flow for shared bus learning is shown in the figure below. The P2L automation applied in Step 4 creates updated TCD files based on the structure learned from examining the memory cluster core RTL.

Figure 6-11. Shared Bus Learning Data Processing Flow

Physical-to-Logical (P2L) Mapping Automation

During the RAM integration process, you can implement the logical memory address space using any available physical memory, and multiple physical memories may be used.

For example, in Figure 6-1, logical memory LM_2 instantiates one physical memory whereas logical memory LM_1 instantiates two physical memories in a horizontal stacking configuration. Both physical memories of LM_1 are enabled concurrently and their data ports combine to form the data path of LM_1. Logical memory LM_3 incorporates horizontal and vertical stacking configurations where two physical memories are active at a given time. Each pair of memories form the lower or upper address range, depending on the address value.

After addition of physical memories into the cluster core RTL, the logical memory TCD must indicate how the physical memories will be activated during memory BIST. The logical memory TCD must reflect the connectivity of the physical memories to the logical memory ports. This is accomplished with the addition of two types of wrappers to the Logical Memory Tessent Core Description files:

- MemoryGroupAddressDecoding wrapper — This wrapper specifies the address signals used to activate the physical memories and defines the selection codes. It is needed when a vertical stacking configuration is present.

- PhysicalToLogicalMapping wrapper — This wrapper associates the data, address, and group write enable ports of the physical memory to the ports of the logical memory.

The set_memory_cluster_library_generation_options command automates the creation of these wrappers. The command derives the mappings of the physical memories within a logical memory from the memory cluster core RTL. The command populates the extracted information into the logical memory TCD and writes out the updated files. The following section describes how to use this command.

Implementing the P2L Mapping Flow

The following procedure describes how to use the set_memory_cluster_library_generation_options command to automatically map a memory cluster core. The process updates the logical memory TCD files with the MemoryGroupAddressDecoding and PhysicalToLogicalMapping wrappers that describe the physical memory configurations within each logical memory.

Prerequisites

- Design that contains only the memory cluster core and its physical memories

- Memory cluster TCD

- Logical memory TCD without P2L information

- Physical memory TCD

Procedure

1. Load the design and memory libraries

   The design containing a single memory cluster core and physical memories is loaded and references to the TCD files for the memory cluster, the logical memories, and the physical memories are loaded. This step is identical to the standard DFT design flow and is done within the DFT context of Tessent Shell.

   ```
   set_context dft -rtl

   # Load the library for the technology cells
   read_cell_library ../tech_cells.lib

   # Search paths for memory TCD files
   # They may be read in explicitly using read_core_descriptions
   set_design_sources -format tcd_memory -y MEM -extensions {lib lvlib}

   # Search paths for design files
   # They may be read in explicitly using read_verilog
   set_design_sources -format verilog -y RTL -extensions {v vb}

   # Load the top-level design
   read_verilog RTL/WIRELESS_CORE.vb
   ```

2. Design elaboration and setup

   The full design is elaborated and the tool is configured for memory test, which includes analysis of the memory cluster. This step is also identical to the standard DFT design flow.

   ```
   set_current_design WIRELESS_CORE

   # Define the clock reaching the core
   add_clocks clk -period 12ns

   # Enable analysis for memory test
   set_design_level sub_block
   set_dft_specification_requirements -memory_test on
   ```

3. Enable the P2L mapping

   While in setup system mode, enable P2L mapping using the set_memory_cluster_library_generation_options command. Refer to the command description to determine the mapping configuration options you want to include.

   ```
   set_memory_cluster_library_generation_options \
     -generate_physical_to_logical_info

   // Warning: The default memory cluster initialization file for
   // cluster libraries generation was created.

   // Edit this file manually if the default initialization sequence
   // needs to be modified.
   ```

The command creates a memory cluster initialization file. This is a Tessent dofile that the tool will source to configure the memory cluster into MemoryBIST mode during the next circuit analysis phase. The initialization file is saved in the current working directory. An example file name is shown below:

```
WIRELESS_CORE_rtl.memory_cluster_mbist_mode_init
```

Normally, the initialization file requires no modification. It may be customized to adjust the protocol to enable MemoryBIST mode for the core and tracing of the physical memories within the core. If the tool reports problems with the application of the memory cluster initialization sequence, refer to the "Enabling MemoryBIST Mode for a Cluster" topic.

4. Generate the P2L mapping

The circuit analysis and extraction of the physical-to-logical mapping occurs during the transition from system mode setup to system mode analysis, as shown in the following example:

```
check_design_rules

// ----------------------------------------------------------------------------
// Begin RTL synthesis.
// -------------------
// Synthesized modules=18, Time=1.09 sec.
// Note: There were 10 modules selectively synthesized. There were also 6 sub-modules
// created by synthesis.
//    Use 'get_module -filter is_synthesized' to see them.
//    You can also use 'set_quick_synthesis_options -verbose on' to have the
//    synthesis step report the synthesized module names in the transcript as
//    they are being synthesized.
// ----------------------------------------------------------------------------
// Warning: Rule FP2 violation occurs 1 times
// Flattening process completed, cell instances=482, gates=918, PIs=2, POs=0,
// CPU time=0.01 sec.
// ----------------------------------------------------------------------------
// Begin circuit learning analyses.
// -------------------------------
// Learning completed, CPU time=0.00 sec.
// ---------------------------------------
// Begin Shared Bus memory cluster library generation.
// Memory Cluster Instance: 'CLUSTERInst'.
// Memory Bist interface ID: 'I1'.
//    Processing of LM_0 logical memory.
//    Processing of LM_1 logical memory.
//    Processing of LM_2 logical memory.
//    Processing of LM_3 logical memory.
//    Processing of LM_4 logical memory.
// Shared Bus memory cluster library generation completed.
// ---------------------------------------
```

The circuit analysis is based on the flat design model, which is an internal, flattened representation of the hierarchical design. The memory cluster initialization file is sourced to condition the memory cluster core RTL for memory test. The tool applies the

selection code for each logical memory. By tracing through the flat model, the tool attempts to locate the physical memory instances and to determine the stacking configuration within the logical memory.

Results

Upon successful extraction, the P2L mappings are populated into the logical memory TCD within Tessent Shell. By default, the complete logical memory TCD is written to the *memory_tcd_outdir* folder found in the current working directory. The output filenames are the same as the original logical memory TCD files.

The new files include updates to the logical memory templates. You should review the added PhysicalToLogicalMapping wrappers for completeness. If the MemoryInstanceName property does not appear in the PhysicalToLogicalMapping wrapper, refer to the limitations listed in "Shared Bus Learning Assumptions and Limitations" for possible reasons and determine if you need to add the property manually. If the original file contains other types of configuration data, they are not preserved after P2L generation and must be retrieved from the original file.

_____**Note**_____
Siemens EDA recommends defining custom algorithms and operation sets in files separate from the cluster and logical memory templates.

Library Validation

The shared bus memory cluster library validation implements a tracing-based approach, where the content of the memory cluster TCD and logical library TCDs are validated against the data collected from the learned memory cluster structure. The trace-based validation approach is an extension of the library validation performed in the DFT flow, where the focus is on the consistency of the semantics among the TCDs for the memory cluster, logical memories, and physical memories.

Shared bus memory cluster library validation provides the following functionality:

- Validation that no memory is missing memory BIST, including instances that are determined to be physical memories connected to a shared bus interface, but are not accessible using the selection codes specified in the memory cluster TCD.

- Validation of the mappings specified in the LogicalMemoryToInterfaceMapping and PhysicalToLogicalMapping wrappers. This includes the following paths:

  - DataInput

  - DataOutput

  - Address

  - ReadAddress

  - WriteAddress

  - GroupWriteEnable

- Validation of the PipelineDepth property

Validation is based on user-provided matching expressions, meaning that it is necessary for you to specify a pattern using wildcards or regular expressions that enables the tool to identify physical memory modules before the validation starts. The following sections describe how to use the set_memory_cluster_validation_options command to perform the validation.

Implementing the Library Validation Flow

The following procedure describes how to use the set_memory_cluster_validation_options command to perform a trace-based validation of the memory cluster TCD, logical memory TCDs, and physical memory TCDs against the structure traced from the memory cluster core RTL.

Prerequisites

- Design that contains only the memory cluster core and its physical memories

- Memory cluster TCD

- Logical memory TCDs with P2L information

- Physical memory TCDs

Procedure

1. Load the design and memory libraries

   The design containing a single memory cluster core and physical memories is loaded and references to the TCD files for the memory cluster, the logical memories, and the physical memories are loaded. This step is identical to the standard DFT design flow and is done within the DFT context of Tessent Shell.

   ```
   set_context dft -rtl

   # Load the library for the technology cells
   read_cell_library ../tech_cells.lib

   # Search paths for memory TCD files
   # They may be read in explicitly using read_core_descriptions
   set_design_sources -format tcd_memory -y MEM -extensions {lib lvlib}

   # Search paths for design files
   # They may be read in explicitly using read_verilog
   set_design_sources -format verilog -y RTL -extensions {v vb}

   # Load the top-level design
   read_verilog RTL/WIRELESS_CORE.vb
   ```

2. Design elaboration and setup

   The full design is elaborated and the tool is configured for memory test, which includes analysis of the memory cluster. This step is also identical to the standard DFT design flow.

   ```
   set_current_design WIRELESS_CORE

   # Define the clock reaching the core
   add_clocks clk -period 12ns

   # Enable analysis for memory test
   set_design_level sub_block
   set_dft_specification_requirements -memory_test on
   ```

3. Enable the memory cluster TCD and logical memory TCD validation

Enable shared bus library validation while in setup system mode by using the
set_memory_cluster_validation_options command:

```
set_memory_cluster_validation_options {SYNC.* RAM.*} -regexp

// Warning: The default memory cluster initialization file for
// cluster libraries validation was created.
// Edit this file manually if the default initialization sequence
// needs to be modified.
```

The command creates a memory cluster initialization file. This is a Tessent dofile that
the tool will source to configure the memory cluster into MemoryBIST mode during the
next circuit analysis phase. The initialization file is saved in the current working
directory. An example file name is shown below:

```
WIRELESS_CORE_rtl.memory_cluster_mbist_mode_init
```

Normally, the initialization file requires no modification. It may be customized to adjust
the protocol to enable MemoryBIST mode for the core and tracing of the physical
memories within the core. If the tool reports problems with the application of the
memory cluster initialization sequence, refer to the "Enabling MemoryBIST Mode for a
Cluster" topic.

4. Validate the memory cluster TCD and logical memory TCD

The circuit analysis and validation occurs during the transition from system mode setup to system mode analysis, as shown in the following example:

```
check_design_rules

// ---------------------------------------------------------------------------
// Begin RTL synthesis.
// -------------------
// Synthesized modules=18, Time=1.09 sec.
// Note: There were 10 modules selectively synthesized. There were also 6 sub-modules
// created by synthesis.
//    Use 'get_module -filter is_synthesized' to see them.
//    You can also use 'set_quick_synthesis_options -verbose on' to have the
//    synthesis step report the synthesized module names in the transcript as
//    they are being synthesized.
// ---------------------------------------------------------------------------
// Warning: Rule FP2 violation occurs 1 times
// Flattening process completed, cell instances=482, gates=918, PIs=2, POs=0,
// CPU time=0.01 sec.
// ---------------------------------------------------------------------------
// Begin circuit learning analyses.
// -------------------------------
// Learning completed, CPU time=0.00 sec.
// -------------------------------------
// Begin Shared Bus memory cluster library validation.
// Memory Cluster Instance: 'CLUSTERInst'.
// Memory Bist interface ID: 'I1'.
//    Processing of LM_0 logical memory.
//    Processing of LM_1 logical memory.
//    Processing of LM_2 logical memory.
//    Processing of LM_3 logical memory.
//    Processing of LM_4 logical memory.
// Shared Bus memory cluster library validation completed.
// -------------------------------------
```

The circuit analysis is based on the flat design model, which is an internal, flattened representation of the hierarchical design. The memory cluster initialization file is sourced to condition the memory cluster core RTL for memory test. The tool applies the selection code for each logical memory. By tracing through the flat model, the tool attempts to locate the physical memory instances and to determine the stacking configuration within the logical memory.

Memory Cluster TCD Validation at the Logical Level

The typical usage for the set_memory_cluster_validation_options command in the shared bus learning flow is to validate the final memory cluster TCD and logical memory TCD after the RAM integration process. The command also provides you with the option of performing validation at the logical level only. In this usage, you would be validating the memory cluster TCD before you have implemented the physical memories of the chosen technology into the design.

Validation at logical level requires that you identify logical memories in the design. Logical memories should be considered as memory modules, which are activated using unique selection

codes from the memory cluster TCD. Clear logical memory boundaries may or may not exist in the design, therefore in the RTL before the RAM integration phase, generic (behavioral) RAMs should be instantiated and considered as the logical memories. Each selection code from the memory cluster TCD will activate a single generic RAM. Logical-level validation will not succeed in cases where clear logical memory boundaries do not exist, such as where multiple generic RAMs are instantiated within a common parent instance and both need to be simultaneously selected.

Enabling MemoryBIST Mode for a Cluster

Circuit tracing is based on the flat design model, which is an internal, flattened representation of the hierarchical design. The memory cluster must be configured in MemoryBIST mode to analyze the circuit relevant for memory test. The MemoryBIST mode entry sequence must be provided to Tessent MemoryBIST and can be obtained from the documentation for your specific IP.

The subsequent sections describe how the initialization sequence is created and how you can customize it.

Memory Cluster Initialization File Generation

As described in "Implementing the P2L Mapping Flow" the tool automatically generates a dofile which contains the default memory cluster initialization sequence used in the simulation context on the flat model. The default initialization sequence assumes a generic protocol, and constrains the following inputs of the shared bus interface defined in the memory cluster TCD:

- Port function InterfaceReset — defines the Shared Bus Interface Reset port(s)

- Port function BistOn — defines the Shared Bus Interface Request port(s)

- Port function ConfigurationData — defines the Shared Bus Interface Configuration Data port(s)

An example initialization sequence is shown below. The interface reset and request ports are set to their active values. The signals are held for 20 clock cycles, before the interface reset is deactivated. The signals are then held for 50 clock cycles.

```
# Cluster module: CLUSTER

# Enable active-low interface reset port
add_simulation_forces [get_ports nrst] -value 0;

# Enable active-high interface request port
add_simulation_forces \
  [get_pins MbistOn1 \
    -of_instances [get_instances \
    -of_modules [get_modules CLUSTER -use_module_matching_options]]] \
    -value 1;

# Apply 20 system clock cycles
simulate_clock_pulses [get_clocks] -repetitions 20;

# Release interface reset port
add_simulation_forces [get_ports nrst] -value 1;

# Apply 50 system clock cycles
simulate_clock_pulses [get_clocks] -repetitions 50;
```

If your IP provider delivers an electronic file, the memlibGenerate utility will automatically transfer the reset, request, and configuration data ports into the memory cluster TCD. The file may identify additional inputs that must be asserted for the duration of memory test mode. If the memory cluster TCD is automatically created, these signals will also be appropriately constrained in the default memory cluster initialization file. The memlibGenerate utility assumes that the Shared Bus Interface reset is an active low signal.

Memory Cluster Initialization File Modification

The default initialization file created by the set_memory_cluster_library_generation_options command may be incomplete, and the protocol may require further customization based on the cluster core IP documentation. In addition, some of the input ports which need to be driven to achieve MBIST mode, or to initialize/reset the cluster module, may be unknown. This may occur when the memory cluster TCD is manually created rather than automatically generated by the memlibGenerate utility.

The tool provides the ability for you to edit the initialization file manually to address this situation. As described in "Implementing the P2L Mapping Flow", the file is automatically generated and is automatically sourced when check_design_rules is run. All changes made to the file by the user after the generation will be applied by the tool when executing the cluster initialization sequence.

The initialization file is stored in the current working directory with the following naming convention:

<design_name>_<insertion_id>.memory_cluster_mbist_mode_init

The check_design_rules command automatically applies the initialization file, which contains the cluster initialization sequence for each Shared Bus cluster module in the design. Each

sequence is preceded by the cluster module name and should initialize all Shared Bus interfaces belonging to the particular cluster module.

When you modify the cluster initialization file, it is important that you use design introspection commands, such as get_pins and get_ports, instead of specifying explicit hierarchical paths to pins/ports. This is needed because a quick-synthesis is done as part of the cluster validation procedure and some post-synthesis names may change during the flow. The get_modules -use_module_matching_options command and switch are useful when you are looking for modules whose names were potentially uniquified.

If signals added to the modified initialization file do not correspond to primary inputs or the cluster module itself, they will not be automatically preserved in the flat design model. When this occurs, the tool will issue an error informing you that the pin does not exist in the flat model. You must then run the following command after default initialization file generation (with either set_memory_cluster_library_generation_options or set_memory_cluster_validation_options) and before the check_design_rules command is run:

```
set_attribute_value pin_port_name -name preserve_in_flat -value "yes"
```

Memory Cluster Initialization File Debugging

During the analysis, if MemoryBIST mode was not entered successfully, then one of the following situations will happen, depending on the specific IP:

- A MemoryBIST acknowledge signal error (the signal being "X" or an inactive state) might be reported by the tool if your IP defines one.

- No physical memory may be accessible from the Shared Bus Interface.

- All physical memories are accessible, regardless of the selection values specified using the Shared Bus Interface.

Typically, the Shared Bus Interface of your specific IP implements the MBIST acknowledge signal and Tessent MemoryBIST will examine its value after the initialization sequence is simulated during check_design_rules execution. MemoryBIST mode will not be entered if the MBIST acknowledge signal is 'X' or is at an inactive polarity and the tool will issue an error message. The error message includes the command to invoke Tessent Visualizer and display the tracing path. The displayed path is from the InterfaceReset port to the MBIST acknowledge port. The corresponding simulation values from the current simulation context are also displayed, as seen in the figure below.

Figure 6-12. MemoryBIST Request to Acknowledge Path With Incorrectly Initialized Register
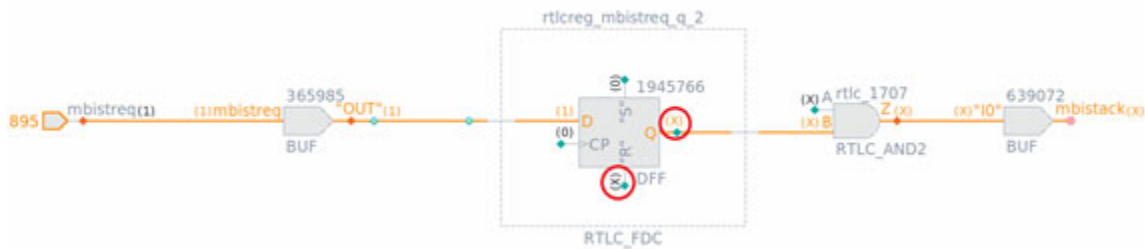
The initialization sequence can be debugged interactively by:

1. Adding a new simulation context (refer to the add_simulation_context command).

2. Sourcing the modified initialization file from your working directory.

3. Observing the influence of the modified sequence on the MBIST acknowledge signal in Tessent Visualizer.

The following steps may also be helpful in the cases where Tessent MemoryBIST reports an MBIST acknowledge signal error:

1. Inspect the memory cluster initialization file for completeness. Typically, the following signals are required to be constrained by the sequence:

   o A pin/port with Port Function BistOn — Enables MemoryBIST mode and disables some functional logic.

   o A pin/port with Port Function InterfaceReset — Specifies the signal that is used to reset the Shared Bus Interface.

   o A pin/port with Port Function ConfigurationData — Specifies the signal that is used to configure access to the logical memories.

   o Other signals that are documented by your IP provider that must maintain the same state through all MemoryBIST testing.

   If some of these signals are not constrained by the initialization sequence, review the documentation of your specific IP to confirm the required initialization sequence.

2. If the initialization sequence seems to contain all the typically required signals, you can pinpoint the source of the MBIST acknowledge signal becoming an 'X' or inactive value by using Tessent Visualizer. For further information, refer to the "Using Tessent Visualizer to Debug Design Issues" topic in the *Tessent Shell User's Manual*.

3. Investigate if there is additional logic that is not part of the original IP from your provider that is connected to your memory cluster. The logic may also need to be constrained in the memory cluster initialization file. To test for this condition, you can

try to run check_design_rules with Shared Bus Learning disabled. If check_design_rules reports any blocking conditions for memory clock tracing, these same blocking conditions will be present when performing Shared Bus Learning. These additional blocking conditions will need to be handled by adding the proper constraints in the memory cluster initialization file.

_____ **Note** _____

Siemens EDA recommends running the Shared Bus Learning features on a Shared Bus memory cluster in isolation, where the memory cluster is instantiated within the current design with no other logic around the memory cluster. However, if you have run the Learning features on a Shared Bus memory cluster in the context of a design with functional logic surrounding the memory cluster, then you may need to edit the initialization sequence to ensure clocks and control signals propagate to the memory cluster. Running the Learning features in the context of a design is discouraged as it can unnecessarily complicate debug efforts.

Shared Bus Learning Assumptions and Limitations

The following assumptions and limitations apply when implementing the Shared Bus Learning automation flow:

1. The MemoryInstanceName property must be added manually to either the memory cluster TCD or logical TCD files if they require the use of the %VARIABLE% notation (as defined in Table 6-1) in the MemoryInstanceName property to uniquely specify the physical memory instance. Refer to Table 6-15 and the related text for an example.

2. If a logical memory template is used by more than one logical memory, as shown by LM_0 and LM3 in the figure below, and the MemoryInstanceName property is needed, editing is required to uniquify the instances.
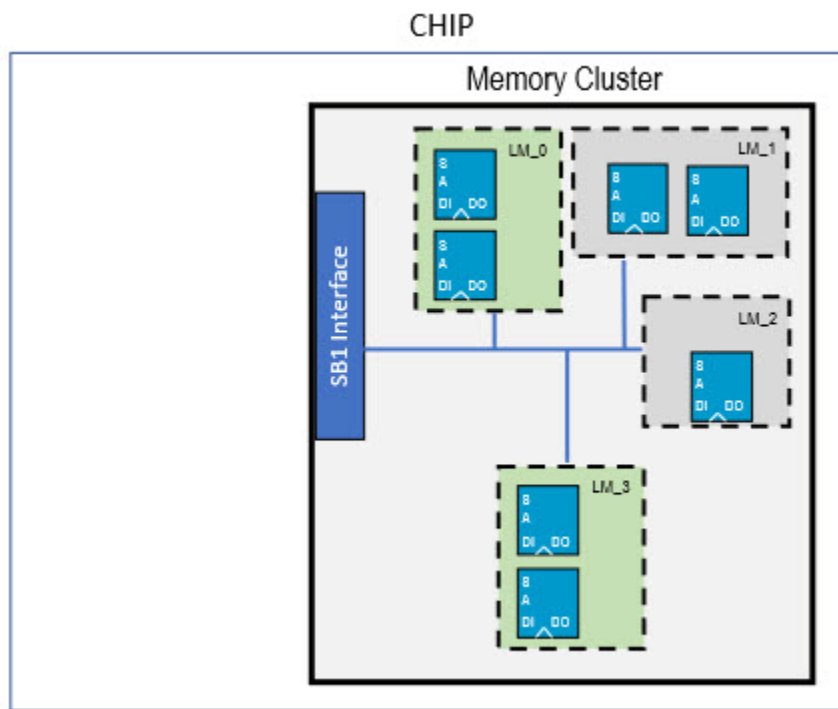
Referring to the figure above, the instance names for each physical memory in LM_0 and LM_3 relative to the memory cluster module are:

```
LM_0_inst/upper_mem_inst
LM_0_inst/lower_mem_inst
LM_3_inst/upper_mem_inst
LM_3_inst/lower_mem_inst
```

Using this as an example, two implementation solutions are illustrated:

a. Solution 1 — Manually update the MemoryInstance property in the memory cluster TCD and logical memory TCD files after running P2L mapping automation.

The relevant portion of the input memory cluster TCD file contains:

```
LogicalMemoryToInterfaceMapping(LM_0) {
  MemoryTemplate : LM_64x8;
  ...
}
...
LogicalMemoryToInterfaceMapping(LM_3) {
  MemoryTemplate : LM_64x8;
  ...
}
```

The relevant portion of the input logical memory TCD contains:

```
MemoryTemplate(LM_64x8) {
  <PORT_WRAPPER_DEFINITIONS>
  ...
  <ADDRESS_COUNTER_WRAPPER>
}
```

Note that P2L mapping automation does not add the MemoryInstanceName property to either the memory cluster TCD or logical memory TCD. The user must manually update these files. The relevant portion of the final memory cluster TCD file after manual edits (shown in red):

```
LogicalMemoryToInterfaceMapping(LM_0) {
  MemoryTemplate : LM_64x8;
  ...
  MemoryInstanceName : LM_0_inst;
  <PIN_MAPPINGS_WRAPPER>
}
...
LogicalMemoryToInterfaceMapping(LM_3) {
  MemoryTemplate : LM_64x8;
  ...
  MemoryInstanceName : LM_3_inst;
<PIN_MAPPINGS_WRAPPER>
}
```

The relevant portion of the final logical memory TCD file after running P2L automation (green highlight additions) and adding manual edits (red highlight additions):

```
MemoryTemplate(LM_64x8) {
  <PORT_WRAPPER_DEFINITIONS>
  ...
  <ADDRESS_COUNTER_WRAPPER>
  PhysicalToLogicalMapping(UPPER_MEM) {
    MemoryTemplate : <PHYSICAL_TEMPLATE_NAME>;
    MemoryInstanceName : upper_mem_inst;
    <PIN_MAPPINGS_WRAPPER>
  }
  PhysicalToLogicalMapping(LOWER_MEM) {
    MemoryTemplate : <PHYSICAL_TEMPLATE_NAME>;
    MemoryInstanceName : lower_mem_inst;
    <PIN_MAPPINGS_WRAPPER>
  }
}
```

b. Solution 2 — Manually uniquify the logical memory templates reference by LM_0 and LM_3. This edit must be completed in the memory cluster TCD and logical memory TCD before using P2L mapping automation.

The relevant portion of the input memory cluster TCD file after manual edits (shown in red) contains:

```
LogicalMemoryToInterfaceMapping(LM_0) {
  MemoryTemplate : LM_64x8;
  ...
}
...
LogicalMemoryToInterfaceMapping(LM_3) {
  MemoryTemplate : LM_64x8_0;
  ...
}
```

The relevant portion of the input logical memory TCD after manual edits (shown in red) contains:

```
MemoryTemplate(LM_64x8) {
  <PORT_WRAPPER_DEFINITIONS>
  ...
  <ADDRESS_COUNTER_WRAPPER>
}
MemoryTemplate(LM_64x8_0) {
  <PORT_WRAPPER_DEFINITIONS>
  ...
  <ADDRESS_COUNTER_WRAPPER>
}
```

Note that the memory cluster TCD and logical memory TCD do not require editing after P2L mapping automation. The relevant portion of the output logical memory TCD after running P2L mapping automation (green highlight additions):

```
MemoryTemplate(LM_64x8) {
  <PORT_WRAPPER_DEFINITIONS>
  ...
  <ADDRESS_COUNTER_WRAPPER>
  PhysicalToLogicalMapping(UPPER_MEM) {
    MemoryTemplate : <PHYSICAL_TEMPLATE_NAME>;
    MemoryInstanceName : LM_0_inst/upper_mem_inst;
    <PIN_MAPPINGS_WRAPPER>
  }
  PhysicalToLogicalMapping(LOWER_MEM) {
    MemoryTemplate : <PHYSICAL_TEMPLATE_NAME>;
    MemoryInstanceName : LM_0_inst/lower_mem_inst;
    <PIN_MAPPINGS_WRAPPER>
  }
}
MemoryTemplate(LM_64x8_0) {
  <PORT_WRAPPER_DEFINITIONS>
  ...
  <ADDRESS_COUNTER_WRAPPER>
  PhysicalToLogicalMapping(UPPER_MEM) {
    MemoryTemplate : <PHYSICAL_TEMPLATE_NAME>;
    MemoryInstanceName : LM_3_inst/upper_mem_inst;
    <PIN_MAPPINGS_WRAPPER>
  }
  PhysicalToLogicalMapping(LOWER_MEM) {
    MemoryTemplate : <PHYSICAL_TEMPLATE_NAME>;
    MemoryInstanceName : LM_3_inst/lower_mem_inst;
    <PIN_MAPPINGS_WRAPPER>
  }
}
```

_____ **Note** _____

Solution 2 will result in a larger area overhead than Solution 1 requires.
_____

3. Memory clusters that contain multi-port memories are not currently supported.

The following assumption and limitation applies when using the shared bus learning validation feature at the logical level.

1. It is assumed that each selection code activates only one logical memory.

Design Loading

As with the standard memory BIST flow, design loading is the first step in the Tessent MemoryBIST insertion using Tessent Shell. The step consists of setting the correct context, reading libraries, reading the design, and elaborating the design.

To utilize the Shared Bus feature, the MemoryCluster, logical and physical memory core library files must be loaded. An example is provided below.

Example

```
set_context dft -rtl
read_core_descriptions {../data/MEM/CLUSTER.tcd_mem_cluster_lib \
                        ../data/MEM/Logical.lvlib \
                        ../data/MEM/SYNC_1RW_16x8.tcd_mem_lib \
                        ../data/MEM/SYNC_1RW_32x4.tcd_mem_lib
                        }
set_design_sources -format verilog -y ../data/MEM -extension vb
read_verilog {../data/RTL/CORE.vb ../data/RTL/CLUSTER.vb}
set_current_design CORE
```

For more details on design loading, please refer to Chapter 3, Planning and Inserting MemoryBIST.

Specify and Verify DFT Requirements

As with the standard memory BIST flow, you must specify the DFT specification requirements with the set_dft_specification_requirements command. This enables the DRC specific to memory BIST and instructs the create_dft_specification command to include the MemoryBist or MemoryBisr wrappers.

Example

The following example shows how the memory BIST DFT specification requirements are specified and how the design level is defined at the chip level.

```
set_dft_specification_requirements -memory_test on
set_design_level chip
```

There is the additional DFTSpecification property (DftSpecification()MemoryBist/Controller()/ MemoryCluster()/memory_access_level) and associated DefaultsSpecification property (DefaultsSpecification/DftSpecification/MemoryBist/MemoryClusterOptions/ memory_access_level), which is auto by default, that affects the memory bist controller configuration. A summary of the values for the memory_access_level property is as follows:

- **logical —** The memory BIST controller operates at the logical memory level which means that a virtual memory (emulating a memory from the controller viewpoint) is created for the logical memory. The physical composition of the logical memory is irrelevant; all physical memories that form the logical memory are tested in one step. A single memory BIST interface is created for each logical memory.

___ **Note** _____

The tool reports an error when the memory access level is set to "logical", and a logical memory TCD does not define a RedundancyAnalysis wrapper while a physical memory TCD within the logical memory does.
_____

- **physical —** The memory BIST controller operates at the physical memory level, with a virtual memory being created for each individual physical memory that forms the logical array. Each physical memory has its own memory BIST interface and a dedicated controller step associated with the physical memory access code, if it is defined in the MemoryGroupAddressDecoding wrapper in the logical MemoryTemplate.

  In case a RedundancyAnalysis wrapper is present in the logical memory core library, its content is disregarded when option Physical is chosen.

- **auto —** Enabling this option balances the pros and cons of the two options described above. Virtual memories are generated for the physical memories when the following conditions are met, otherwise a virtual memory is created for the logical memory:

  o At least one PhysicalToLogicalMapping wrapper exists in the logical memory core library that defines the relationship between physical and logical memory ports,

  o The logical memory has no BIRA logic associated with it - no RedundancyAnalysis wrapper is present in that logical memory core library,

___ **Note** _____

The default memory BIST controller configuration generated for a share bus cluster module is different for the LogicVision design flow (equivalent to memory_access_level: logical) than in the Tessent Shell MemoryBIST flow (memory_access_level: auto). To create a MemoryCluster memory BIST controller with the same configuration as the LogicVision design flow, the memory_access_level needs to be set to logical.
_____

For more details on design loading, please refer to Chapter 3, Planning and Inserting MemoryBIST.

Create DFT Specification

There no difference in usage when compared to standard memory BIST flow, an example is given below.

```
ANALYSIS> create_dft_specification -replace
//
//  Begin creation of DftSpecification(CORE,rtl)
//    Creation of RtlCells wrapper
//    Creation of IjtagNetwork wrapper
//    Creation of MemoryBist wrapper
//
//  Done  creation of DftSpecification(CORE,rtl)
//
/DftSpecification(CORE,rtl)
```

Below is an example DFTSpecification produced from create_dft_specification for a design block with a single memory cluster module instantiation.

```
DftSpecification(CORE,rtl) {
  IjtagNetwork {
    HostScanInterface(ijtag) {
      Sib(sti) {
        Attributes {
          tessent_dft_function : scan_tested_instrument_host;
        }
        Sib(mbist) {
        }
      }
    }
  }
  MemoryBist {
    ijtag_host_interface : Sib(mbist);
    Controller(c1) {
      clock_domain_label : clk;
      AdvancedOptions {
        observation_xor_size : off;
      }
      MemoryCluster(cluster1) {
        instance_name : CLUSTERInst;
      }
    }
  }
}
```

Once the DftSpecification has been created, you can use report_memory_cluster_configuration to report the cluster configuration for cluster modules in you design. An example is given below.

```
ANALYSIS> report_memory_cluster_configuration

// Memory cluster: CLUSTER, expansion level: auto
// ==============================================
// ----  --------  -------  --------  ---------  -------  --------  ----------------
// Step  Access    Logical  Physical  Cluster    Logical  Physical  Memory
//       level     access   access    interface  memory   memory    instance
//                 code     code      id         id       id        id
// ----  --------  -------  --------  ---------  -------  --------  ----------------
// 0     physical  3'b001   -         I1         LM_0     MEM_0     I1_LM_0_MEM_0
//                                    I1         LM_0     MEM_1     I1_LM_0_MEM_1
// 1     physical  3'b010   -         I1         LM_1     MEM_0     I1_LM_1_MEM_0
// 2     physical  3'b011   1'b0      I1         LM_2     MEM_b0_L  I1_LM_2_MEM_b0_L
//                                    I1         LM_2     MEM_b0_M  I1_LM_2_MEM_b0_M
// 3     physical  3'b011   1'b1      I1         LM_2     MEM_b1_L  I1_LM_2_MEM_b1_L
//                                    I1         LM_2     MEM_b1_M  I1_LM_2_MEM_b1_M
// 4     physical  3'b100   1'b0      I1         LM_3     MEM_0     I1_LM_3_MEM_0
// 5     physical  3'b100   1'b1      I1         LM_3     MEM_1     I1_LM_3_MEM_1
```

Process DFT Specification

There is no difference in usage when compared to the standard memory BIST flow, and an example is shown below.

```
ANALYSIS> process_dft_specification
//
//  Begin processing of /DftSpecification(CORE,rtl)
//     --- IP generation phase ---
//    Validation of IjtagNetwork
//    Validation of MemoryBist
//    Processing of RtlCells
//      Generating Verilog RTL Cells
//        Verilog RTL : ./tsdb_outdir/instruments/CORE_rtl_cells.instrument/
CORE_rtl_tessent_and2.v
//        Verilog RTL : ./tsdb_outdir/instruments/CORE_rtl_cells.instrument/
CORE_rtl_tessent_clk_and2.v
//        Verilog RTL : ./tsdb_outdir/instruments/CORE_rtl_cells.instrument/
CORE_rtl_tessent_or2.v
//        Verilog RTL : ./tsdb_outdir/instruments/CORE_rtl_cells.instrument/
CORE_rtl_tessent_clk_or2.v
//        Verilog RTL : ./tsdb_outdir/instruments/CORE_rtl_cells.instrument/
CORE_rtl_tessent_buf.v
//        Verilog RTL : ./tsdb_outdir/instruments/CORE_rtl_cells.instrument/
CORE_rtl_tessent_clk_buf.v
//        Verilog RTL : ./tsdb_outdir/instruments/CORE_rtl_cells.instrument/
CORE_rtl_tessent_inv.v
//        Verilog RTL : ./tsdb_outdir/instruments/CORE_rtl_cells.instrument/
CORE_rtl_tessent_clk_inv.v
//        Verilog RTL : ./tsdb_outdir/instruments/CORE_rtl_cells.instrument/
CORE_rtl_tessent_mux2.v
//        Verilog RTL : ./tsdb_outdir/instruments/CORE_rtl_cells.instrument/
CORE_rtl_tessent_clk_mux2.v
//
//      Loading the generated RTL verilog files (2) to enable instantiating the contained
modules
//      into the design.
//
//      Loading the generated structural verilog files (8) to enable instantiating the
contained modules
//      into the design.
//    Processing of IjtagNetwork
//      Generating design files for IJTAG SIB module CORE_rtl_tessent_sib_1
//        Verilog RTL : ./tsdb_outdir/instruments/CORE_rtl_ijtag.instrument/
CORE_rtl_tessent_sib_1.v
//        IJTAG ICL   : ./tsdb_outdir/instruments/CORE_rtl_ijtag.instrument/
CORE_rtl_tessent_sib_1.icl
//        Tcd Scan    : ./tsdb_outdir/instruments/CORE_rtl_ijtag.instrument/
CORE_rtl_tessent_sib_1.tcd_scan
//      Generating design files for IJTAG SIB module CORE_rtl_tessent_sib_2
//        Verilog RTL : ./tsdb_outdir/instruments/CORE_rtl_ijtag.instrument/
CORE_rtl_tessent_sib_2.v
//        IJTAG ICL   : ./tsdb_outdir/instruments/CORE_rtl_ijtag.instrument/
CORE_rtl_tessent_sib_2.icl
//
//      Loading the generated RTL verilog files (2) to enable instantiating the contained
modules
//      into the design.
//    Processing of MemoryBist
//      Generating the Shared Bus memory cluster synthesis models.
//      Generating the cluster initialization iProcs for the Shared Bus memory clusters.
//      Generating design files for MemoryBist Controller(c1)
//      Generating design files for Bist Access Port
```

```
//       Generating design files for MemoryBist controller assembly
//     --- Instrument insertion phase ---
//     Inserting instruments of type 'ijtag'
//     Inserting instruments of type 'memory_bist'
//
//     Writing out modified source design in ./tsdb_outdir/dft_inserted_designs/
CORE_rtl.dft_inserted_design
//     Writing out specification in ./tsdb_outdir/dft_inserted_designs/CORE_rtl.dft_spec
//
//  Done  processing of DftSpecification(CORE,rtl)
//
/DftSpecification(CORE,rtl)
```

Extract ICL

The extract_icl command prepares the current design for pattern generation, as is done in the standard memory BIST flow, by finding all modules (both Tessent instruments and non-Siemens EDA instruments) with their associated ICL modules and, if no DRC violations are detected, creates the ICL for the current design.

The root of the design was specified with the set_current_design command during design elaboration in the Design Loading step. The Create Patterns Specification and Process Patterns Specification steps use the ICL that was created for the root of the design. You can use the open_visualizer command to debug ICL extraction DRC violations. Refer to the "Debugging DRC Violations with Tessent Visualizer" section in the Tessent IJTAG User's Manual.

Create Patterns Specification

The Create Patterns Specification step creates the default patterns specification in the same manner at the standard memory BIST flow. The patterns specification is a configuration file that tells you what tests are created using process_patterns_specification. You can edit or configure the default patterns specification to generate the patterns specification you want.

Below is an example PatternsSpecification produced from create_patterns_specification for a design block with a single memory cluster module instantiation.

```
PatternsSpecification(CORE,rtl,signoff) {
  Patterns(ICLNetwork) {
    ICLNetworkVerify(CORE) {
    }
  }
  Patterns(MemoryBist_P1) {
    ProcedureStep(initialize_memory_cluster) {

iCall(CORE_rtl_tessent_mbist_c1_shared_bus_assembly_inst_CORE_rtl_tessent
_mbist_c1_shared_bus_glue_logic_inst.initialize_memory_cluster) {
      }
    }
    ClockPeriods {
      clk : 1.25ns;
    }
    TestStep(run_time_prog) {
      MemoryBist {
        run_mode : run_time_prog;
        reduced_address_count : on;

Controller(CORE_rtl_tessent_mbist_c1_shared_bus_assembly_inst_CORE_rtl_te
ssent_mbist_c1_controller_inst) {
          DiagnosisOptions {
            compare_go : on;
            compare_go_id : on;
          }
        }
      }
    }
  }
}
```

Notice there is reference to an iCall to initialize the cluster module. This is covered in the next section.

By default, the memory BIST controller tests all memories in a cluster module in a single TestStep in the order according to the controller steps enumerated by the report_memory_cluster_configuration command.

Depending on the selected memory access level, logical or physical memories can be tested individually using the freeze_step property in the PatternsSpecification.

An example is given below where only the first scheduled memory is tested. The configuration_data value can also be specified in the MemoryClusterOptions wrapper to override the default ConfigurationData value if defined in the LogicalMemoryToInterfaceMapping wrapper of the memory cluster library, which defaults to the value auto if unspecified in the controller wrapper.

```
PatternsSpecification(CORE,rtl,signoff) {
  Patterns(ICLNetwork) {
    ICLNetworkVerify(CORE) {
    }
  }
  Patterns(MemoryBist_P1) {
    ProcedureStep(initialize_memory_cluster) {

iCall(CORE_rtl_tessent_mbist_c1_shared_bus_assembly_inst_CORE_rtl_tessent
_mbist_c1_shared_bus_glue_logic_inst.initialize_memory_cluster) {
      }
    }
    ClockPeriods {
      clk : 1.25ns;
    }
    TestStep(run_time_prog) {
      MemoryBist {
        run_mode : run_time_prog;
        reduced_address_count : on;

Controller(CORE_rtl_tessent_mbist_c1_shared_bus_assembly_inst_CORE_rtl_te
ssent_mbist_c1_controller_inst) {
          AdvancedOptions +{
            freeze_step : 0;
            MemoryClusterOptions +{
              configuration_data(I1) : 2'b00;
            }
          }
          DiagnosisOptions {
            compare_go : on;
            compare_go_id : on;
          }
        }
      }
    }
  }
}
```

Process Patterns Specification

There is no difference in usage when compared to the standard memory BIST flow and an example is shown below.

```
SETUP> process_patterns_specification
//
//   Begin processing of /PatternsSpecification(CORE,rtl,signoff)
//
//      Processing of /PatternsSpecification(CORE,rtl,signoff)/
Patterns(ICLNetwork)
//
//         Creation of pattern 'ICLNetwork'
//           Solving ICLNetworkVerify(CORE)
//
//         Writing pattern file './tsdb_outdir/patterns/
CORE_rtl.patterns_signoff/ICLNetwork.v'
//
//      Processing of /PatternsSpecification(CORE,rtl,signoff)/
Patterns(MemoryBist_P1)
//         Processing of ProcedureStep(initialize_memory_cluster)
//         Processing of TestStep(run_time_prog) instrument 'memory_bist'
//
//         Creation of pattern 'MemoryBist_P1'
//           Solving ProcedureStep(initialize_memory_cluster)
//           Solving TestStep(run_time_prog)
//
//         Writing pattern file './tsdb_outdir/patterns/
CORE_rtl.patterns_signoff/MemoryBist_P1.v'
//         Generating design files for Monitor module
MemoryBist_P1_CLOCK_MONITOR
//           Verilog module : ./tsdb_outdir/patterns/
CORE_rtl.patterns_signoff/MemoryBist_P1_CLOCK_MONITOR.v
//         Writing simulation data dictionary file './tsdb_outdir/patterns/
CORE_rtl.patterns_signoff/simulation.data_dictionary'
//
//   Done  processing of /PatternsSpecification(CORE,rtl,signoff)
//
//   Writing configuration data file './tsdb_outdir/patterns/
CORE_rtl.patterns_spec_signoff'.
```

During process_patterns_specification, an initialization sequence (iProc) for the Shared Bus interface is created, which is generally needed if the Shared Bus memory cluster module implements interface ports that use the InterfaceReset or BistOn port functions. The Shared Bus memory cluster initialization sequence is driven by TDR bits and is performed before launching the memory BIST controller. The initialization sequence is referenced in the PatternsSpecification/Patterns/ProcedureStep with an iCall. If the initialization sequence needs modification, it can be found at the following location:

```
tsdb_outdir/instruments/<design_name>_<design_id>_mbist.instrument/
<design_name>_<design_id>_tessent_mbist_<controller_id>_shared_bus_glue_logic.pdl
```

An example of the initialization iProc is given below. The Shared Bus cluster module in this example has an interface reset and does not have a BIST enable (a port with a BistOn port function)

```
iProcsForModule CORE_rtl_tessent_mbist_c1_shared_bus_glue_logic
iProc initialize_memory_cluster {} {
  iWrite nrst_toCluster 0b1
  iApply
  iNote "Activating InterfaceReset port(s) of the cluster module CLUSTER
via outputs of the glue logic module instance [get_icl_scope -iCall]"
  iWrite nrst_toCluster 0b0
  iApply
  iRunLoop 16
  iNote "Deactivating InterfaceReset port(s) of the cluster module CLUSTER
via outputs of the glue logic module instance [get_icl_scope -iCall]"
  iWrite nrst_toCluster 0b1
  iApply
  iRunLoop 16
}
```

Run and Check Testbench Simulations

There is no difference in usage when compared to the standard memory BIST flow and the commands are shown below.

```
run_testbench_simulations
check_testbench_simulations
```

Test Logic Synthesis

There is no difference in usage when compared to the standard memory BIST flow and the command shown below.

```
run_synthesis
```

Handling MemoryCluster Modules With Repairable Memories

This section explains the considerations, methods and prerequisites for built-in repair analysis (BIRA) and built-in self-repair (BISR) generation and insertion for memories accessed through a Shared Bus interface.

BIRA and BISR Generation for a Memory Cluster Module

Tessent MemoryBIST is able to insert the BIRA and BISR logic for memories accessed through a Shared Bus interface. By default, BIRA logic is placed outside the Shared Bus memory cluster module and the BISR logic is placed inside the Shared Bus memory cluster module near the memory to be repaired.

If the memory access level is set to "logical", then for logical memories that consist of more than one physical memory, you must have a one-to-one mapping between logical and physical memories to achieve maximum repair resolution. By default, the memory_access_level in the MemoryClusterOptions wrapper is set to auto, which automatically creates for each individual physical memory its own memory BIST interface, enabling direct access at the Shared Bus cluster module's level.

_____ **Note** _____

The tool reports an error when the memory access level is set to "logical", and a logical memory TCD does not define a RedundancyAnalysis wrapper while a physical memory TCD within the logical memory does.

_____

The BIRA logic is instantiated inside the memory BIST controller module as in a normal design. However, two instances of the BISR registers for each physical memory are inserted inside the design. The first BISR register instance is located near the memory emulation logic and captures the BIRA fuse information. The second BISR register instance is located near the real memory instance inside the memory cluster module and drives the memory repair ports. This means that the core logic must be changed to support the Tessent MemoryBIST repair feature. Figure 6-13 provides an overview of a design after BIRA and BISR insertion. The BIRA and BISR logic are shown in salmon.

Figure 6-13. Design Overview of a Memory Cluster Module With BISR
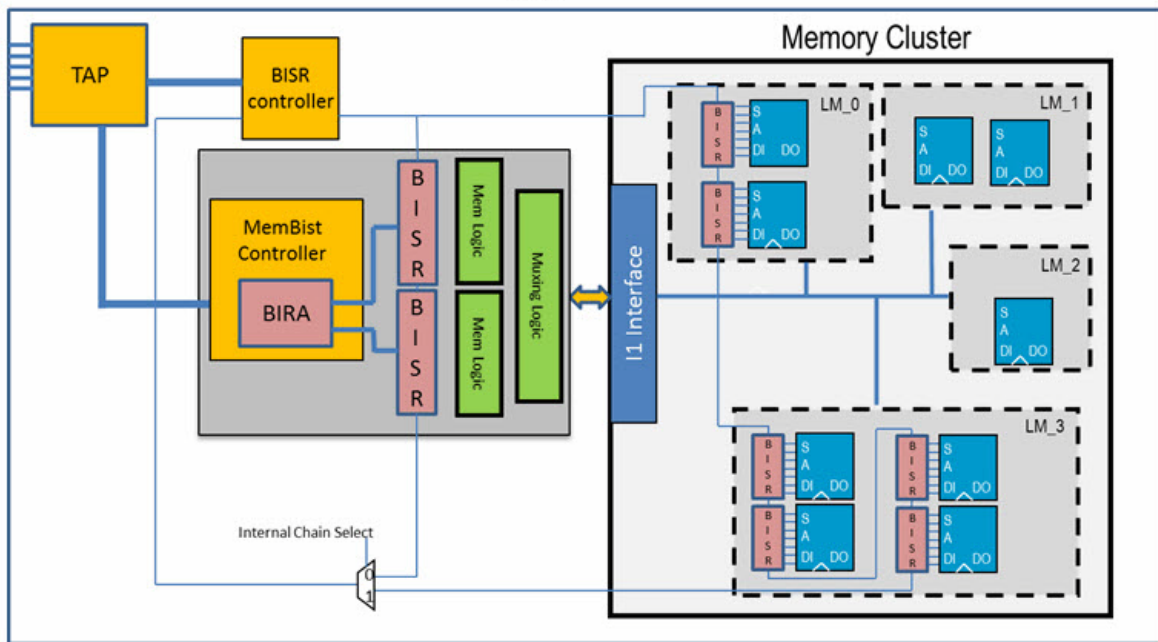
> **Note**
>
> Logical memories LM_1 and LM_2 do not contain memories with redundancy. Therefore, those memories do not have a BISR register.

The BISR registers are connected serially and tied to a multiplexer that selects the BISR chain to scan out. The BISR controller controls the multiplexer select signal. The external BISR registers (located outside of the memory cluster module) are selected when the BISR controller performs the BIRA-to-BISR transfer. Once this transfer is complete, the BISR chain is rotated, and the values that are captured by the external BISR registers are copied to the corresponding BISR registers inside the Shared Bus memory cluster module that drive the memory repair ports. This process is identical to the existing BIRA and BISR flow and does not require additional test steps to perform the memory BIST pre-repair, BISR programming, or memory BIST post-repair.

Design and Library File Prerequisites for BIRA and BISR

The following sections describe how to prepare the design and memory library files so that the BIRA and BISR hardware can be inserted into the design.

Functional Design Preparation for BIRA and BISR

The memory cluster module does not require any design changes in preparation for BISR and BIRA hardware insertion.

Memory Cluster Library File Preparation for BIRA and BISR

With memory_access_level in the MemoryClusterOptions wrapper set to auto or physical (auto is the default), access is automatically created for each individual physical memory, with its own memory BIST interface, enabling direct access of the physical memory at the Shared Bus cluster module's level.

When repairable memories are integrated in the Shared Bus memory cluster, their hierarchical instance paths must be identified. The MemoryInstanceName property in the Shared Bus cluster and logical memory TCD must be populated with the memory instance path relative to the Shared Bus memory cluster module.

Figure 6-14. Four Physical Memories With Redundancy
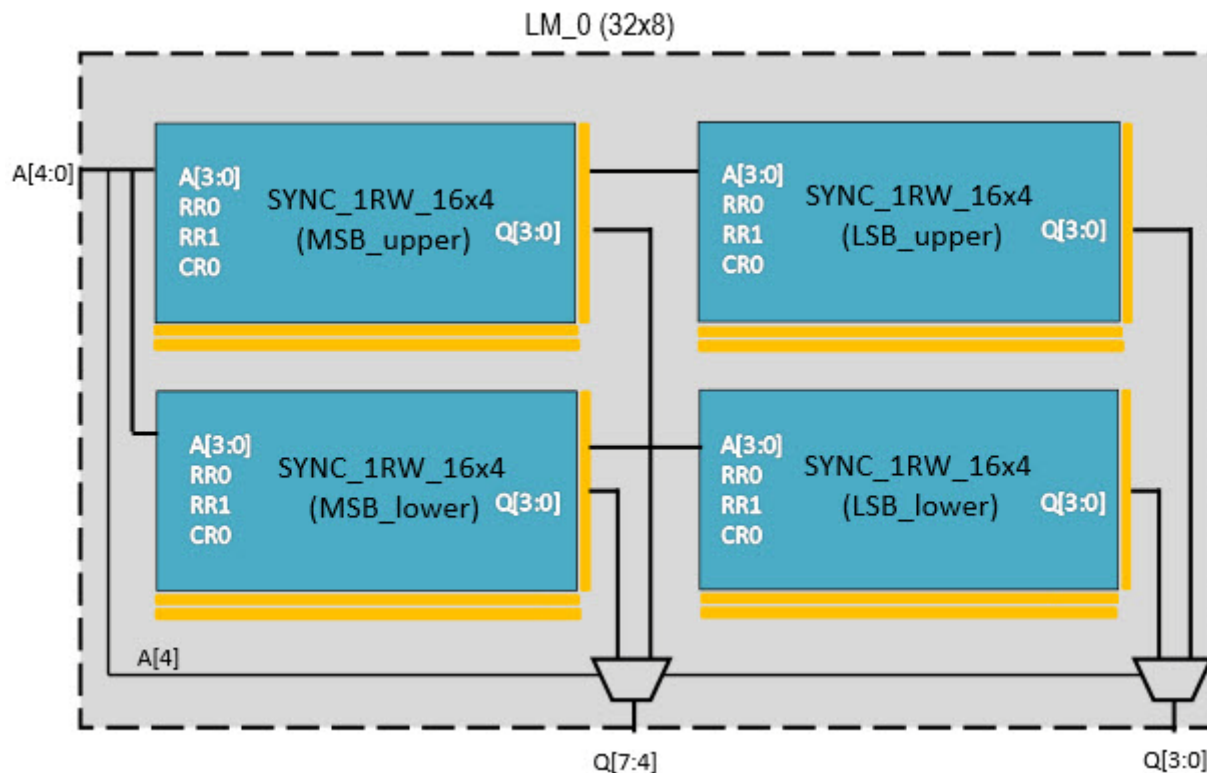
Figure 6-14 shows logical memory LM_0 implemented with 4 physical memories with redundancy. The identical physical memories are instantiated in a 2 by 2 stacking configuration. For simplicity, the figure does not show all connections. The RR0 and RR1 ports on the physical memories enable redundant row elements (shown with gold-color horizontal bars), and the CR0 port enables the redundant column element (shown with gold-color vertical bars). Assume that LM_0 is instantiated within the Shared Bus memory cluster module. The instance path to LM_0, relative to the Shared Bus memory cluster, is populated in the Shared Bus memory cluster TCD as shown below:

```
MemoryClusterTemplate (CLUSTER) {
    LogicalMemoryToInterfaceMapping(LM_0) {
        MemoryInstanceName: LM_0_inst;
    }
}
```

The instance path for the physical memories is relative to its logical memory. The instance paths are populated in the logical memory TCD of LM_0 as shown below. The physical memory instance paths are formed by concatenating the definitions from the Shared Bus memory cluster TCD and the logical memory TCD.

```
MemoryTemplate(LM_0) {
    PhysicalToLogicalMapping(MSB_lower) {
        MemoryTemplate: SYNC_1RW_16x4;
        MemoryInstanceName: MSB_lower_inst;
    }
    PhysicalToLogicalMapping(LSB_lower) {
        MemoryTemplate: SYNC_1RW_16x4;
        MemoryInstanceName: LSB_lower_inst;
    }
    PhysicalToLogicalMapping(MSB_upper) {
        MemoryTemplate: SYNC_1RW_16x4;
        MemoryInstanceName: MSB_upper_inst;
    }
    PhysicalToLogicalMapping(LSB_upper) {
        MemoryTemplate: SYNC_1RW_16x4;
        MemoryInstanceName: LSB_upper_inst;
    }
}
```

Figure 6-15. Logical Memory Placed Outside of a Shared Bus Memory Cluster
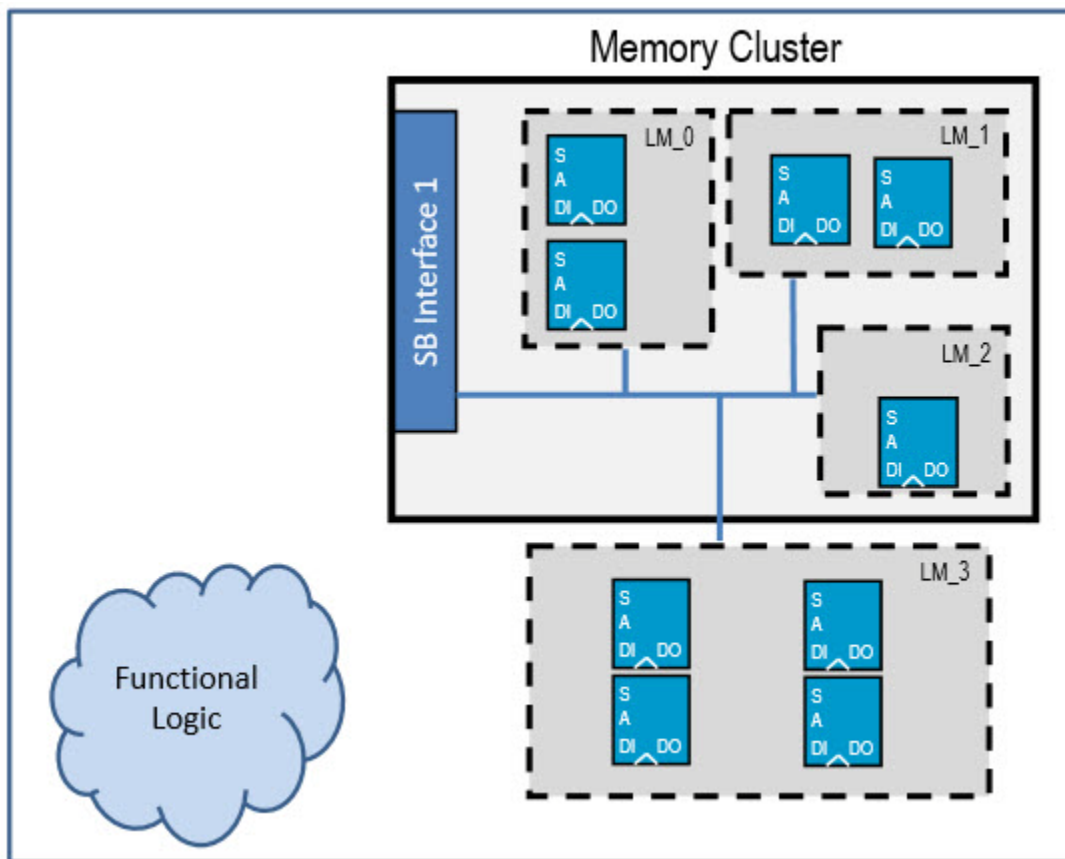
In Figure 6-15, logical memory LM_3 is instantiated at the same hierarchy level as the Shared Bus memory cluster. The instance path to LM_3 would require to traverse from the Shared Bus memory cluster to its parent module, then down into LM_3. The MemoryInstanceName

property accepts one or more "../" path modifiers to create a relative path, which can be used as shown below for this example to reach LM_3 outside the Shared Bus memory cluster:

```
LogicalMemoryToInterfaceMapping(LM_3) {
  MemoryInstanceName: ../LM_3_inst;
}
```

Special variables can be specified within the MemoryInstanceName property to reference portions of the hierarchical instance path to the Shared Bus memory cluster. Using the variables described in Table 6-1, the instance path to logical memory LM_3 can be derived as shown below. The %CLUSTER_PARENT% variable is expanded to the instance path of the module instantiating the Shared Bus memory cluster.

```
LogicalMemoryToInterfaceMapping(LM_3) {
  MemoryInstanceName: %CLUSTER_PARENT%/LM_3_inst;
}
```

Table 6-1 documents the available variables for use inside the Shared Bus memory cluster and logical memory TCD to derive the memory instance. The memory instance paths can be parameterized, based on the design structure, without using absolute instance names. If no %...% variable is used, the final memory instance is computed by appending the Shared Bus memory cluster instance with the MemoryInstanceName property value.

When a %...% variable is used, the memory instance path is no longer relative to the Shared Bus memory cluster. Instead the entire memory instance path must be provided using %...% variables or manual specification of design hierarchies.

The examples in Table 6-1 are computed assuming that the Shared Bus memory cluster instance path is the following:

```
wrapper_inst/core_inst/block_inst/memory_cluster_inst
```

Table 6-1. Shared Bus MemoryInstanceName Variables

Variable	Usage

%TOP%	Provide full instance from the chip-level. Use this method to avoid the concatenation of the Shared Bus memory cluster instance hierarchy. This variable is used to start specifying the memory instance container from the root of the design.
	
	`%TOP%/my_wrapper_inst/mem_inst =>`
	`my_wrapper_inst/mem_inst`

%CLUSTER_PARENT%	First parent of the Shared Bus cluster instance.
	
	`%CLUSTER_PARENT% =>`
	`wrapper_inst/core_inst/block_inst`
	
	`%CLUSTER_PARENT%/mem_wrapper_inst/mem_inst =>`
	`wrapper_inst/core_inst/block_inst/mem_wrapper_inst/`
	`mem_inst`

Table 6-1. Shared Bus MemoryInstanceName Variables (cont.)

Variable	Usage

%CLUSTER_LEAF%	Leaf name of the Shared Bus cluster instance. Typically used with %CLUSTER_PARENT% to denote memory instances located in sibling hierarchy.
	
	`%CLUSTER_LEAF% =>`
	`memory_cluster_inst`
	
	`%CLUSTER_PARENT%/%CLUSTER_LEAF%_tlb_wrapper =>`
	`wrapper_inst/core_inst/block_inst/`
	`memory_cluster_inst_tlb_wrapper`

%CLUSTER_PARENT#%	The #th parent of the Shared Bus cluster instance, where # is an integer value that can be 2 or higher.
	
	`%CLUSTER_PARENT2% => wrapper_inst/core_inst`
	
	`%CLUSTER_PARENT3% => wrapper_inst`

%CLUSTER_PARENT#_LEAF%	Leaf name of the #th parent of the Shared Bus cluster instance, where # is an integer value and can be 2 or higher.
	
	`%CLUSTER_PARENT2_LEAF% => core_inst`
	
	`%CLUSTER_PARENT3_LEAF% => wrapper_inst`

%LM_ID%	Provides a method to parameterize a physical memory instance path based on a logical memory ID. Refer to "Tcl Processing of MemoryInstanceName" for an example and usage information.

Tcl Processing of MemoryInstanceName

Occasionally, you may have Shared Bus clusters where the instance paths to some physical memories require more complex manipulation than the variables outlined in the prior section can offer. For example, the case when you use a generate loop to instantiate the Shared Bus interface and the physical memory instance paths use the generate loop index.

You can use the MemoryInstanceName property in the cluster and logical memory TCD to encode and perform basic string mapping using a Tcl-based function. The parameterization values shown in Table 6-1, combined with string substitution, provide a method to adapt the physical memory instance path without uniquifying the cluster or logical memory TCD.

The following example describes how you use the %LM_ID% variable for this parameterization. Consider a Shared Bus core named CLUSTER, which has two logical memories with *logical_memory_id*'s of LM_0 and LM_1, and each logical memory references the same logical memory template LM_TEMPLATE. Each logical memory is composed of four

physical memories with *physical_memory_id*'s of PM_0, PM_1, PM_2, and PM_3. The
MemoryClusterTemplate for this example Shared Bus core is:

```
MemoryClusterTemplate(CLUSTER) {
  MemoryBistInterface(I1) {
    LogicalMemoryToInterfaceMapping(LM_0) {
      MemoryTemplate: LM_TEMPLATE;
      MemoryInstanceName: "";
    }
    LogicalMemoryToInterfaceMapping(LM_1) {
      MemoryTemplate: LM_TEMPLATE;
      MemoryInstanceName: "";
    }
  }
}
```

The physical memories of LM_0 have the following instance paths:

- PM_0: hier_0_inst/hier0_**p0**_inst/mem0_inst

- PM_1: hier_0_inst/hier0_**p0**_inst/mem1_inst

- PM_2: hier_0_inst/hier1_**p0**_inst/mem0_inst

- PM_3: hier_0_inst/hier1_**p0**_inst/mem1_inst

The physical memories of LM_1 have the following instance paths:

- PM_0: hier_0_inst/hier0_**p1**_inst/mem0_inst

- PM_1: hier_0_inst/hier0_**p1**_inst/mem1_inst

- PM_2: hier_0_inst/hier1_**p1**_inst/mem0_inst

- PM_3: hier_0_inst/hier1_**p1**_inst/mem1_inst

Note that the memory instance paths are unique for each *logical_memory_id*.

Without using the %LM_ID% variable, you cannot make use of a single MemoryInstanceName
property to describe the instance paths for both logical memories. You would therefore need to
uniquify the common LM_TEMPLATE as shown.

```
MemoryClusterTemplate(CLUSTER) {
  MemoryBistInterface(I1) {
    LogicalMemoryToInterfaceMapping(LM_0) {
      MemoryTemplate: LM_TEMPLATE_0;
      MemoryInstanceName: "";
    }
    LogicalMemoryToInterfaceMapping(LM_1) {
      MemoryTemplate: LM_TEMPLATE_1;
      MemoryInstanceName: "";
    }
  }
}
```

The logical memory templates would then be as follows:

```
MemoryTemplate(LM_TEMPLATE_0) {
  PhysicalToLogicalMapping(PM_0) {
    MemoryInstanceName: hier0_inst/hier0_p0_inst/mem0_inst;
  }
  PhysicalToLogicalMapping(PM_1) {
    MemoryInstanceName: hier0_inst/hier0_p0_inst/mem1_inst;
  }
  PhysicalToLogicalMapping(PM_2) {
    MemoryInstanceName: hier0_inst/hier1_p0_inst/mem0_inst;
  }
  PhysicalToLogicalMapping(PM_3) {
    MemoryInstanceName: hier0_inst/hier1_p0_inst/mem1_inst;
  }
}

MemoryTemplate(LM_TEMPLATE_1) {
  PhysicalToLogicalMapping(PM_0) {
    MemoryInstanceName: hier0_inst/hier0_p1_inst/mem0_inst;
  }
  PhysicalToLogicalMapping(PM_1) {
    MemoryInstanceName: hier0_inst/hier0_p1_inst/mem1_inst;
  }
  PhysicalToLogicalMapping(PM_2) {
    MemoryInstanceName: hier0_inst/hier1_p1_inst/mem0_inst;
  }
  PhysicalToLogicalMapping(PM_3) {
    MemoryInstanceName: hier0_inst/hier1_p1_inst/mem1_inst;
  }
}
```

When you use the %LM_ID% variable, the common logical memory template does not need to be uniquified. The example below adds the ability to modify the MemoryInstanceName, based on any of the variables listed in Table 6-1. It makes use of the Tcl "string map" command and the %LM_ID% variable.

```
MemoryClusterTemplate(CLUSTER) {
  MemoryBistInterface(I1) {
    LogicalMemoryToInterfaceMapping(LM_0) {
      MemoryTemplate: LM_TEMPLATE;
      MemoryInstanceName: "";
    }
    LogicalMemoryToInterfaceMapping(LM_1) {
      MemoryTemplate: LM_TEMPLATE;
      MemoryInstanceName: "";
    }
  }
}
```

```
MemoryTemplate(LM_TEMPLATE) {
  PhysicalToLogicalMapping(PM_0) {
    MemoryInstanceName: "[string map {LM_0 p0 LM_1 p1}\
      {hier0_inst/hier0_%LM_ID%_inst/mem0_inst}]";
  }
  PhysicalToLogicalMapping(PM_1) {
    MemoryInstanceName: "[string map {LM_0 p0 LM_1 p1}\
      {hier0_inst/hier0_%LM_ID%_inst/mem1_inst}]";
  }
  PhysicalToLogicalMapping(PM_2) {
    MemoryInstanceName: "[string map {LM_0 p0 LM_1 p1}\
      {hier0_inst/hier1_%LM_ID%_inst/mem0_inst}]";
  }
  PhysicalToLogicalMapping(PM_3) {
    MemoryInstanceName: "[string map {LM_0 p0 LM_1 p1}\
      {hier0_inst/hier1_%LM_ID%_inst/mem1_inst}]";
  }
}
```

The MemoryInstanceName properties use:

```
"[string map {LM_0 p0 LM_1 p1} {hier0_inst/hier0_%LM_ID%_inst/mem0_inst}]";
```

The "string map" Tcl command is composed of two arguments, each enclosed by "{}". The first argument, shown in red, contains one or more string pairs. The first string in each pair is the "search string", while the second string is the "replacement string". For the example above, the "string map" sequence is:

- When the string "LM_0" is found, replace it with the string "p0".

- When the string "LM_1" is found, replace it with the string "p1".

The second argument, shown in green, is the target string the command operates on. Note the %LM_ID% variable is embedded in the string. The tool replaces the variable by the *logical_memory_id* value from the MemoryBistInterface/LogicalMemoryToInterfaceMapping wrapper for the cluster, which in this case is LM_0 and LM_1. The result from the "string map" command is then used for the MemoryInstanceName property.

Parallel Testing of Multiple Shared Bus Interfaces on a Memory Cluster

This section describes how to perform parallel testing of memories accessed through different Shared Bus interfaces located on a Shared Bus memory cluster module.

Figure 6-16 illustrates a Shared Bus memory cluster implementing two Shared Bus interfaces. The corresponding MemoryCluster TCD specifies two MemoryBistInterface wrappers. Tessent MemoryBIST assigns one controller for the Shared Bus memory cluster. The Shared Bus interfaces are accessed sequentially. For example, logical memories LM_0, LM_1 and LM_2 of interface 1 may be tested before LM_3 of interface 2. At present, multiple Shared Bus interfaces on the same Shared Bus memory cluster cannot be driven in parallel by the same memory BIST controller.

Figure 6-16. Memory Cluster With Two Shared Bus Interfaces
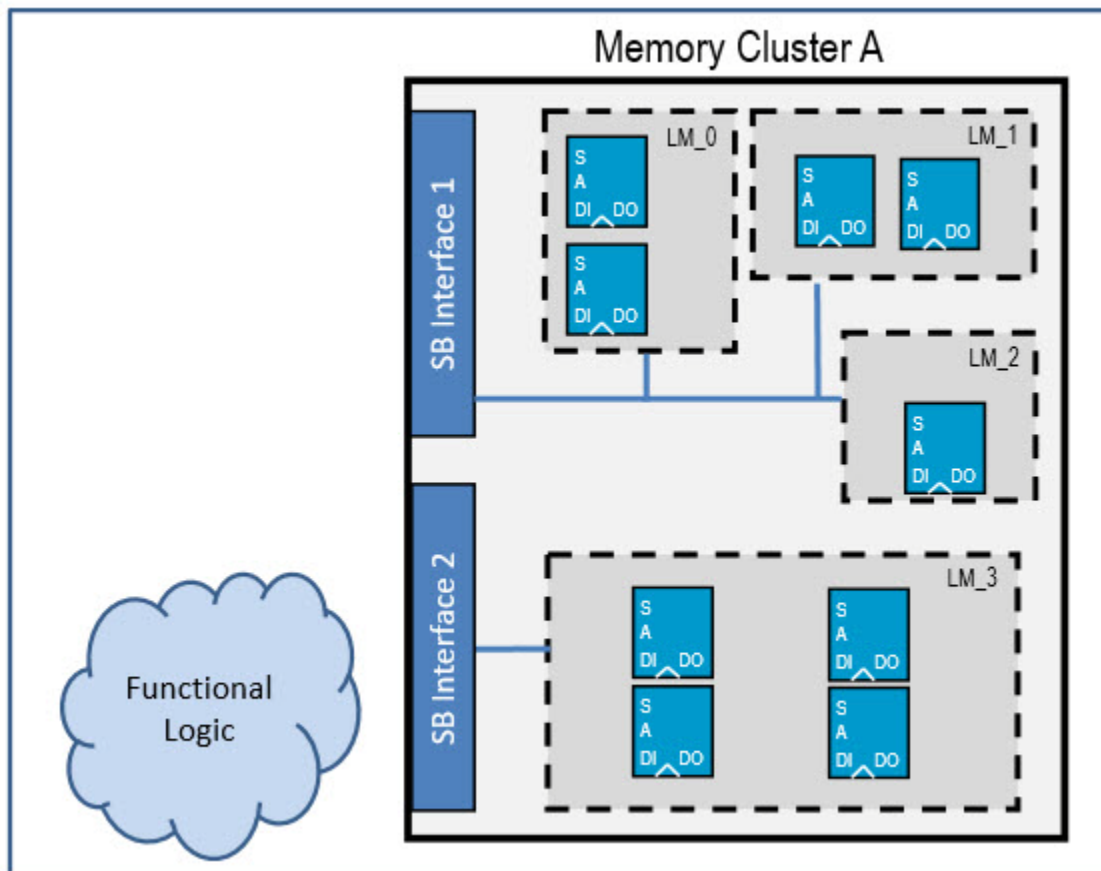
The method to work around the limitation is to model each Shared Bus interface as separate Shared Bus memory clusters. Tessent MemoryBIST dedicates one controller per interface as shown in Figure 6-17 below.

Figure 6-17. Modeling a Memory Cluster With Two Shared Bus Interfaces

Design Modifications

Incorporating the work-around of modeling each Shared Bus interface as separate Shared Bus memory clusters, requires some modification to the design.

For the example of Figure 6-17, create and instantiate a new module next to interface 2. The new module connects to the ports of interface 2 and feeds through the same signals to its boundary. The new module effectively becomes another Shared Bus memory cluster (memory cluster B). One controller drives interface 1 of the original Shared Bus memory cluster A. The second controller attaches to the new Shared Bus memory cluster B, connected to interface 2.

Memory Cluster TCD Modifications

Incorporating the work-around of modeling each Shared Bus interface as separate Shared Bus memory clusters, requires some modification to the Shared Bus memory cluster TCD.

Referring to the example of Figure 6-17, in the MemoryCluster TCD of Shared Bus memory cluster A, only the specification of the MemoryBistInterface wrapper for interface 1 is kept.

Shared Bus memory cluster A is modeled to consist of logical memories LM_0, LM_1, and LM_2 only. Similarly, a MemoryCluster TCD is created for Shared Bus memory cluster B. It specifies the MemoryBistInterface wrapper for interface 2 and contains logical memory LM_3.

```
MemoryClusterTemplate(clusterA) {
  ...
  MemoryBistInterface(sb1) {
    ...
    LogicalMemoryToInterfaceMapping (LM_0) { ... }
    LogicalMemoryToInterfaceMapping (LM_1) { ... }
    LogicalMemoryToInterfaceMapping (LM_2) { ... }
  }
} // clusterA

MemoryClusterTemplate(clusterB) {
  ...
  MemoryBistInterface(sb2) {
    ...
    LogicalMemoryToInterfaceMapping (LM_3) { ... }
  }
} // clusterB
```

Hierarchical instance paths of repairable memories must be identified in the MemoryInstanceName property of the Shared Bus memory cluster and logical memory TCD. In Figure 6-17, logical memory LM_3 is associated to the new Shared Bus memory cluster B that is instantiated at the same hierarchical level as the original Shared Bus memory cluster A. The instance path from Shared Bus memory cluster B to LM_3 would require to traverse from cluster B up to its parent module then down into cluster A.

Using the variables of Table 6-1, the instance path to logical memory LM_3 can be derived as shown below. The %CLUSTER_PARENT% variable is expanded to the instance path of the module instantiating Shared Bus memory cluster B.

```
MemoryClusterTemplate(clusterB) {
  ...
  MemoryBistInterface(sb2) {
    ...
    LogicalMemoryToInterfaceMapping (LM_3) {
      MemoryInstanceName : %CLUSTER_PARENT%/cluster_a_inst/LM_3_inst;
    }
  }
} // clusterB
```

This chapter describes how to implement custom user-defined algorithms (UDAs) under Tessent Shell MemoryBIST flow.

Usage Context

Tessent MemoryBIST is provided with a library of predefined memory test algorithms generically referred to as the Tessent Library Algorithms. Memory core description files then explicitly point to those algorithms to get implemented during memory BIST insertion.

It is however, possible to implement additional algorithms that are not part of the Tessent library; these are typically called custom or User-Defined Algorithms (UDAs). This chapter explains how to implement them using a Tessent Shell flow.

> **Tip**
> The syntax for UDAs is exactly the same whether a custom algorithm is hard-coded (defined at design time) or soft-coded (defined at silicon test time).

Overview and Terminology

There is ample literature covering memory test in general, test algorithms, and memory fault coverage, so this section focuses only on the essential information.

A typical memory test usually applies a series of specific sequences across the entire memory array. A very simple March test is described as the following steps:

1. From the bottom of the memory to the top, write a background of <dataPattern>;

2. From bottom to top, read <dataPattern> and immediately write <inverseDataPattern>;

3. From bottom to top, read <inverseDataPattern> and immediately write <dataPattern>;

4. From bottom to top, read <dataPattern>.

The very same algorithm could also be written using this commonly-used compact syntax:

Simple March algorithm = ^{w0}, ^{r0w1}, ^{r1w0}, ^{r0};

The above algorithm is essentially split into four distinct phases, with each phase shown within braces. For each phase:

- The caret symbol ("^") translates to "increasing address locations", while a lowercase letter "v" (not shown) would imply "decreasing address locations".

- The letter "r" stands for a memory read while "w" means a memory write.

- The "0" and "1" symbols respectively translate to "non-inverted data pattern" and "inverted data pattern" (not necessarily to logic 0 or 1 values).

The above description is very high-level. It does not state the exact data pattern used or the exact address locations to test. To be applied to an actual memory, the test algorithm would therefore need an initialization sequence specifying such information.

A typical Tessent MemoryBIST UDA can be described similarly; it includes an initialization section followed by the actual high-level algorithm. A custom algorithm description contains several test phases called instructions; each instruction consists of a given operation that is repeatedly applied until specific next conditions are met. At that point, the memory BIST controller proceeds to the next sequential instruction - that is, phase - and continues until the entire algorithm has been run.

Because a single memory BIST controller can test multiple memories in parallel, the default minimum and maximum address locations vary. Although the minimum address is normally an

all-zero value, the maximum address is usually derived from the highest possible bank, row, and column address segments reached by all BIST controller-tested memories.

Looking at the preceding example, one might therefore describe this algorithm as follows, which would more closely match the Tessent MemoryBIST language:

- instruction(w0) = operation: write, write_data: 000...000, address_counter: increment, until_address_equals: max_address;

- instruction(r0w1) = operation: read_modify_write, read_compare_data: 000...000, write_data: 111...111, address_counter: increment, until address_equals: max_address;

- instruction(r1w0) = operation: read_modify_write, read_compare_data: 111...111, write_data: 000...000, address_counter: increment, until address_equals: max_address;

- instruction(r0) = operation: read, read_compare_data: 000...000, address_counter: increment, until address_equals: max_address.

The following sections in this chapter expand the UDA concept in more detail.

Defining a Custom Algorithm

To successfully implement a UDA, first determine what the algorithm does. If its high-level description is available (from existing documentation or elsewhere), use it; otherwise try to break it down into smaller phases and for each, determine how the address counter and apply/ expect data registers behave.

> **Note**
> It is usually a bad idea to reverse-engineer a low-level algorithm description (such as the raw pattern output of a memory tester) and try to derive a higher-level algorithm from it. Such cycled sequences do not convey the algorithmic nature of memory tests very well. They also often contain hardware-specific optimizations or implementation shortcuts, making it very difficult to precisely duplicate using a memory BIST controller.

For any given target algorithm, analyze its various implicit test sub-phases and answer the following questions for each:

- At what address is the test phase starting (that is, bottom, top, or elsewhere)?

- In what direction is the address counter going (that is, incrementing, decrementing, idle)?

- Is it incrementing/decrementing over the entire address range or just across some specific address segments (that is: banks, rows or columns)?

- What is the increment/decrement value (for example: +/-1, +/-2, and so on, or something else)?

- What memory operation(s) is (are) applied at any given address, before moving on to the next address location (for example: write, read, read-modify-write, write-read, no-op, and so on)?

- What data pattern is written (if any)? What data is expected to be read (if any)?

- Do address counters and data registers need to implement a particular physical-to-logic mapping (for example, a scrambling table)?

Once you have answers to the above questions, coding a UDA is a lot simpler. Many algorithm characteristics directly translate into Tessent MemoryBIST properties.

Optimizing Custom Algorithms and Operation Sets

This section explains how to optimize Tessent MemoryBIST custom algorithms and operation sets by eliminating redundant memory accesses, which are sometimes introduced because of the requirement that each operation performed by the controller is at least two ticks long. Several properties are available in Tessent MemoryBIST to eliminate these redundancies, enabling a reduction in test time and an improvement of test quality for some memories.

Optimization Overview

The architecture of Tessent MemoryBIST is based on an architecture where each operation performed by the controller requires at least two cycles to run. In built-in operation sets like SYNC or SYNCWR, address counters, data registers, instruction pointer, and many other registers only increment every second clock cycle. This architecture simplifies timing closure significantly without sacrificing defect coverage.

Timing closure is simplified because most of the controller is working at half speed and synthesis and layout only have a few timing critical signals to concentrate on. From an at-speed defect coverage perspective, the only requirement for most documented fault models is to perform a read or write on a first memory location in a first clock cycle and perform another read or write to a second location in the next clock cycle. This is achieved at the boundary of the 2-tick operations when the address changes.

For example, consider a March test phase such as R0W1. For each address, a "0" is first read and then a "1" is written. Using the ReadModifyWrite operation of the SYNC or SYNCWR operation set, this operation requires two clock cycles and the address counter is effectively clocked at half speed. However, at-speed coverage is achieved at the transition between the two address locations because a "1" is written at a first location and a "0" is read at a different location in consecutive clock cycles. This test is very stressful for the memory, especially if the two locations are performed in the same column.

The built-in operation sets were optimized for library algorithms, which are mainly based on ReadModifyWrite operations. However, these operation sets do not allow an efficient implementation of other algorithms in certain cases. Redundant memory accesses increase test time and some memories require the ability to change the address on more than two consecutive clock cycles to test their decoders adequately. Several properties are available in Tessent

MemoryBIST to create custom algorithms and operation sets eliminating these redundant memory accesses.

The following sections describe the OperationSet wrapper properties that are useful for eliminating redundant memory accesses in custom algorithms, and provide examples of how these properties can be used. Two groups of properties are described, the first of which enables modifying the address within an operation, and the second enables modifying the data pattern within an operation. These sections are followed by a discussion on the implications on diagnosis when using these properties. The final section provides a set of additional optimization recommendations to consider for improving custom operation sets.

Optimizing Properties Modifying the Address Within an Operation

Three properties allow a change of the address applied to the memory within an operation: column_address_count_enable, row_address_count_enable, and switch_address_register.

The Cycle/AdvancedSignals column_address_count_enable and row_address_count_enable properties are useful to advance the address counter on every clock cycle for fast initialization or complete readout of the memory. Figure 7-1 shows the relevant portions of a custom algorithm, and Figure 7-2 shows the corresponding operation set illustrating the use of these properties. The algorithm writes a checkerboard pattern using a "fast column" address sequence implemented with address register A and reads the pattern back using a "fast row" address sequence implemented with address register B. The checkerboard pattern is obtained by inverting the Write and Expect data with the least significant bit (LSB) of the column and row address bits.

Figure 7-1. Simple Algorithm Writing and Reading all Memory Locations

```
MemoryOperationsSpecification {
  Algorithm (FastWriteRead) {
    TestRegisterSetup {
      operation_set_select: MyCustomOpSet;
      AddressGenerator {
        AddressRegisterA {
          x1_carry_in: y1_carry_out;
          y1_carry_in: none;
        }
        AddressRegisterB {
          x1_carry_in: none;
          y1_carry_in: x1_carry_out;
        }
      }
      DataGenerator {
        load_write_data : all_zero;
        load_expect_data: all_zero;
        invert_data_with_column_bit: c[0];
        invert_data_with_row_bit : r[0];
      }
```

```
MicroProgram {
  Instruction (InitializeRAM) {
    branch_to_instruction: InitializeRAM;
    operation_select: Write2CellsFastY;
    AddressCommands {
      address_select: select_a;
      x1_address: increment;
      y1_address: increment;
    }
    DataCommands {
      write_data: data_reg;
    }
    NextConditions {
      x1_end_count: on;
      y1_end_count: on;
    }
  }

  Instruction (ReadRAM) {
    branch_to_instruction: ReadRAM;
    operation_select: Read2CellsFastX;
    AddressCommands {
      address_select: select_b;
      x1_address: increment;
      y1_address: increment;
    }
    DataCommands {
     expect_data: data_reg;
    }
    NextConditions {
      x1_end_count: on;
      y1_end_count: on;
    }
  }
 }
 }
}
```

The algorithm has two instructions, InitializeRAM and ReadRAM using operation
Write2CellsFastY and Read2CellsFastX respectively. Each instruction runs until the last
address of the memory is reached. When using the Write and Read operations of the built-in
operation sets (SYNC or SYNCWR), this algorithm takes 2*2*N cycles, where N is the number
of address locations, because the address registers only update at the end of the 2-cycle
operations. However, the custom operation set of Figure 7-2 reduces the number of cycles to
2*N. This is because the address register selected by the instruction updates in both the first and
second clock tick of the selected operation, because of the presence of the
column_address_count_enable and the row_address_count_enable properties in the first Cycle
of operation Write2CellsFastY and Read2CellsFastX respectively.

Figure 7-2. Example Operation Set Using row_address_count_enable and column_address_count_enable

```
MemoryOperationsSpecification {
  OperationSet (MyCustomOpSet) {
    …
    SignalPipelineStages {
      strobe_data_out: 1;
    }
    Operation (Write2CellsFastY) {
      Cycle {
        select: on;
        write_enable: on;
        read_enable: off;
        AdvancedSignals {
          column_address_count_enable: on; // Write fast column
        }
      }
      Cycle {
      }
    }

    Operation (Read2CellsFastX) {
      Cycle {
        select: on;
        write_enable: off;
        read_enable: on;
        AdvancedSignals {
          row_address_count_enable: on; // Read fast row
        }
        strobe_data_out: on;
      }
      Cycle {
        strobe_data_out: on;
      }
    }
  }
}
```

Note that these properties are "sticky" within an operation in that they retain their value (on or off) from one tick to the next unless explicitly set to the opposite value. The default is off at the beginning of an operation. Note that these operations need to be used with the appropriate address register (A and B respectively), whereas the built-in Write and Read operations are used with any address register.

For the majority of algorithms, the row and column increment/decrement value is 1. By default, Tessent MemoryBIST generates address registers that are optimized for this case and run at higher speed. Also, starting with version 2016.3, it is no longer necessary to code the algorithm using X0/Y0 address segments to benefit from optimized address registers. X0/Y0 address segments are automatically inferred as needed. This enhancement has several advantages. The same algorithm can be used for any memory configuration. Prior to the 2016.3 release, Y0 address segments were not allowed for memories with no or only one column address bits. This limitation affected test time and complexity of the test plan for controllers containing a mix of

memories with different numbers of column address bits. Timing closure is now easier as the hardware is always configured to achieve the best performance.

If it is intended to use a row or column increment/decrement value of 2 or more in any custom algorithm, hard or soft-coded, the value of max_x0_segment_bits and max_y0_segment_bits need to be set to auto in the MemoryBist/Controller/AlgorithmResourceOptions wrapper of the DftSpecification. The optimized address registers are not used in this case and timing closure might be more difficult for high speed circuits.

The use of column_address_count_enable and row_address_count_enable properties assumes an even number of column and row addresses in a BIST controller step. This is never an issue for column addresses. For the case where the maximum number of rows in a step is determined by a memory with an odd number of rows, the number of rows is automatically increased by one for that step. This means that the memory is deselected whenever the address counter reaches the maximum row address. No intervention from the user is required to handle this situation.

There is an additional, related assumption to be aware of when coding custom operations using column_address_count_enable and row_address_count_enable with optimized address registers (for example, max_x0_segment_bits: 1 and max_y0_segment_bits: 1). These properties assume that the address at the start of the operation, and after execution of the operation, is Even if the AddressCommands property of the instruction is increment, and Odd, if decrement. The address sequence might be slightly different than the one expected if this assumption is not true. For example, consider the algorithm of Figure 7-3 and the associated operation set of Figure 7-4. The algorithm writes column bars in memory and reads the memory along diagonals. When initializing the memory and reading a memory diagonal, the address changes on every clock cycle.

The initialization of the memory is essentially identical to the previous example using the InitializeRAM instruction and Write2CellsFastY operation. Address register A is segmented the same way as well, counting fast column. The only difference related to the initialization is that the data pattern is inverted with the LSB of the column address to create column bars. This pattern causes the memory output to flip on every clock cycle when executing the ReadDiagonal_Inst instruction using the ReadDiagonal operation and address register B. Address register B is configured so that both the row address and column address can increment at the same time to read along diagonals. The column_address_count_enable and row_address_count_enable are both used in the operation so that address transitions occur on every clock cycle.

For the first diagonal, the address sequence is regular (R0C0, R1C1, R2C2, …) until reaching the last row. The row address counter wraps around to row address 0 to start the next diagonal. The column address at that time is determined by the ratio of the number of rows divided by the number of columns. The remainder of this division is guaranteed to be even by construction. However, the column address value is not important because it is overridden when executing instruction AddColumnOffset_Inst. This instruction does not perform any write or read to the memory but increments the column address of address register A that keeps track of the

diagonal number, and the result is copied back into address register B to provide the starting point of the next diagonal.

For the second diagonal, the column address is Odd, and the AddressCommands property is increment. This produces a broken diagonal pattern (R0C1, R1C0, R2C3, R3C2, …). Instead of incrementing by 1 on every cycle, the column address is decrementing by 1, then incrementing by 3, decrementing by 1, and so on. The behavior is similar for all diagonals starting at an odd column address such that all memory locations are read, and both the row and column addresses are changed on every clock cycle. Therefore, there is no loss of defect coverage due to this behavior. Note that regular diagonals are obtained if max_x0_segment_bits and max_y0_segment_bits are set to auto instead of 1, but optimized address registers can not be used in that case.

Figure 7-3. Fast Diagonal Algorithm

```
MemoryOperationsSpecification {
  Algorithm (FastDiagonal) {
    TestRegisterSetup {
      operation_set_select: FastDiagonalOpset;
      AddressGenerator {
        AddressRegisterA {
          x1_carry_in: y1_carry_out;
          y1_carry_in: none;
        }
        AddressRegisterB {
          x1_carry_in: none;
          y1_carry_in: none;
        }
      }
      DataGenerator {
        invert_data_with_column_bit : c[0]; // create column bars
      }
    }

    MicroProgram{
      Instruction ( InitializeRAM ) {
        branch_to_instruction: InitializeRAM;
        operation_select: Write2CellsFastY;
        AddressCommands {
          address_select: select_a;
          x1_address: increment;
          y1_address: increment;
        }
        DataCommands {
          write_data: data_reg;
        }
        NextConditions {
          x1_end_count: on;
          y1_end_count: on;
        }
      } // end instruction InitializeRAM
```

```
      Instruction ( ReadDiagonal_inst ) {
        branch_to_instruction: ReadDiagonal_inst;
        operation_select: ReadDiagonal;
        AddressCommands {
          address_select: select_b;
          x1_address: increment;
          y1_address: increment;
        }
        DataCommands {
          expect_data: data_reg;
        }
        NextConditions {
          x1_end_count: on;
        }
      }

      Instruction ( AddColumnOffset_inst ) {
        branch_to_instruction: ReadDiagonal_inst;
        operation_select: NoOperation;
        AddressCommands {
          address_select: select_a_copy_to_b;
          x1_address: hold;
          y1_address: increment;
        }
        DataCommands {
          expect_data: data_reg;
        }
        NextConditions {
          y1_end_count: on;
          RepeatLoopA {    // Repeat algorithm with inverse data
            branch_to_instruction: InitializeRAM;
            Repeat1 {
              write_data_sequence: inverse;
              expect_data_sequence: inverse;
            }
          }
        }
      } // end instruction
    } // MicroProgram
  }
}
```

Figure 7-4. Fast Diagonal Operation Set

```
MemoryOperationsSpecification {
  OperationSet (FastDiagonalOpset) {
  //
  //Pipeline strobe so that it can be specified in same cycle as read
  // enable and each operation can read two locations
    SignalPipelineStages {
      strobe_data_out: 1;
    }
    Operation (NoOperation) {
      Cycle {
        select: on;
      }
      Cycle {
      }
    } // end of operation (NoOperation)

    Operation (Write2CellsFastY) {
      Cycle {
        select: on;
        write_enable: on;
        AdvancedSignals {
         column_address_count_enable: on;
        }
      }
      Cycle {
      }
    } // end of operation (Write2CellsFastY)

    Operation (ReadDiagonal) {
      Cycle {
        select: on;
        write_enable: off;
        strobe_data_out: on;
        AdvancedSignals {
          column_address_count_enable: on;
          row_address_count_enable: on;
        }
      }
      Cycle {
        strobe_data_out: on;
      }
    } // end of operation (ReadDiagonal)
  } // end of operation set
}
```

The number of diagonals is the same as the number of column addresses. This is reflected in the NextConditions wrapper of the AddColumnOffset_Inst instruction. All diagonals are re-tested using the inverse data pattern, as indicated in the RepeatLoopA wrapper of the same instruction.

The Cycle/AdvancedSignals switch_address_register property is useful in algorithms where several memory operations are performed on a reference cell and its neighbors. An example of such an algorithm is the LVBitSurroundDisturb Algorithm, where each write to a neighbor is followed by a read to the reference cell. The library algorithm currently uses the SYNC

operation set so that two operations are required to write to a neighbor and read the reference cell. Both operations are two cycles long, resulting in half of the memory accesses being redundant. The switch_address_register property enables combining the two instructions into a single instruction through making use of a specialized operation. In Figure 7-5, the instruction WRITE_AWAY_READ_HOME_INST1 replaces instructions WRITE_AWAY_CELL1 and READ_HOME_CELL1 of the original algorithm. The instruction makes use of the operation called WRITE_AWAY_READ_HOME. In the first clock cycle, the write to the neighbor is performed using the data specified by the write_data property of the instruction. In the second cycle, the read to the reference is performed and compared to the data specified by the expect_data property of the instruction. To continue with the optimization, WRITE_AWAY_CELL2 and READ_HOME_CELL2 of the original algorithm are combined in the same manner. Note that RepeatLoopB in READ_HOME_CELL2 must now point to WRITE_AWAY_READ_HOME_INST1 instead of WRITE_AWAY_CELL1.

Figure 7-5. Example Usage for the switch_address_register Property

```
MemoryOperationsSpecification {
  Algorithm (LVBitSurroundDisturb_optimized) {
  …
    MicroProgram {
    …
      Instruction (WRITE_AWAY_READ_HOME_INST1) {
        branch_to_instruction: WRITE_AWAY_READ_HOME_INST1;
        operation_select: WRITE_AWAY_READ_HOME;
        AddressCommands {
          address_select: select_b;
        }
        DataCommands {
          write_data: inverse_data_reg;
          expect_data: data_reg;
        }
        AddressCommands {
          x1_address: hold;
          y1_address: increment;
        }
        CounterCommands {
          counter_a: increment;
        }
        NextConditions {
          counter_a_end_count: On;
        }
      }
    }
    …
  }
```

```
OperationSet (MyCustomOpSet) {
  …
  SignalPipelineStages {
    strobe_data_out: 1;
  }
  …
  Operation (WRITE_AWAY_READ_HOME) {
      Cycle {
        select: on;
        write_enable: on;
      }
      Cycle {
        write_enable: off;
        AdvancedSignals {
          switch_address_register: on;
        }
        // Strobe delayed with SignalPipelineStages/StrobeDataOut: 1;
        strobe_data_out: on;
      }
    }
  }
}
```

Optimizing Properties Modifying the Data Within an Operation

The Cycle/AdvancedSignals wrapper invert_write_data and invert_expect_data properties are useful for modifying the data pattern within an operation when several memory accesses are performed to the same address as part of this operation.

For example, consider the R0W1R1 and R1W0R0 sequences used in a custom algorithm. A value is read in the first clock cycle but the opposite value is read in the third clock cycle. Because only one DataCommands/expect_data property can be specified in a given instruction, the invert_expect_data property must be used in the selected operation to adjust the expected value, as shown in . Note that invert_expect_data is not "sticky", contrary to the majority of other properties. That is, it is only on in the cycles where it is explicitly set to on. The invert_expect_data property must be specified in the same cycle as the corresponding strobe_data_out.

Figure 7-6. Example Illustrating the use of invert_expect_data

```
MemoryOperationsSpecification {
  OperationSet (my_opset) {
    Operation (ReadWriteRead) {
      Cycle {
        write_enable: off;
        // First read operation. Expect data specified by
        // expect_data of instruction
        read_enable: on;
        // Strobe delayed with SignalPipelineStages{strobe_data_out:1;}
        strobe_data_out: on;
      }
```

```
      Cycle {
      // Write operation. Write data specified by instruction write_data
      // is the inverse of the value specified by instruction expect_data
        write_enable: on;
        read_enable: off;
      }

      Cycle {
        write_enable: off;
        // Second read operation. Expect data specified by expect_data
        // is inverted using the invert_expect_data property
        read_enable: on;
        AdvancedSignals {
          invert_expect_data: on;
        }
        strobe_data_out: on;
      }
    }
  }
}
```

Another example is the W0W1W0W1R1 sequence used in a custom algorithm. Multiple writes of different values are performed in consecutive cycles. Because only one DataCommands/write_data property can be specified in a given instruction, the invert_write_data property must be used in the selected operation to adjust the write value as shown in . Note that invert_write_data is also not sticky.

Figure 7-7. Example Illustrating the use of invert_write_data

```
MemoryOperationsSpecification {
  OperationSet (my_opset) {
    Operation (WriteToggleRead) {
      Cycle {
        select : on;
        write_enable : on;
      }
      Cycle {
        AdvancedSignals {
          invert_write_data: on;
        }
      }
      Cycle {
        AdvancedSignals {
          invert_write_data: off;
        }
      }
      Cycle {
        AdvancedSignals {
          invert_write_data: on;
        }
      }
      Cycle {
        write_enable: off;
        strobe_data_out: on; //Early strobe is pipelined
      }
    }
  }
}
```

Diagnosis Considerations

When using the properties described in the two previous sections, strobes probably occur in consecutive cycles.

For example, when applying the ReadWriteRead operation of Figure 7-6 "Example illustrating the use of invert_expect_data" in an algorithm testing several memory locations, strobes occur in consecutive cycles at every address transition. In such a case, there is one limitation concerning the operation of Stop-On- Nth-Error.

In the PatternsSpecification used to generate testbenches and test patterns, the DiagnosisOptions/StopOnErrorOptions/data_compare_time_slots property needs to be set to even or odd when running in Stop-On-Nth-Error mode to avoid obtaining inconsistent diagnostic results if errors are occurring in consecutive clock cycles. This requirement means that the test must be run twice to extract all memory failures. This trade-off simplifies timing closure and improves the performance of the Stop-On-Nth-Error mode.

Optimization Recommendations

The following are optimization recommendations for algorithms, operation sets and relevant memory library file properties.

Minimize the Use of X0/Y0 Segmentation

Before the 2016.3 release, it was necessary to explicitly segment the row and column address register in the algorithm for optimizing controller speed when using row_address_count_enable and column_address_count_enable properties in the selected operation set. The disadvantage of this method is that three versions of the same algorithm were required to handle memories with 0, 1, or 2 or more column address bits respectively. This requirement is no longer necessary and a single algorithm can now be used. The software automatically implements the segmentation as needed for each type of memory. Refer to the "Optimizing Properties Modifying the Address Within an Operation" section for further information.

Use Unsized Constants or Variables

- Set the least significant bit of the bank address in AddressRegisterA or AddressRegisterB for any memory configuration:

```
AddressRegisterA {
  load_bank_address: b1;
}
```

As opposed to:

```
AddressRegisterA {
  load_bank_address: 3'b001;
}
```

- Set the maximum count value of CounterA:

```
TestRegisterSetup {
  load_counter_a_end_count: num_bank_address_bits;
}
```

As opposed to:

```
TestRegisterSetup {
  load_counter_a_end_count: 3;
}
```

Standardize and Minimize the Size of Data Registers

The majority of algorithms only require two bits for write and expect data registers. The data is replicated as needed for each memory. However, the default register size is chosen to be the size of the largest data path of all memories attached to the controller when using a soft programmable controller or hard-coded custom algorithms. This increases controller area and routing congestion between the controller and the memories unnecessarily.

The DftSpecification AlgorithmResourceOptions/data_register_bits property needs to be explicitly set when generating the controller in that case. The value of this property should not be higher than the length of any unique pattern specified by the DataGenerator load_write_data and load_expect_data properties in any of the algorithms. Use of the symbolic values all_zero, all_one, and so on, for these properties ensures this recommendation, otherwise the considerations outlined in the following examples apply. Also note that you do not need to specify these properties if the pattern is all 0s, which is the default setting.

Example 1

The following DftSpecification and Algorithm wrapper settings:

```
AlgorithmResourceOptions {
  data_register_bits: N;
}

DataGenerator {
}
```

are equivalent to the following settings:

```
AlgorithmResourceOptions {
}

DataGenerator {
  load_expect_data: N'b00..00;
  load_write_data:  N'b00..00;
}
```

Where N represents the bit width of the memory data path.

Example 2

When the following DftSpecification and Algorithm settings:

```
AlgorithmResourceOptions {
  data_register_bits: M;  // Where M < N
}

DataGenerator {
  load_expect_data: N'b01..0101;
  load_write_data:  N'b01..0101;
}
```

are replaced with these settings:

```
AlgorithmResourceOptions {
}

DataGenerator {
  load_expect_data: M'b01..0101;
  load_write_data:  M'b01..0101;
}
```

The pattern specified in the algorithm is truncated and the intent of the algorithm is preserved.

Example 3

When the following DftSpecification and Algorithm settings:

```
AlgorithmResourceOptions {
  data_register_bits: M;  // Where M > N
}

DataGenerator {
  load_expect_data: N'b01..0101;
  load_write_data:  N'b01..0101;
}
```

are replaced with these settings:

```
AlgorithmResourceOptions {
}

DataGenerator {
  load_expect_data: M'b01..0101;
  load_write_data:  M'b01..0101;
}
```

The pattern specified in the algorithm is padded with 0s and the intent of the algorithm is *NOT* preserved.

Use a Single Operation Set for All Algorithms

It is more efficient to develop a common operation set for all algorithms applicable to certain memory types, such as synchronous RAM, ROM, DRAM, and CAM. This ensures consistency in the implementation of common operations, such as ReadModifyWrite, and provides some area reduction. This also provides more flexibility when writing soft algorithms that were not originally available at the time that the controller was generated. Operation sets cannot be modified after the controller is generated, and only a single operation set can be specified when executing any algorithm. Note that Siemens EDA now provides a general operation set applicable to more than 50 library algorithms. This operation set can be augmented, if needed, for new algorithms requiring specialized operations.

Use One of SMarchCHKB* Library Algorithms as Default

Each of the SMarchCHKB* algorithms is actually a combination of algorithms that have been developed over the past 20 years. They have special properties that are not applicable to custom algorithms.

- Application to multi-port memories

  The algorithms have been designed to be applied to both single- and multi-port memories in parallel, while testing multi-port specific faults.

- Automation of Parallel Static Retention Testing

Pattern generation is greatly simplified because the algorithm is automatically split into the three phases (StartToPause, PauseToPause, PauseToEnd) and test patterns are generated with instructions for the test engineer. Each phase only includes the algorithm instructions that are essential to the retention test to optimize test time.

- Checkerboard pattern application

The data pattern is automatically adjusted to the particular memory configuration to apply the proper checkerboard pattern required for performing specialized tests, such as bit line shorts or coupling.

Use the Physical Data Map Correctly

It is usually not necessary to provide a PhysicalDataMap wrapper. There are a few common mistakes that might result in a lower quality test or significant area increase of the memory interface.

- Inversion based on column address c[0]

As an example, Data[0]: d[0] xor c[0], which indicates each column is mirrored with respect to its neighbors. This mapping should not be specified for an SRAM. By default, SMarchCHKB* library algorithms assume this inversion is present for an SRAM, and applies the correct data pattern based on the effective value of BitGrouping, which is typically "1" for memories with column address bits and "N" for memories without. However, an incorrect pattern is applied if the PhysicalDataMap wrapper contains the equation shown in this example.

- Inversion of data for data bits implemented in different sub-arrays

As an example, for a memory with N data bits implemented as two sub-arrays of N/2 bits, the following PhysicalDataMap wrapper should not be used:

```
PhysicalDataMap {
  Data[0] : d[0];
  Data[1] : d[1];
  ...
  Data[N/2 - 1] : d[N/2 - 1];
  Data[N/2] : ~d[N/2];
  Data[N/2 + 1] : ~d[N/2 + 1];
  ...
  Data[N-1] : ~d[N-1];
}
```

This usage causes the addition of unnecessary logic. The default data mapping does not result in any reduction of the fault coverage.

Optimization Benefits

Following the optimization recommendations outlined in this section enables the designer to realize significant advantages:

- Simplification of pattern generation, as it is no longer necessary to know the memory configuration to select the appropriate algorithm

- Reduction in the number of scan operations necessary to select the algorithms, as it is no longer necessary to operate in freeze_step mode. The same algorithm can be applied to all controller steps

- Consistency and simpler maintenance of the algorithms

- Reduction of the MemoryBIST logic area requirements

Example Algorithm: March C- Algorithm

A March C- algorithm can be generically described as follows:

1. Write background of D-data incrementing from address minimum to address maximum.

2. Read D-data and write Dbar-data incrementing from address minimum to address maximum.

3. Read Dbar-data and write D-data incrementing from address minimum to address maximum.

4. Read D-data and write Dbar-data decrementing from address maximum to address minimum.

5. Read Dbar-data and write D-data decrementing from address maximum to address minimum.

6. Read D-data decrementing from address maximum to address minimum.

This basic algorithm is well-known in the memory test field; one of its variants can be described in standard notation syntax as follows:

$$\{\updownarrow (w0); \Uparrow(r0, w1); \Uparrow(r1, w0); \Downarrow(r0, w1); \Downarrow (r1, w0); \updownarrow(r0)\}$$

Note that in the preceding algorithm variant, phases corresponding to steps 1 and 6 show arrows pointing both up and down; these phases can run either way. For simplicity, this example still assumes "up" and "down" directions for steps 1 and 6 (respectively).

Coding the Algorithm **395**

Coding the Algorithm

Coding of the algorithm involves setting up the initial MBIST controller register configuration, development of the microprogram wrapper and lastly, incorporating any algorithm optimizations that may be needed.

TestRegisterSetup Wrapper

The TestRegisterSetup wrapper determines the initial configuration (setup) of an MBIST controller's registers. To specify it, first answer the following key algorithm questions:

- What library (or possibly custom) operation set is to be used? [Answer: SyncWR]

- How does the address counter count (for example, column or row first)? [Answer: Row first]

- What default write and expect data pattern is to be used? [Answer: All zeros]

The test register setup portion of this algorithm would therefore look like this (line numbers added to the left for reference):

```
1    MemoryOperationsSpecification {
2      Algorithm (MarchCMinus){
3        TestRegisterSetup {
4          operation_set_select : SyncWR;
5          AddressGenerator {
6            AddressRegisterA {
7              y1_carry_in : x1_carry_out;
8              x1_carry_in : none;
9            }
10         }
11         DataGenerator {
12           load_write_data  : all_zero;
13           load_expect_data : all_zero;
14         }
15       }
16       MicroProgram {
17       }
18     }
19   }
```

Line 4 in the preceding example indicates the operation set from which you want to choose operations. Lines 6-9 configure address register "A" to count fast rows and lines 12-13 specify the intended data pattern.

MicroProgram Wrapper

The MicroProgram wrapper represents the actual test algorithm. For each algorithm step, answer the following questions:

- What specific memory BIST operation is applied, from the selected operation set?

- Do the address segments decrement, hold, or increment?

- If appropriate, what data pattern should be written or observed?

- When should the algorithm advance to its next step or phase?

Procedure

1. The MBIST controller first writes the data pattern (zeros for example) from address min to address max:

```
1   MemoryOperationsSpecification {
2     Algorithm (MarchCMinus) {
3       TestRegisterSetup {
4       }
5       Microprogram {
6         Instruction (M0_W0) {
7           operation_select : Write;
8           AddressCommands {
9             x1_address : increment;
10            y1_address : increment;
11          }
12          DataCommands {
13            write_data : data_reg;
14          }
15          NextConditions {
16            x1_end_count : on;
17            y1_end_count : on;
18          }
19        }
20      }
21    }
22  }
```

In the preceding example, the very first algorithm instruction word is arbitrarily named "M0_W0"; it indicates this is step zero of a March algorithm and this step writes zeros across the entire memory.

Line 7 indicates that memory BIST must repeatedly apply the "write" operation. Lines 9-10 specify that after every write application, a column or row increment (or both) should occur. Line 13 implies that MBIST should write the data_reg value (zeros, for example) to the memory.

Lines 15-18 state that when row and column address segments both reach their maximum values (or "end count"), the algorithm should then move on to the next step (or "instruction").

_____ **Note** _____

Tessent MemoryBIST uses "post-increment"-style address counters. Any memory operation is therefore fully run before address counters change; the next address becomes effective only after the selected operation's last clock tick.

If an address segment increments and reaches its maximum value, it wraps around and rolls back to its minimum value. Conversely, a decrementing address segment reaching its minimum value wraps around to its maximum value.

_____

This algorithm phase effectively writes zeros across the entire memory array. Once it reaches the top of the memory, it performs one last write and then wraps around to the bottom of the memory (that is its minimum address).

2. This part of the March C- algorithm can be written as follows:

```
1   MemoryOperationsSpecification {
2      Algorithm (MarchCMinus) {
3         TestRegisterSetup {
4         }
5         Microprogram {
6           Instruction (M0_W0) {
7           }
8           Instruction (M1_R0W1) {
9             operation_select : ReadModifyWrite;
10            AddressCommands {
11              x1_address : increment;
12              y1_address : increment;
13            }
14            DataCommands {
15              write_data : inverse_data_reg;
16              expect_data : data_reg;
17            }
18            NextConditions {
19              x1_end_count : on;
20              y1_end_count : on;
21            }
22          }
23        }
24      }
25  }
```

Following the very same methodology used with step 1, line 8 now indicates that zeros are expected to be read from the memory and ones are written to it. Lines 15-16 thus properly reflect the data to write and expect.

Because the algorithm reads and then writes each and every memory location, the selected operation (on line 9) is chosen to be read-modify-write. After going through all address locations, the next algorithm instruction is selected (as indicated by lines 19-20).

3. This step turns out to be the same as step 2, except for its write and expect data patterns (which are the opposite). It can, therefore, be written as follows:

```
1   MemoryOperationsSpecification {
```

```
2    Algorithm (MarchCMinus) {
3       TestRegisterSetup {
4       }
5       Microprogram {
6          Instruction (M0_W0) {
7          }
8          Instruction (M1_R0W1) {
9          }
10         Instruction (M2_R1W0) {
11            operation_select : ReadModifyWrite;
12            AddressCommands {
13               x1_address : increment;
14               y1_address : increment;
15               inhibit_last_address_count : on; // <= Note this
16            }
17            DataCommands {
18               write_data : data_reg;
19               expect_data : inverse_data_reg;
20            }
21            NextConditions {
22               x1_end_count : on;
23               y1_end_count : on;
24            }
25         }
26      }
27   }
28 }
```

Line 15 in the preceding example requires additional explanations.

Recall that the next step (step 4) performs memory operations in a decrementing address order fashion. Given that the memory BIST controller uses post-increment address counters, it would normally complete step 3 with a read-modify-write operation performed at the maximum memory address, and then roll back or wrap around to the minimum address location.

However, because step 4 is about to be performed with decrementing address pointers, a method is needed to instruct the MBIST controller to effectively hold or stop at the top of the memory, after completing its last memory operation of step 3. The inhibit_last_address_count property achieves this purpose; it prevents the address segments from wrapping around when a minimum or maximum address is reached.

_____ **Tip** _____

ⓘ As a rule of thumb, use the inhibit_last_address_count property whenever the next immediate algorithm step (instruction) modifies the addressing direction.

_____

The next step starts from the top of the memory - that is, at its maximum address.

4. This phase of the algorithm is just like step 2, however, it is performed with decrementing (instead of incrementing) address segments. At this point, you may already know the description:

```
1  MemoryOperationsSpecification {
```

```
 2    Algorithm (MarchCMinus) {
 3       TestRegisterSetup {
 4       }
 5       Microprogram {
 6          Instruction (M0_W0) {
 7          }
 8          Instruction (M1_R0W1) {
 9          }
10          Instruction (M2_R1W0) {
11          }
12          Instruction (M3_R0W1) {
13             operation_select : ReadModifyWrite;
14             AddressCommands {
15                x1_address : decrement;
16                y1_address : decrement;
17             }
18             DataCommands {
19                write_data : inverse_data_reg;
20                expect_data : data_reg;
21             }
22             NextConditions {
23                x1_end_count : on;
24                y1_end_count : on;
25             }
26          }
27       }
28    }
29 }
```

Note that nothing special needs to be done about the next conditions and end count properties with decrementing address segments. Tessent MemoryBIST considers the step to be completed when the final address location is the minimum address and address segments decrement.

5. This phase of the algorithm is just like step 3 but uses decrementing address segments:

```
 1 MemoryOperationsSpecification {
 2    Algorithm (MarchCMinus) {
 3       TestRegisterSetup {
 4       }
 5       Microprogram {
 6          Instruction (M0_W0) {
 7          }
 8          Instruction (M1_R0W1) {
 9          }
10          Instruction (M2_R1W0) {
11          }
12          Instruction (M3_R0W1) {
13          }
14          Instruction (M4_R1W0) {
15             operation_select : ReadModifyWrite;
16             AddressCommands {
17                x1_address : decrement;
18                y1_address : decrement;
19             }
20             DataCommands {
21                write_data : data_reg;
```

```
22              expect_data : inverse_data_reg;
23            }
24         NextConditions {
25            x1_end_count : on;
26            y1_end_count : on;
27         }
28       }
29     }
30   }
31 }
```

Contrary to step 3, no inhibit_last_address_count property is needed in the preceding phase. The very next (and final) algorithm step is also run in a decrementing address order, so the BIST controller is allowed to roll back to the top once it completes this phase.

6. This final phase performs reads across the entire array:

```
1  MemoryOperationsSpecification {
2     Algorithm (MarchCMinus) {
3        TestRegisterSetup {
4        }
5        Microprogram {
6           Instruction (M0_W0) {
7           }
8           Instruction (M1_R0W1) {
9           }
10          Instruction (M2_R1W0) {
11          }
12          Instruction (M3_R0W1) {
13          }
14          Instruction (M4_R1W0) {
15          }
16          Instruction (M5_R0) {
17             operation_select : Read;
18             AddressCommands {
19                x1_address : decrement;
20                y1_address : decrement;
21             }
22             DataCommands {
23                expect_data : data_reg;
24             }
25             NextConditions {
26                x1_end_count : on;
27                y1_end_count : on;
28             }
29          }
30       }
31    }
32 }
```

The BIST controller completes the test by reading the last memory address location (that is at the bottom of the memory) and rolling back to the top location. Afterwards, test results are shifted out. Assuming everything went fine, no comparison error is reported in the test patterns.

Algorithm Optimization

The whole March C- algorithm previously described is implemented as six discrete steps and defines six separate MBIST instruction wrappers. While this example makes it easy to establish a 1:1 step <=> instruction correspondence, its implementation would, however, also require six separate microcode words to be available in a soft-programmable MBIST controller. This may be more than what is available in a silicon device.

Tessent MemoryBIST supports the use of RepeatLoopA/RepeatLoopB wrapper loops, making it simple to re-run a previously-defined algorithm portion. Steps 3, 4, and 5 could, therefore, be eliminated by repeating step 2 with only minor modifications. For more information, refer to the Instruction/NextConditions wrapper description for the RepeatLoopA and RepeatLoopB wrapper properties.

Advanced Multi-Port Testing With a UDA

The information in this section provides examples of custom algorithms and operation sets that can be applied to multi-port memories that have features not adequately covered by library algorithms, or for diagnosis purposes.

Tessent MemoryBIST supports several properties enabling you to perform operations on inactive ports of multi-port memories in parallel with the read/write operations dictated by the algorithm. These concurrent operations are controlled by the selected operation set. The sections listed below provide several examples showing the flexibility of concurrent operations, and how they can be used in custom algorithms to implement test/diagnosis sequences that might be unique to certain memory configurations or features.

Fast Test Sequences Based on SyncWRvcd

The examples outlined in this section are used to generate test sequences that apply long bursts where the address and data change on every clock cycle, while testing for potential interference between ports of a 1R1W memory. The custom algorithms use the library operation set syncWRvcd. The first example uses 0s and 1s as data patterns for controllers equipped with data registers having less than 8 bits, while pseudo-random patterns are used in the second example when larger data registers are available. Other data patterns can be used as needed.

The first instruction (WRITE_BACKGROUND) of the custom algorithm of Figure 7-8 is writing column stripes in a special way. Address register A is segmented so that the reference address is incremented by 2 instead of 1. The selected operation, WRITEREAD_COLUMN_SHADOWREADWRITE, causes a value of all 0s to be written to all even addresses using normal writes and a value of all 1s to be written to all odd addresses using concurrent writes. Refer to Figure 7-9 for a description of the instruction. The address sequence realized for the write port is 0,1,2,3,…, N-2, N-1. The address of the read port is derived from the address on the write port by inverting the least significant bit, for example 1,0,3,2,…, N-1, N-2. Both address and data change on every clock cycle on both the read and write port.

The second instruction (READMODIFYWRITE_CONCWRITEREAD) of the algorithm uses the same method to generate the address for reading back the column stripes. The difference is the address on the read port is 0,1,2,3,…, N-2, N-1 and the one on the write port is 1,0,3,2,…, N-1, N-2. The selected operation, READMODIFYWRITE_COLUMN_SHADOWWRITEREAD, implements a concurrent write during the read of the first cycle and the result is compared in the second cycle, where a concurrent read and normal write are also performed. Refer to Figure 7-9 for a description of the instruction. The result of the concurrent read at odd addresses is not compared in the first

iteration of the algorithm. This is completed in subsequent iterations where the LSB of address register A is set. The "Pseudo Concurrent Write" section shows how to compare the result of concurrent read operations in fewer iterations.

The third instruction (READ_BACKGROUND) uses address register B to read all addresses to verify that none of the memory cells were corrupted by the concurrent operations. The address changes on every other clock cycle when executing this instruction. This instruction also sets up loop A to repeat the first three instructions using the inverse data pattern.

The fourth instruction (INCR_COLUMN_ADDRESS) sets the LSB of address register A so that the reference address sequence starts at 1, but still increments by 2. This enables the comparison of the read results from odd addresses, performed concurrently with writes at even addresses that were not done in previous iterations.

Figure 7-8. Fast Test Sequence With Simple Data Patterns

```
MemoryOperationsSpecification {
  Algorithm ( ConcurrentReadWriteColumn ) {
    TestRegisterSetup {
      AddressGenerator {
        AddressRegisterA {
          number_y0_bits : 1;
          x1_carry_in    : y1_carry_out;
          y1_carry_in    : none;
          Y0CarryIn      : none;
        }
        AddressRegisterB {
          x1_carry_in : y1_carry_out;
          y1_carry_in : none;
        }
      }

      DataGenerator {
        invert_data_with_row_bit    : none;
        invert_data_with_column_bit : c[0];
      }
      operation_set_select : SyncWRvcd;
    }
```

```
MicroProgram {
  Instruction ( WRITE_BACKGROUND ) {
    operation_select    : WRITEREAD_COLUMN_SHADOWREADWRITE;
    branch_to_instruction : WRITE_BACKGROUND;
    AddressCommands {
      address_select : select_a;
      x1_address     : increment;
      y1_address     : increment;
      y0_address     : hold;
    }
    DataCommands {
      write_data : zero;
    }
    NextConditions {
      x1_end_count : on;
      y1_end_count : on;
    }
  }

  Instruction ( READMODIFYWRITE_CONCWRITEREAD ) {
    operation_select    : READMODIFYWRITE_COLUMN_SHADOWWRITEREAD;
    branch_to_instruction : READMODIFYWRITE_CONCWRITEREAD;
    AddressCommands {
      address_select : select_a;
      x1_address     : increment;
      y1_address     : increment;
    }
    DataCommands {
      expect_data : zero;
      write_data  : zero;
    }
    NextConditions {
      x1_end_count : on;
      y1_end_count : on;
    }
  }
}
```

```
Instruction ( READ_BACKGROUND ) {
  operation_select      : READ;
  branch_to_instruction : READ_BACKGROUND;
  AddressCommands {
    address_select : select_b;
    x1_address     : increment;
    y1_address     : increment;
  }
  DataCommands {
    expect_data : zero;
    write_data  : zero;
  }
  NextConditions {
    x1_end_count : on;
    y1_end_count : on;
    RepeatLoopA {
      branch_to_instruction : WRITE_BACKGROUND;
        Repeat1 {
          write_data_sequence : inverse;
          expect_data_sequence: inverse;
        }
    }
  }
}
}

Instruction ( INCR_COLUMN_ADDRESS ) {
  operation_select      : NOOPERATION;
  branch_to_instruction : WRITE_BACKGROUND;
  AddressCommands {
    address_select : select_a;
    x1_address     : hold;
    y1_address     : hold;
    y0_address     : increment;
  }
  NextConditions {
    y0_end_count   : on;
  }
}
}
```

For controllers with data registers of 8 bits or more, it is possible to replace the 1s and 0s data pattern with a pseudo-random pattern. Only three small changes are required to the algorithm shown above in Figure 7-8:

1. Specify an arbitrary seed in the DataGenerator wrapper using load_write_data and load_expect_data. The value must be the same for both data registers, and different than all 0s.

2. Modify the first instruction to make use of the pseudo-random data generator by specifying write_data : data_reg_prdg.

3. Modify the second instruction to generate the corresponding expected data value by specifying expect_data : data_reg_prdg.

Another variation of the algorithm consists of using operation READMODIFYWRITE_ROW_SHADOWWRITEREAD to perform concurrent operations on adjacent rows instead of adjacent columns. Column stripes are replaced by row stripes. The row address counter is segmented so that it increments by 2 instead of 1, and the column address counter is not segmented in this case. Note that this variation of the algorithm works on memories with or without column address bits. This is not the case of the original algorithm.

Figure 7-9 shows the main operations of the syncWRvcd operation set used by the custom algorithm. Note that properties described in a Cycle wrapper are "sticky", meaning that they retain their value (on or off) from one cycle to the next unless explicitly set to the opposite value. The only exception is strobe_data_out, which must be specified for each cycle requiring a compare.

Figure 7-9. Portion of syncWRvcd Library Operation Set

```
MemoryOperationsSpecification {
  OperationSet (syncWRvcd) {
   …
    Operation (READMODIFYWRITE_COLUMN_SHADOWWRITEREAD) {
      Cycle {
        read_enable  : on;
        write_enable : off;
        ConcurrentPortSignals {
          write_column_address : on;
          write_data_polarity  : inverse;
          read_enable          : on;
        }
      }
      Cycle {
        read_enable  : off;
        write_enable : on;
        ConcurrentPortSignals {
          read_enable          : on;
          read_column_address : on;
        }
        strobe_data_out : on;
      }
    }
```

```
Operation (READMODIFYWRITE_ROW_SHADOWWRITEREAD) {
  Cycle {
    read_enable  : on;
    write_enable : off;
    ConcurrentPortSignals {
      write_row_address   : on;
      write_data_polarity : inverse;
      read_enable         : on;
    }
  }
  Cycle {
    read_enable  : off;
    write_enable : on;
    ConcurrentPortSignals {
      read_enable      : on;
      read_row_address : on;
    }
    strobe_data_out : on;
  }
}

Operation (WRITEREAD_COLUMN_SHADOWREADWRITE) {
  Cycle {
    read_enable  : off;
    write_enable : on;
    ConcurrentPortSignals {
      read_enable         : on;
      read_column_address : on;
    }
  }
  Cycle {
    read_enable  : on;
    write_enable : off;

    ConcurrentPortSignals {
      write_column_address : on;
      write_data_polarity  : inverse;
      read_enable          : on;
      read_column_address  : on;
    }
  }
}
}
}
```

Pseudo Concurrent Write

The result of concurrent read operations are not compared to an expected value when using library operation sets. However, it is possible to do so by using custom operation sets and provide the additional observation capability on a read port by performing a concurrent read operation while performing an algorithm write.

This type of operation is called a "pseudo concurrent write", and provides additional capability for the following memory configurations:

- 2R2W — supporting concurrent write

- nRmW, where $n \neq m$ — not supporting concurrent write

The principle is the same for both cases. A strobe is added to compare the result of a concurrent read operation performed during an algorithm write. The logical port used for reading is different than the one used for writing, but are part of the same test port. Note that memory configurations including any number of RW ports do not meet this criterion, and the pseudo concurrent write concept is not applicable. This is why it is not used in library operation sets and algorithms.

Figure 7-10 shows an example of an operation implementing pseudo concurrent write. A read is performed in the first cycle at the reference address controlled by the algorithm, and the result compared to the value specified by the algorithm expect_data property. In the second cycle, a write is performed at the reference address at the same time a concurrent read is performed on all inactive read ports. The output of the logical read port that is part of the current test port is compared. In this example, the expected value is the inverse of the value specified by the expect_data property because of the invert_expect_data property being set to on.

Figure 7-10. Pseudo Concurrent Write Example

```
MemoryOperationsSpecification {
  OperationSet (my_opset) {
    SignalPipelineStages {
      strobe_data_out): 1; // Delay strobe by 1 for synchronous memories
      …
    }
    Operation (READ_WRITE_CONCREAD) {
      Cycle {
        read_enable      : on;
        write_enable     : off;
        strobe_data_out  : on;
      }
      Cycle {
        read_enable  : off;
        write_enable : on;
        AdvancedSignals {
          invert_expect_data : on;
        }
        ConcurrentPortSignals {
          read_enable      : on;
          read_row_address : on;
        }
        strobe_data_out : on;
      }
    }
  }
}
```

This operation can be used for all memories of configuration nRmW with $n \neq m$, which are not supporting concurrent write. The subset of port combinations covered by the pseudo concurrent write capability are indicated within the TSDB *<design_name>_<design_id>*_tessent_mbist_c1.generation.log file, which contains a section describing the composition of test ports for a memory. This file is located in the following TSDB folder:

```
tsdb_outdir/instruments/<design_name>_<design_id>_mbist.instrument
```

As an example, for a 2R3W memory, the test ports could be defined as follows:

```
Test Port          Logical Read/Write Ports
---------          ------------------------
0                  R1 / W1
1                  R2 / W2
2                  R1 / W3
```

The composition of the test ports is sensitive to the order of reference of logical ports in the memory TCD file. In the example above, logical port R1 was referenced before R2 and this is why it is used twice. The order can be modified to take into account the physical layout of the memory. It might not be possible to test all port combinations of interest using the pseudo concurrent write capability but the concurrent read operations without compare are always applicable and provide the same coverage as long as the location is read and its value compared before being written again. This could happen in the current or subsequent algorithm phase.

Note that all possible port combinations can be covered for nR1W and 2R2W memories. In the second case, pseudo concurrent write operations are applied in addition to the concurrent write operations already supported by the library operation set syncWRvcd. The previous example of Figure 7-10 is modified to illustrate this in Figure 7-11. The second tick is identical and exercises the combination of a read port with the write port of a same test port. A concurrent write is added in the first tick to exercise the combination of the same read port with the second write port. The other two combinations are exercised when executing the algorithm on the second test port.

Figure 7-11. Combining Pseudo Concurrent Write and Concurrent Write Operations Example

```
MemoryOperationsSpecification {
  OperationSet (my_opset) {
    SignalPipelineStages {
      strobe_data_out): 1; // Delay strobe by 1 for synchronous memories
      …
    }
    Operation (READ_CONCWRITE_WRITE_CONCREAD) {
      Cycle {
        read_enable     : on;
        write_enable    : off;
        AdvancedSignals {
          invert_write_data : on;
        }
        ConcurrentPortSignals {
          write_column_address : on;
        }
        strobe_data_out : on;
      }
      Cycle {
        read_enable  : off;
        write_enable : on;
        AdvancedSignals {
          invert_expect_data : on;
        }
        ConcurrentPortSignals {
          read_enable      : on;
          read_row_address : on;
        }
        strobe_data_out : on;
      }
    }
  }
}
```

Concurrent Write to the Reference Address

Some memories allow writing a memory location from one port while reading or writing the same location from a different port. Until the v2015.4 release, it was only possible to write and read the same location, using a custom operation that includes a normal write and concurrent read.

Figure 7-12 shows the operation of Figure 7-11, "*Combining Pseudo Concurrent Write and Concurrent Write Operations Example*", modified to illustrate this. The only change is highlighted in red, and consists of setting ConcurrentPortSignals/read_row_address to off in the second Cycle wrapper. The ConcurrentPortSignals/read_column_address property is not shown, but defaults to off as explained in a previous section. A concurrent read to the reference address is performed because ConcurrentPortSignals/read_enable is set to on, and the other two properties are set to off.

Figure 7-12. Example Operation With Write and Concurrent Read to the Same Address

```
MemoryOperationsSpecification {
  OperationSet (my_opset) {
    SignalPipelineStages {
      strobe_data_out): 1; // Delay strobe by 1 for synchronous memories
      …
    }
    Operation (READ_CONCWRITE_WRITE_CONCREAD) {
      Cycle {
        read_enable     : on;
        write_enable    : off;
        AdvancedSignals {
          invert_write_data : on;
        }
        ConcurrentPortSignals {
          write_column_address : on;
        }
        strobe_data_out : on;
      }
      Cycle {
        read_enable  : off;
        write_enable : on;
        AdvancedSignals {
          invert_expect_data : on;
        }
        ConcurrentPortSignals {
          read_enable       : on;
          read_row_address : off;
        }
        strobe_data_out : on;
      }
    }
  }
}
```

It is now possible to perform a concurrent write to the reference address while reading or writing it from another port. The property ConcurrentPortSignals/write_enable was implemented to provide this feature. The example of Figure 7-11 is further modified to illustrate the usage of this property, as shown in Figure 7-13.

Figure 7-13. Example Operation With Write and Concurrent Write to the Same Address

```
MemoryOperationsSpecification {
  OperationSet (my_opset) {
    SignalPipelineStages {
      strobe_data_out): 1; // Delay strobe by 1 for synchronous memories
      …
    }
    Operation (READ_CONCWRITE_WRITE_CONCREAD) {
      Cycle {
        read_enable      : on;
        write_enable     : off;
        AdvancedSignals {
          invert_write_data : on;
        }
        ConcurrentPortSignals {
          write_column_address : on;
        }
        strobe_data_out : on;
      }
      Cycle {
        read_enable  : off;
        write_enable : on;
        AdvancedSignals {
          invert_expect_data : on;
        }
        ConcurrentPortSignals {
          write_enable  : on;
        }
        strobe_data_out : on;
      }
    }
  }
}
```

The change is highlighted in red and consists of setting ConcurrentPortSignals/write_enable to on in the second Cycle wrapper. ConcurrentPortSignals/write_column_address and write_row_address are not explicitly set and default to off. Note that for backward compatibility reasons with previous operation sets, write_enable is inferred to on whenever write_column_address or write_row_address is set to on. The behavior of ConcurrentPortSignals/read_enable, read_row_address and read_column_address is not changed.

Note that library operation sets never attempt to write a memory location from one port while reading or writing from a different port because the result is unpredictable for most memories. If a memory does allow this type of access, a custom operation set and algorithm must be created if you want to test this feature.

Making a UDA Available to MemoryBIST

The procedures in the following sections describe how to implement a custom UDA (User-Defined Algorithm) with Tessent MemoryBIST. This task is normally performed through the DFT specification within the design flow.

A custom UDA that has been separately validated or certified can be used in larger designs. UDAs can either be hard-coded in silicon or soft-coded in microcode memory (which is implemented as flop arrays - not as SRAM).

The syntax of a UDA remains exactly the same, regardless of whether the algorithm is to be hard coded or soft-coded. However, the UDA must be provided at design time if it is to be hard-coded. The UDA can be modified when it is soft-coded, even post-silicon, and is only shifted into MemoryBIST controllers at run time.

Chapter 2, "Getting Started", covers the implementation of MemoryBIST, which by default implements (hard-codes) the built-in library algorithm specified in the Memory TCD for memories tested by a memory BIST controller. The procedures that follow focus on the additional steps required to hard-code UDAs into a design, and explain how to implement the necessary logic for the MemoryBIST controller to run soft-coded UDA.

MemoryBIST Configuration for Hard-Coded UDA

This procedure demonstrates how to implement a custom UDA with Tessent MemoryBIST that will be hard-coded in the controller RTL. This task is normally performed through the DFT specification in the design flow. The UDA can be configured to be the default algorithm for the controller, or as an algorithm that is available in addition to the default algorithm. For the latter case, both algorithms would be hard-coded in the controller RTL with this procedure.

> _____ **Note** _____
> This procedure can also be used to hard-code an additional Tessent MemoryBIST library algorithm rather than a UDA. For this implementation, Step 2 would be skipped, and the value assigned to the extra_algorithms property in Step 4 would simply be the library algorithm name.

The custom algorithm can define and make use of a custom operation set, or utilize a standard Tessent MemoryBIST library operation set. The library operation sets that are available are those built into the memory BIST controller, which by default are those specified by the memory TCD for the memories tested by the controller, unless another is explicitly specified in the DftSpecification for the controller.

Prerequisites

- Confirm the UDA has been separately certified or validated before implementing it.

- The UDA should be comprised of a MemoryOperationsSpecification wrapper, that contains the Algorithm wrapper. If a custom operation set is required, the OperationSet wrapper defining the operation set should also be present.

Procedure

1. Invoke Tessent Shell, set the tool context to dft -rtl mode, read cell libraries, set design sources, and read all HDL for your design as described in "Design Loading".

2. Read in the UDA during setup with the read_core_descriptions command, as shown in the following example:

   ```
   SETUP> read_core_descriptions \
           data/design/mem/algo_march.tcd_mem_lib
   ```

3. Proceed with a normal TS-MBIST flow as described in "Specify and Verify DFT Requirements" and "Create DFT Specification" to generate a DFT specification.

4. Edit the DFT specification, as described in "Edit/Configure the DFT Specification According to Your Requirements", and add the following properties for the memory BIST controller instance(s) you want to add the UDA:

   - AdvancedOptions/extra_algorithms

   - AdvancedOptions/extra_operation_sets

   The example below shows one method, adding the UDA named "march" and an operation set named "sync2" to memory BIST controller c1:

   ```
   ANALYSIS>set dft_spec [get_config_elements DftSpecification]
   ANALYSIS>set ctl_wrap [get_config_elements \
           memorybist/controller(c1) -in $dft_spec]
   ANALYSIS>add_config_element AdvancedOptions -in $ctl_wrap
   ANALYSIS>set_config_value AdvancedOptions/extra_algorithms \
           -in $ctl_wrap march
   ANALYSIS>set_config_value AdvancedOptions/extra_operation_sets \
           -in $ctl_wrap sync2
   ```

   a. Alternately, if you want the UDA to be the default algorithm and operation set for the controller rather than being an additional algorithm hard-coded into the RTL, change the properties of the preceding sequence as follows:

      ```
      ANALYSIS>set_config_value AdvancedOptions/algorithm \
              -in $ctl_wrap march
      ANALYSIS>set_config_value AdvancedOptions/operation_set \
              -in $ctl_wrap sync2
      ```

The resulting DftSpecification with the UDA configured as an extra algorithm is now:

```
DftSpecification(top,rtl) {
  ...
  MemoryBist {
    Controller(c1) {
      AdvancedOptions {
        extra_algorithms : march;
        extra_operation_sets : sync2;
      }
      Step {
        ...
      }
    }
  }
}
```

5. Proceed with the rest of the standard MBIST flow, as described in Chapter 2. The indicated UDA(s) is hard-coded into the MBIST controller's RTL.

6. When generating testbenches or patterns, follow the steps described in "Selecting a Hard-Coded UDA for Execution" to select and run the UDA you want to run.

Results

UDAs that are implemented with this procedure become part of the RTL of the specific memory BIST controllers that are generated during this insertion. They are thus considered as being hard-coded. From this point on, they can be invoked exactly like any built-in Tessent library algorithm.

MemoryBIST Configuration for Soft-Coded UDA

This procedure outlines how to configure a Tessent MemoryBIST controller with the necessary hardware to run a soft-coded UDA, which is shifted into the MemoryBIST controller microcode at run time.

_____Note_____

The controller also supports loading Tessent MemoryBIST library algorithms at run time that are compatible with the parameters configured in the controller's hardware. Additionally, controllers that are configured to run soft-coded algorithms can also run hard-coded algorithms configured in the controller hardware.

Prerequisites

• Determine the number of instructions you want to accommodate in the microcode memory of the soft programmable controller. The number of instructions in the UDA you plan to use should not exceed this value. The trade-off in the area required for the implementation of higher instruction counts should be carefully considered.

Procedure

1. Invoke Tessent Shell, set the tool context to dft -rtl mode, read cell libraries, set design sources, and read all HDL for your design as described in "Design Loading".

2. Continue with a normal Tessent Shell MemoryBIST flow as described in "Specify and Verify DFT Requirements" and "Create DFT Specification" to generate a DFT specification.

3. Edit the DFT specification, as described in "Edit/Configure the DFT Specification According to Your Requirements", and add an AlgorithmResourceOptions wrapper to the memory BIST controller instance(s) that you want to add the hardware to support soft-coded UDAs. The following example shows one method, where the memory BIST controller c1 will have the wrapper added:

   ```
   ANALYSIS>set dft_spec [get_config_elements DftSpecification]
   ANALYSIS>set ctl_wrap [get_config_elements \
        memorybist/controller(c1) -in $dft_spec]
   ANALYSIS>add_config_element AlgorithmResourceOptions -in $ctl_wrap
   ```

4. Specify the instruction count needed for the UDA, by setting the soft_instruction_count property as shown in the following example:

   ```
   ANALYSIS>set_config_value AlgorithmResourceOptions/\
        soft_instruction_count -in $ctl_wrap 16
   ```

   For this example, 16 instruction words are specified.

5. If your UDA specifies any of the following properties:

   - load_bank_address_min or load_bank_address_max

   - load_column_address_min or load_column_address_max

   - load_row_address_min or load_row_address_max

   Specify the soft_algorithm_address_min_max property to on, as shown in the following example:

   ```
   ANALYSIS>set_config_value AlgorithmResourceOptions/\
        soft_algorithm_address_min_max -in $ctl_wrap on
   ```

   This enables the user to specify a user-defined address count range in the soft algorithm.

6. Depending on design usage and area requirements, specify the preserve_microcode_initial_values property as needed. The following example specifies this property to off:

   ```
   ANALYSIS>set_config_value AlgorithmResourceOptions/\
        preserve_microcode_initial_values -in $ctl_wrap off
   ```

   Refer to the property description to understand the usage and area impacts and determine the proper setting for your design.

The resulting DftSpecification after completing these steps is now:

```
DftSpecification(top,rtl) {
  ...
  MemoryBist {
    Controller(c1) {
      AlgorithmResourceOptions {
        soft_instruction_count : 16;
        preserve_microcode_initial_values : off;
        soft_algorithm_address_min_max : on;
      }
      Step {
        ...
      }
    }
  }
}
```

7. Proceed with the rest of the standard MBIST flow, as described in Chapter 2. The MemoryBIST controller will have the necessary hardware to run soft-coded UDA.

8. When generating testbenches or patterns, follow the steps described in "Selecting a Soft-Coded UDA for Execution" to select and load the UDA you want to run.

Results

The MemoryBIST controller configured by the preceding steps will now have the necessary hardware incorporated to load and run a soft-coded UDA.

Selecting a UDA for MemoryBIST Execution

The procedures in the following sections describe how to run a user-defined algorithm (UDA) that was either hard-coded in a Tessent MemoryBIST controller, or is to be run on a controller that is configured to support soft-coded algorithms. This task is normally performed during the creation of the patterns specification within the design flow.

Selecting a Hard-Coded UDA for Execution

This procedure explains how to run a custom UDA that was previously hard-coded in a Tessent MemoryBIST controller. This part of the flow is normally performed using the pattern specification.

Memory BIST controllers typically test multiple memories in parallel. Each tested memory has a default algorithm associated with it, which is indicated in its memory TCD file. When a pattern specification is first created, the tool automatically considers all default algorithms and accordingly determines a suitable test plan that tests those memories in the most efficient way.

Unless a custom algorithm is defined as the default algorithm for a given memory, it is not automatically run. For this reason, you may want to explicitly instruct the memory BIST controller to run one at a given point in time.

This procedure contains the necessary steps to select a UDA (or any other Tessent Library algorithm) from the list of all hard-coded algorithms built into the memory BIST controller.

Prerequisites

- This procedure assumes the UDA has already been hard-coded in a target memory BIST controller.

- The procedure assumes a continuation of the steps outlined in "MemoryBIST Configuration for Hard-Coded UDA"; otherwise ensure the design sources are properly set and loaded.

Procedure

1. Generate a patterns specification, as described in "Create Patterns Specification".

   ```
   SETUP>set pat_spec [create_patterns_specification]
   ```

2. Edit the pattern specification, as described in "Edit/Configure the Patterns Specification According to Your Requirements", and add the following properties to the memory BIST controller instance(s) you want your UDA to run:

   - AdvancedOptions/apply_algorithm

- AdvancedOptions/apply_operation_set

The following example shown is derived from the content of the Example provided at the end of this procedure, and selects the UDA "march" and its "sync2" operation set for the first specified controller of the second test pattern:

```
SETUP>set_config_value apply_algorithm -in_wrapper $pat_spec/
    Patterns<1>/TestStep<0>/MemoryBist/Controller<0>/
    AdvancedOptions march
SETUP>set_config_value apply_operation_set -in_wrapper $pat_spec/
    Patterns<1>/TestStep<0>/MemoryBist/Controller<0>/
    AdvancedOptions sync2
```

_____ **Tip** _____

ℹ With the set_config_value command, determining the exact hierarchical names to use may sometimes be challenging. You can use introspection (get_config_elements for example) to determine the appropriate hierarchical path. You can also make use of the <N> syntax form, where <N> represents the Nth occurrence of a given wrapper in a DFT or pattern specification. Refer to the Example for more information.

3. Verify and process the patterns specification using the process_patterns_specification command:

```
SETUP>process_patterns_specification
```

4. Perform the rest of the standard MemoryBIST pattern generation flow, as described in Chapter 2. This typically means running the run_testbench_simulations and check_testbench_simulations commands.

Results

The selected algorithm and operation set (whether a UDA or any other Tessent Library algorithm), configured by the procedure described, is run by the specified memory BIST controller(s) in the wanted controller step.

Examples

The following example dofile hard-codes a UDA named "march" and its operation set "sync2" in the RTL of memory BIST controller c1. The dofile then configures the patterns specification to select the UDA for memory BIST controller c1. The UDA is now selected for BIST execution and simulated.

```
# Enter property context
set_context dft -rtl

# Set design sources and read files
set_design_sources -format verilog -y {data/mem data/rtl} -extension v
read_core_descriptions data/mem/algo_march.tcd_mem_lib
read_verilog data/rtl/blockA.v

set_current_design blockA
set_design_level sub_block

set_dft_specification_requirements -memory_bist auto
add_clock CLK -period 12ns -label clka
check_design_rules

set spec [create_dft_spec]
set_config_value extra_algorithms -in_wrapper
$spec/MemoryBist/Controller(c1)/AdvancedOptions march
set_config_value extra_operation_sets -in_wrapper
$spec/MemoryBist/Controller(c1)/AdvancedOptions sync2

process_dft_specification

extract_icl

set pat_spec [create_patterns_spec]
set_config_value apply_algorithm -in_wrapper
$pat_spec/Patterns(MemoryBist_P1)/TestStep(run_time_prog)/MemoryBist
/Controller(blockA_rtl_tessent_mbist_c1_controller_inst)/AdvancedOptions march

set_config_value apply_operation_set -in_wrapper
$pat_spec/Patterns(MemoryBist_P1)/TestStep(run_time_prog)/MemoryBist
/Controller(blockA_rtl_tessent_mbist_c1_controller_inst)/AdvancedOptions sync2

process_patterns_specification

run_testbench_simulations
check_testbench_simulations
```

The pattern specification generated from the preceding example dofile is:

```
PatternsSpecification(blockA,rtl,signoff) {
  Patterns(ICLNetwork) {
    ICLNetworkVerify(blockA) {
    }
  }
  Patterns(MemoryBist_P1) {
    ClockPeriods {
      CLK : 12.0ns;
    }
    TestStep(run_time_prog) {
      MemoryBist {
        run_mode : run_time_prog;
        reduced_address_count : on;
        Controller(blockA_rtl_tessent_mbist_c1_controller_inst) {
          AdvancedOptions {
            apply_algorithm : march;
            apply_operation_set : sync2;
          }
          RepairOptions {
            check_repair_status : on;
          }
          DiagnosisOptions {
            compare_go : on;
            compare_go_id : on;
          }
        }
      }
    }
  }
}
```

To further illustrate the <N> syntax introspection form shown in Step 2 of this procedure, the following set_config_value command (it should all appear on a single command line):

```
set_config_value apply_algorithm -in_wrapper
   PatternsSpecification(blockA,rtl,signoff)
   /Patterns(MemoryBist_P1)/TestStep(run_time_prog)
   /MemoryBist/Controller(blockA_rtl_tessent_mbist_c1_controller_inst)
   /AdvancedOptions march
```

is effectively equivalent to the following command:

```
set_config_value apply_algorithm -in_wrapper
   PatternsSpecification<0>/Patterns<1>/TestStep<0>
   /MemoryBist/Controller<0>/AdvancedOptions march
```

because the command targets the very first occurrence of "PatternsSpecification", the second occurrence of "Patterns", the first occurrence of "TestStep", and so on, of the PatternsSpecification.

Selecting a Soft-Coded UDA for Execution

The information presented in this section explains how to select and run a custom user-defined algorithm on a Tessent MemoryBIST controller that has been configured to support soft-coded algorithms. Soft-coded algorithms are shifted into the memory BIST controller at run time. The algorithm selection task is normally performed during the patterns specification creation and editing portion of the flow.

The custom algorithm can define and make use of a custom operation set, or utilize a standard Tessent MemoryBIST library operation set. The library operation sets that are available are those built into the memory BIST controller. By default, these are specified by the memory TCD for the memories tested by the controller, unless another operation set is explicitly specified in the DftSpecification for the controller.

Prerequisites

- This procedure assumes the memory BIST controller has been properly configured with the necessary hardware and appropriately-sized microcode memory to functionally support the custom UDA that is to be loaded and run. Refer to "MemoryBIST Configuration for Soft-Coded UDA" for information on how to implement the hardware.

- The UDA should be comprised of a MemoryOperationsSpecification wrapper, that contains the Algorithm wrapper. If a custom operation set is required, the OperationSet wrapper defining the operation set should also be present.

- The procedure assumes a continuation of the steps outlined in "MemoryBIST Configuration for Soft-Coded UDA"; otherwise ensure the design sources are properly set and loaded.

Procedure

1. Read in the UDA in setup mode with the read_core_descriptions command, as shown in the following example:

   ```
   SETUP> read_core_descriptions \
          data/design/mem/algo_march.tcd_mem_lib
   ```

   The algorithm now resides in memory for the tool to scan into the controller during the test pattern run.

2. Complete steps 1 and 2 in "Selecting a Hard-Coded UDA for Execution".

   The patterns specification is edited to indicate the UDA you want to run for the configured controller. The process is identical as that done for hard-coded algorithms.

3. Complete steps 3 and 4 in "Selecting a Hard-Coded UDA for Execution".

Results

The selected algorithm is configured with the procedure described and is run by the specified memory BIST controller(s) in the wanted controller step. The custom algorithm is provided to the tool after the supporting hardware had been generated and implemented in the controller. The patterns specification is edited to specify the custom algorithm to be run on this controller, in the same way as is done for selecting hard-coded algorithms. The tool creates the patterns necessary to scan in the UDA into the controller's microcode, and the test is performed.

Examples

The example dofile that follows, configures controller c1 with the necessary hardware to support soft-coded algorithms. Note that the algorithm specified in the memory TCD for the memories tested by the controller will be built into the controller RTL by default. After the hardware is generated, a UDA named "march" is read into the tool, which is to be used as the soft algorithm. The dofile then configures the patterns specification to select this algorithm for memory BIST controller c1. The UDA is now selected for BIST execution and simulated.

```
# Enter property context
set_context dft -rtl

# Set design sources and read files
set_design_sources -format verilog -y {data/mem data/rtl} -extension v
read_verilog data/rtl/blockA.v

set_current_design blockA
set_design_level sub_block

set_dft_specification_requirements -memory_bist auto
add_clock CLK -period 12ns -label clka
check_design_rules

# Configure controller c1 for soft-coded algorithms
set spec [create_dft_spec]
set_config_value soft_instruction_count -in_wrapper
$spec/MemoryBist/Controller(c1)/AlgorithmResourceOptions 16
set_config_value soft_algorithm_address_min_max -in_wrapper
$spec/MemoryBist/Controller(c1)/AlgorithmResourceOptions off
set_config_value address_segment_x0_y0_allowed -in_wrapper
$spec/MemoryBist/Controller(c1)/AlgorithmResourceOptions on

process_dft_specification

extract_icl
# Read in the UDA that is to be soft-coded into controller c1
read_core_descriptions data/mem/algo_march.tcd_mem_lib

set pat_spec [create_patterns_spec]
set_config_value apply_algorithm -in_wrapper
$pat_spec/Patterns(MemoryBist_P1)/TestStep(run_time_prog)/MemoryBist
/Controller(blockA_rtl_tessent_mbist_c1_controller_inst)/AdvancedOptions march

set_config_value apply_operation_set -in_wrapper
$pat_spec/Patterns(MemoryBist_P1)/TestStep(run_time_prog)/MemoryBist
/Controller(blockA_rtl_tessent_mbist_c1_controller_inst)/AdvancedOptions sync2

process_patterns_specification

run_testbench_simulations
check_testbench_simulations
```

Chapter 8
Tessent MemoryBIST Diagnosis

This chapter describes the diagnosis approaches supported by Tessent MemoryBIST, including a description of features and a guide for using each approach.

Only an overview on diagnosing memory failures to the address and bit level is covered in this chapter as it uses Tessent Shell SiliconInsight. Refer to *Tessent SiliconInsight User's Manual for Tessent Shell* for complete information on using this tool for memory diagnosis.

Memory BIST Diagnosis Approaches

Using memory BIST to test embedded memories provides significant advantage over the direct pin access test methods for PASS/FAIL testing. However, in most cases, it is important to identify the source of physical failure in the memory. This is referred to as the Diagnosis process.

Diagnosis Objectives

Often you are faced with a difficult decision when it comes to the diagnosis requirements of memories. This is mainly because interest is in the functional model of the memory and not the memory's physical structure. However, you need to identify the memory BIST diagnosis capability as part of the design based on manufacturing objectives and yield target.

Depending on the objectives of the diagnosis process, you need to take different design actions. For example, it might be acceptable to identify only the failing memories. This is useful for small memories when root-cause analysis is not required. In this case, you can choose a simple diagnosis approach to monitor the Pass/Fail status on every memory at the end of the test. However, if the objective is to enable offline memory repair, you need to consider another design strategy to monitor the memory test status during the test and to enable the tester to keep track of the failing address or data bit.

Disabling all diagnosis features makes it difficult to meet reasonable yield targets and manufacturing objectives and is therefore not recommended.

Diagnosis Levels

Various diagnosis levels can be implemented on the chip depending on the diagnosis objectives. The following list provides various diagnosis levels and highlights the objectives of each level:

- *Memory-Only Level*
    - Identifies the failing memory only.
    - Is useful for small memories, where no root cause analysis is required.
- *Memory Address Level*
    - Identifies the failing address in a memory.
    - Provides limited failing address mapping.
    - Is useful for applications where soft repair based on address mapping is used.

- *Offline Bit-Mapping*

  o Mapping the failure to bits in memory.

  o Useful when detailed root cause analysis is to be performed for yield improvements. Note that in this case, diagnosis time is not relevant.

Memory BIST Diagnosis Capabilities

Using memory BIST, you can use different runtime options and configuration properties to achieve the required diagnosis level. The diagnosis features in memory BIST are based on serial scan-based diagnosis.

The scan-based diagnosis topology is shown in Figure 8-1. This approach is referred to as Enhanced Stop-On-Nth-Error because the memory BIST controller is equipped with an error count register that is scan initialized before each run. When the controller has encountered a number of errors equal to that specified in the error register, it stops so that all pertinent error information (algorithm step, address, failing bit position, and so forth) can be scanned out of the controller. By iteratively scanning in subsequent error counts and running the controller, all bit fail data can eventually be extracted. The Enhanced Stop-On-Nth-Error diagnosis option can be generated by memory BIST using the failure_limit property of the DftSpecification.

Figure 8-1. Scan-based Diagnosis Approach Topology Using Enhanced Stop-On-Nth-Error Serial Scan
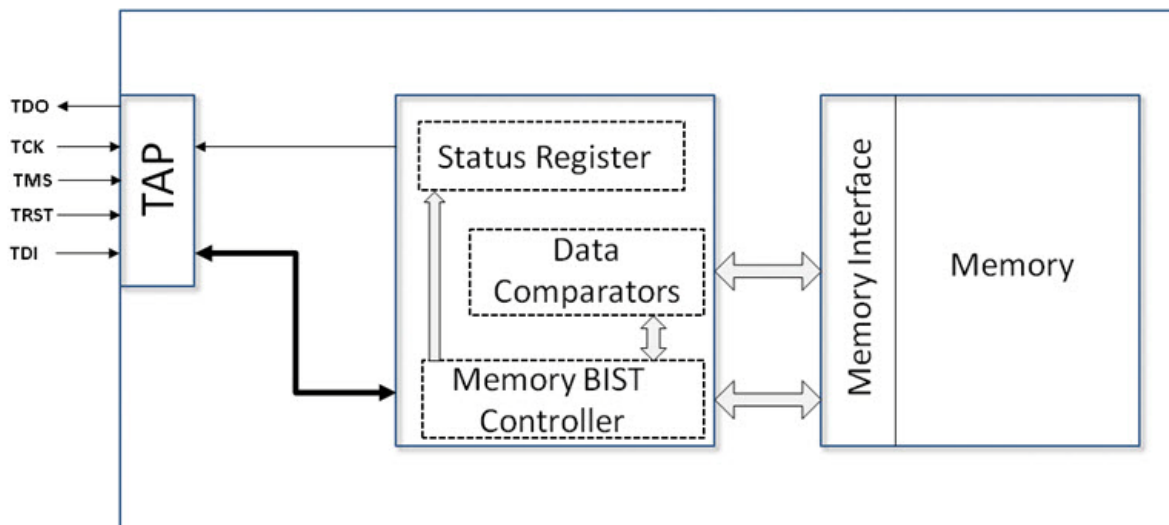
The remainder of this chapter details how to use these diagnosis approaches. In addition, this chapter shows how a specific diagnosis level can be achieved and presents a verification procedure to verify that the chosen diagnosis approach is correctly implemented on the chip.

Achieving Specific Diagnosis Level

This section provides the description of rules you can follow to simplify the decision of which diagnosis level to use and how to achieve that level.

As previously stated, various diagnosis levels can be implemented on the chip based on the diagnosis objectives. Four examples of diagnosis levels are provided in the sub-sections listed below.

Diagnosing Failing Memories Only

The objective of the Memory Only diagnosis level is to identify the failing memory in a group of small memories when no root cause analysis is required. Identifying the failing address or the BIST clock cycle number is not required within this level. You are required to identify only the failing memory, which you can achieve by either:

- using memory repair for at least one memory, in which case a status register is created for all memories with and without repair, or:

- comparing the GO_ID bits at the end of the memory BIST run if the following conditions apply:

    o The controller has either a single memory or step or is run with freeze_step set to *nn*

    o The memories use local comparators

The go_id bits are always part of the SetUp chain of the memory BIST controller. Therefore, if you need to compare the go_id bits to identify the failing memory, you must ensure that the internal Setup chain of the memory BIST controller is operational. In other words, you can correctly shift in and out of the setup chain through the TAP controller.

Using PatternsSpecification, you can create a testbench that compares the GO_ID bits at the end of the memory BIST run using the compare_go_id property.

Using the Enhanced Stop-On-Nth-Error approach might not guarantee the failing memory identifications if the controller has multiple steps and the PatternsSpecification property freeze_step is set to on. The reason is that the number of errors encountered could exceed the value specified for the DftSpecification property failure_limit.

This diagnosis strategy has minimal area overhead and minimal impact on design time. However, using this method makes it impossible to understand the reason behind the memory failure.

Diagnosing Failing Addresses and Bit-Mapping in a Memory

The objective of the Failing Address and Bit-Mapping diagnosis level is to identify the failing memory addresses and bits.

To achieve the Failing Address diagnosis level, you must ensure the following:

- The failure_limit property of the DftSpecification is set to a value large enough to capture all possible failures in any BIST step (default is 4096).

This method enables you to diagnose speed-related failures. This is because the embedded test controller does not interact with the tester, except while shifting and comparing the address register, which is done at a slow speed. The area overhead in this case is mainly because of the error counter that is not high. The design overhead is also kept minimal because the addition of the error count register is done transparently to the user.

Enhanced Stop-On-Nth-Error Approach

The Enhanced Stop-On-Nth-Error (ESOE) diagnosis approach scans out all pertinent failure data from the controller's internal registers. The memory BIST controller is equipped with an error count register that is scan-initialized before each run. When the controller has encountered a number of errors equal to those specified in the error register, it stops so that all pertinent error information (algorithm phase, address, failing bit position, and so forth) can be scanned out of the controller. By iteratively scanning in subsequent error counts and running the controller, all bit fail data are eventually extracted.

```
StopOnErrorOptions {
    failure_limit : <int> | off ;
}
```

The failure_limit property in DftSpecification adds the ESOE capabilities to the controller, including an error counter that can record up to *n* errors. The default failure limit count is 4096.

When using ESOE to diagnose memory failures, the MemoryBIST controller stops the algorithm execution and holds the current state of the controller for analysis the moment the algorithm detects a new error, and the failure counter reaches 0.

Because of the hardware implementation of the MemoryBIST controller, the error detection circuit is always a number of clock cycles behind the address registers and instruction pointers. The number of clock cycles depends on a number of hardware factors such as pipeline stages and varies based on the selected operation set and algorithm.

When ESOE data is extracted from the MemoryBIST controller, it must be processed to identify the precise memory location where the error occurred. Each MemoryBIST controller has a unique set of parameters that must be used to perform ESOE data analysis. Because of the complexity of this analysis, it is not practical to use this in simulation for diagnosis purposes. Note that the ESOE diagnosis process can be simulated along with memory fault injection. It can be used to ensure that the MemoryBIST controller is able to detect faults and stop the MemoryBIST execution. However, the raw diagnosis data from the controller setup chain may not provide useful information to the user.

The SiliconInsight debugger should be used to perform ESOE diagnosis. It contains all the algorithms needed to analyze and process the ESOE results from the MemoryBIST controllers and report the exact memory locations. Refer to *Tessent SiliconInsight User's Manual for Tessent Shell* for more information on memory failure analysis.

Several factors influence what value to specify for the failure_limit property to generate the appropriate hardware.

- **Algorithm complexity.** The number of read-compare operations performed at each memory location in an algorithm determines how many errors might be detected during diagnosis. Because the same memory location is read several times, multiple errors may

be generated depending on the type of faults. However, extracting every error is usually not necessary because most of these errors are repetitive after the first few hundred. As a reference, the total number of read-compare operations for the three most commonly used library algorithms are 14 (SMarchCHKBci), 18 (SMarchCHKBcil), and 26 (SMarchCHKBvcd).

- **Area.** When specifying a non-zero value for the failure_limit property, additional hardware is incorporated in the memory BIST controller and, if local comparators are used, in the memory interfaces. The dominant factor of this hardware is the size of the error and failure limit counters, which are proportional to the logarithm (base 2) of the failure_limit value. The number of bits of the error counter can be multiplied by 30 to obtain an approximate gate count.

- **Application.** There are two applications, memory debug and production of bitmaps during manufacturing. During memory debug, it is possible to focus on a specific memory and even on a specific port of a memory to reduce the number of errors that must be extracted. However, during manufacturing, it is desirable to log memory failures of several memories in parallel as much as possible to minimize the number of test patterns and test time. For manufacturing applications, we recommend that you specify the same value of failure_limit for all controllers in a chip. You can do this by setting the value in the DefaultsSpecification/DftSpecification/MemoryBist/ DiagnosisOptions/StopOnErrorOptions wrapper.

Performing Enhanced Stop-On-Nth-Error Diagnosis for Bitmap Applications

Performing Stop-On-Nth-Error diagnosis consists of the iterative process of running the memory BIST controller and having it stop on successive error counts so that failure data can be scanned out. As a reference, an example ESOE pattern set is shown in the figure below.

Figure 8-2. Example ESOE Pattern Set

```
Patterns(esoe) {
  ClockPeriods {
    clk1 : 20.0ns;
  }
  TestStep(init) {
    AdvancedOptions {
      split_patterns_file : off; // default
      network_end_state : keep;  // default
    }
    MemoryBist {
      run_mode : run_time_prog;
      AdvancedOptions {
        preserve_bist_inputs : on;
      }
      DiagnosisOptions {
        extract_diagnosis_data : off;
      }
      Controller(unita_i1_blka_rtl_tessent_mbist_c1_controller_inst) {
        AdvancedOptions {
          freeze_step : 1;
        }
        DiagnosisOptions {
          compare_go : on;
          compare_go_id : off;  // default
          StopOnErrorOptions {
            failure_limit : 1;
          }
        }
      }
    }
  }
}
```

```
      // The TestStep(exec) and TestStep(incr) sequence can be repeated
   TestStep(exec_1) {
     AdvancedOptions {
       split_patterns_file : on;
       network_end_state : initial;
     }
     MemoryBist {
       run_mode : run_time_prog;
       AdvancedOptions {
         preserve_bist_inputs : on;
       }
       DiagnosisOptions {
         extract_diagnosis_data : auto; // default
       }
       Controller(unita_i1_blka_rtl_tessent_mbist_c1_controller_inst) {
         AdvancedOptions {
           freeze_step : 1;
         }
         DiagnosisOptions {
           compare_go : on;
           compare_go_id : on;
           StopOnErrorOptions {
             failure_limit : auto_increment;
           }
         }
       }
     }
   }


   TestStep(incr_1) {
     AdvancedOptions {
       split_patterns_file : on;
       network_end_state : initial;
     }
     MemoryBist {
       run_mode : increment_failure_limit;
       AdvancedOptions {
         preserve_bisr_inputs : on;
       }
       Controller(unita_i1_blka_rtl_tessent_mbist_c1_controller_inst) {
       }
     }
   }
}
```

The failure limit counter has a limitation that must be taken into account when generating test patterns. The failure limit counter of the enhanced Stop-On-Nth-Error hardware does not check whether the maximum count specified with the DftSpecification property failure_limit (default is 4096) is reached. If the BIST controller is run a number of times so that this maximum count is exceeded, the counter wraps around and starts counting from 0. The same failures are extracted again, if any are present. This behavior might cause an unnecessary increase in the time required to extract failure information because it appears that new failures are present when, in fact, they are old failures. The problem is more likely to occur when several controllers

on the same chip have been generated with a different value for the failure_limit property, and they are run in parallel. To minimize the impact of this limitation, do the following:

- Specify the same value for the failure_limit property for all controllers on a circuit. Do this by setting the value in the DefaultsSpecification.

- If using the same value for all controllers is not possible, only run controllers with the same value in parallel.

- Set the maximum number of iterations in the diagnosis program to be equal to the failure_limit value minus 1.

For the reasons noted in the "Enhanced Stop-On-Nth-Error Approach" section, the SiliconInsight debugger should be used to perform ESOE diagnosis. It contains all the algorithms needed to analyze and process the ESOE results from the MemoryBIST controllers and report the exact memory locations. Refer to *Tessent SiliconInsight User's Manual for Tessent Shell* for more information on memory failure analysis.

This chapter describes various flows, or methods, of how to use Tessent MemoryBIST to implement the memory test solution you want in a design. It explains and provides examples for the most common flows that are generally used for creating and inserting the memory BIST logic. This topic is covered at a higher level and the detailed implementation specifics found in other sections of this manual and the *Tessent Shell Reference Manual* are not repeated.

The tool offers the following flows related to the memory BIST creation and insertion phase:

- Top-Down Flow

- Bottom-Up Flow

Topics to consider when choosing the best implementation flow to use include physical partitioning information and the use of power domains. These items must be carefully accounted for when going through the flow. It's also important to know if all the design information is complete and available, or if the memory BIST is implemented in pieces as the design information is completed.

Top-Down Flow

In the top-down implementation flow you perform both memory BIST generation and insertion activities in one flow through the tool for the whole design from the top level. This approach is generally not used very often unless the design is not very large, or if the design is implemented at the gate level and is completed. For large SoC designs created in a block-level approach, the bottom-up flow is typically used instead of this method.

To use the top-down flow approach, you use the high-level flow sequence of steps explained in the Getting Started chapter and illustrated in Figure 2-1. All of the design and library files for the whole design are required before beginning the flow.

For example purposes, the block-based design diagram shown in Figure 9-1 illustrates a hierarchical design that includes memories spread throughout the design. The top level of the design includes two identical cores. Each core has two copies of blockA and one of blockB. Within blockA there are two different power domains with an island configuration as illustrated.

Figure 9-1. Hierarchical Design Example With Many Memories
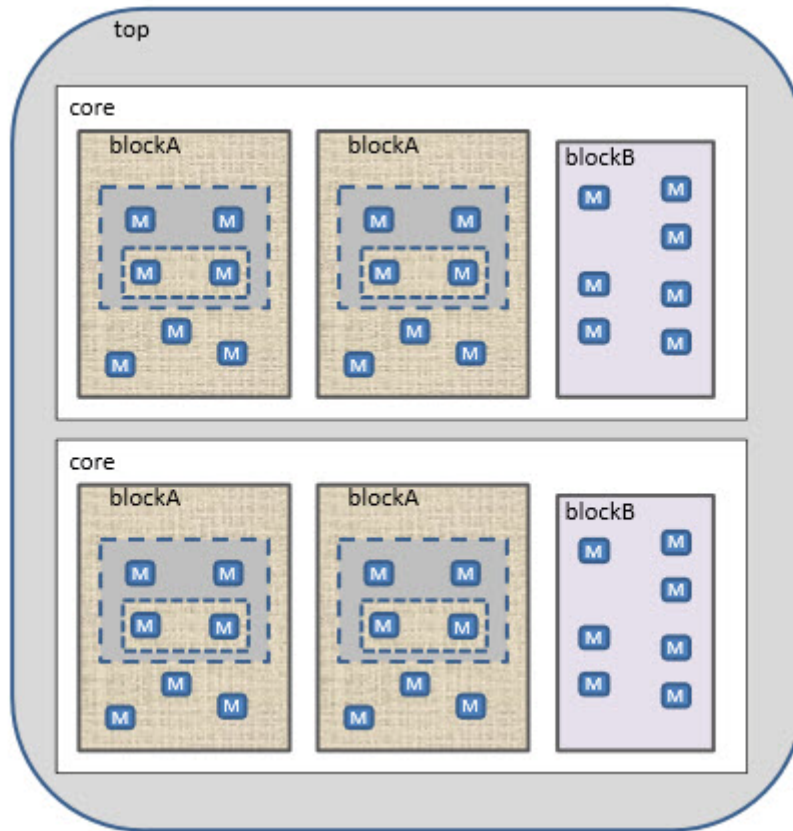
To implement the top-down MBIST flow method with this design requires all of the correct design, library, physical implementation (DEF), and power domain files (CPF or UPF) as a prerequisite. Then the high level flow sequence from the Getting Started chapter is used just once with the design level set to chip. This method creates and inserts all of the necessary memory BIST controllers and other logic all at once for the whole design. The single created TSDB is all-inclusive with the top-down flow because all the work is done from the chip level. A completed block-based design diagram illustrating the addition of the memory BIST controllers is shown in Figure 9-2.

Figure 9-2. Hierarchical Design Example With MemoryBIST Controllers
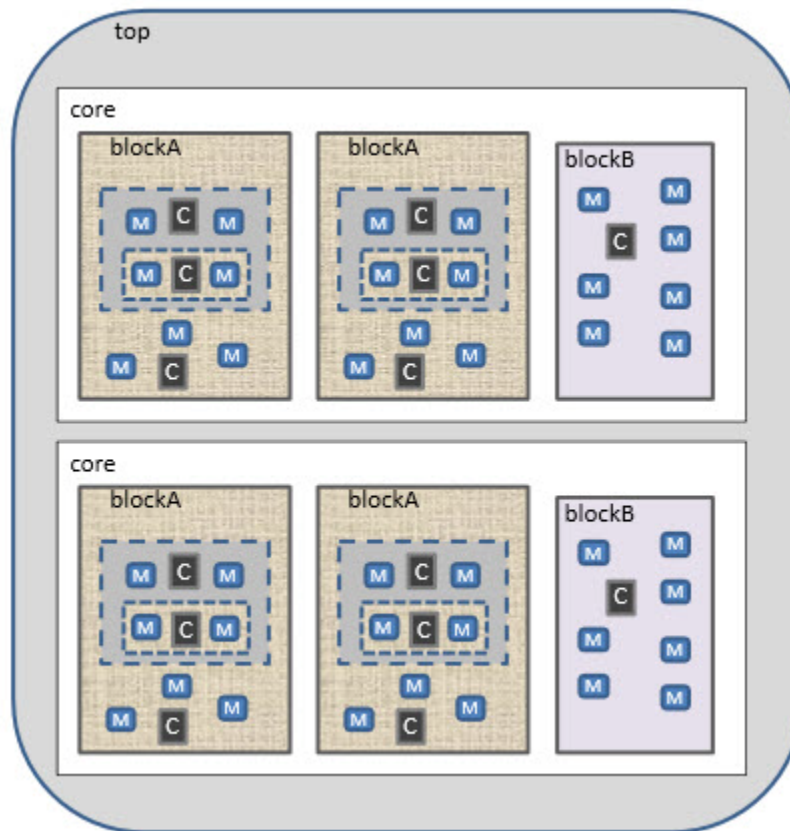
As shown in the diagram, memory BIST controllers are automatically placed where needed, taking into account the power domains and physical configuration information that was provided. For example, within blockA three controllers are needed because of the different power regions in that area of the design. Each power area requires a separate BIST controller.

After the BIST controllers are created and inserted, the patterns and testbenches are created and then simulated. If any issues are found that need correcting, then make those changes and re-run the flow from the beginning.

Bottom-Up Flow

The bottom-up implementation flow performs memory BIST generation and insertion in multiple stages, generally based on the different blocks in a design. This is a common approach for adding memory BIST to large SoC designs that are designed in functional/logical/physical blocks.

There are benefits to using this bottom-up approach that include flexibility, limited scopes to work with, and the ability to implement memory BIST for completed blocks while the whole design is not yet completed. It also facilitates a re-use opportunity where a completed design block with BIST can be re-used in other designs. Many large SoCs are designed by different

groups working on the variety of blocks. This bottom-up, or block-based approach enables each group to complete their design and memory BIST work for their block(s) independent of the other groups.

There are basically two ways to implement the bottom-up flow within Tessent Shell. If all the design blocks are completed and the design files are available, then you could invoke Tessent Shell once and use a bottom-up flow sequence where you first set the design current level to the lowest sub_block and implement memory BIST for it. Then repeat for the other sub_blocks by changing the current design focus and re-running the flow sequence of steps explained in Getting Started for each block. After all of the lowest blocks are done, then move to the next higher level of hierarchy and incorporate the completed blocks into that level and add any needed memory BIST for that level. Continue in this manor until all of the levels of hierarchy are completed, then at the top level bring it all together with design level set to chip. If this approach were used to implement memory BIST for the example design shown in Figure 9-1, then the major steps to use would include those shown in Figure 9-3.

This one tool invocation methodology creates one large Tessent Shell Data Base (TSDB) that contains all of the data, but within that TSDB the results from the separate steps for the blocks and cores are separated. An example and description of this database structure is provided in the "Tessent Shell Data Base (TSDB)" chapter of the *Tessent Shell Reference Manual*.

The other, and likely more common, method to implement a bottom-up flow is to invoke Tessent Shell and create the memory BIST for each block as you are able to. This approach provides the most flexibility in working with different design completion schedules for the blocks. For each block you create a separate TSDB with the data needed for that block. If any changes are required that affects any block, then just redo the BIST implementation for that block that needs it.

Once higher levels of the design are ready, then incorporate the lower level blocks as you add any needed BIST logic at the higher level. This creates a TSDB for the higher level. The steps are basically the same as shown in Figure 9-3, but are done with separate invocations of Tessent Shell instead of all at once.

If you use the same invocation location when starting your different Tessent Shell sessions, then all of the database information for the different design pieces are placed under one large TSDB in a single root directory. Another way to accomplish this is to use the set_tsdb_output_directory command with each session to specify the TSDB output directory that you would like to use.

Figure 9-3. Steps to Implement MemoryBIST for Example Design in One Tool Invocation
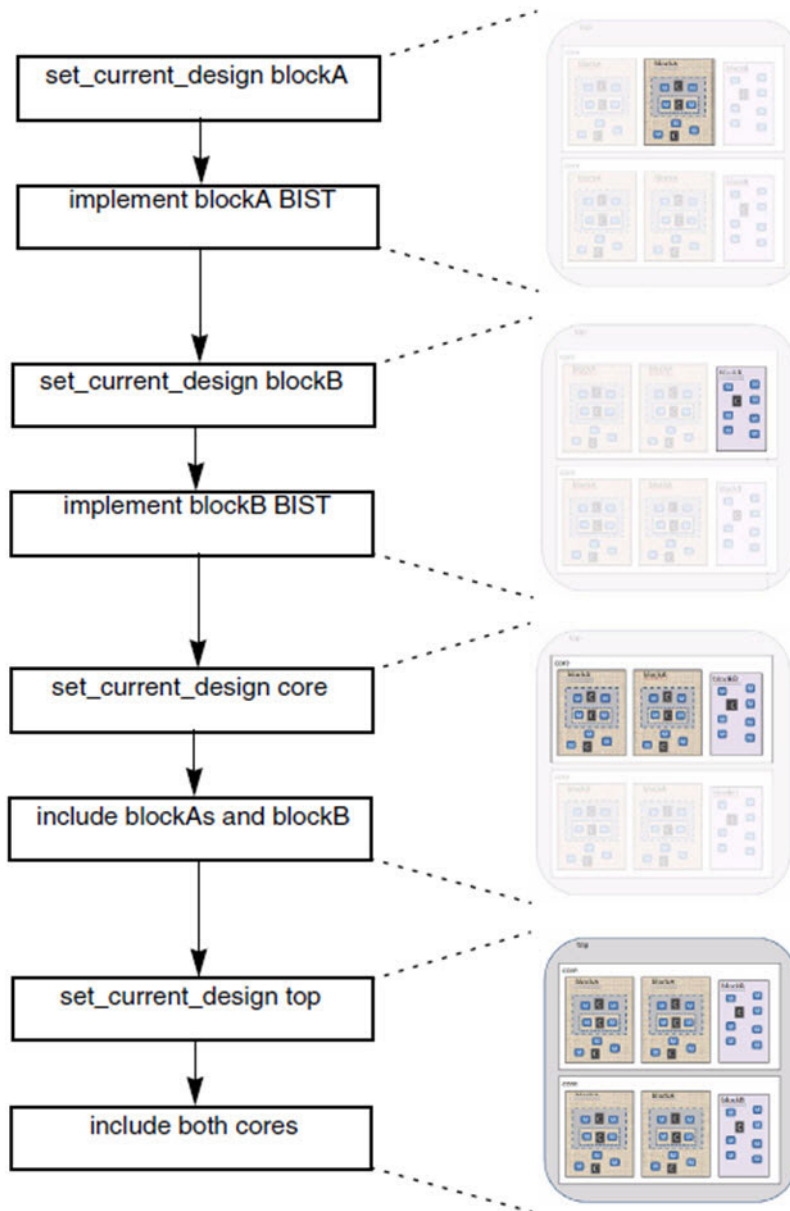
It is also fine if you want to create separate TSDBs for your different blocks and cores because it does not matter if everything is located in one area. When integrating lower-level blocks at the top or chip level, the open_tsdb command makes the contents of the specified TSDB directories visible to the tool.

Appendix A
Tessent Core Description

This chapter describes the configuration data syntax used to describe the following macro module types: memories, boundary scan segments, and fuseboxes.

This appendix uses the following syntax conventions when documenting wrappers and properties used in the library descriptions.

Table A-1. Conventions for Command Line Syntax

Convention	Example	Usage
UPPercase	-STatic	Required argument letters are in uppercase; in most cases, you may omit lowercase letters when entering literal arguments, and you need not enter in uppercase. Arguments are normally case insensitive.
Boldface	set_fault_mode **Uncollapsed** | **Collapsed**	A boldface font indicates a required argument.
[ ]	exit [-force]	Square brackets enclose optional arguments. Do not enter the brackets.
Italic	dofile *filename*	An italic font indicates a user-supplied argument.
{ }	add_fault_sites {**-ALl** | **-UNDEFINED_Cells** }[-VERBose]	Braces enclose arguments to show grouping. Do not enter the braces.
|	add_fault_sites {**-ALl** | **-UNDEFINED_Cells** }[-VERBose]	The vertical bar indicates an either/or choice between items. Do not include the bar in the command.
Underline	set_au_analysis **ON** | **OFf**	An underlined item indicates either the default argument or the default value of an argument.
…	add_clocks *off_state primary_input_pin*… [-Internal]	An ellipsis follows an argument that may appear more than once. Do not include the ellipsis when entering commands.

Table A-2. Syntax Conventions for Configuration Files

Convention	Example	Usage
Italic	***scan_in :*** *port_pin_name*;	An italic font indicates a user-supplied value.
Underline	wgl_type : generic | lsi;	An underlined item indicates the default value.

Table A-2. Syntax Conventions for Configuration Files (cont.)

Convention	Example	Usage
|	logic_level : <u>both</u> | high | low;	The vertical bar separates a list of values from which you must choose one. Do not include the bar in the configuration file.
...	port_naming : *port_naming*, ...;	Ellipses indicate a repeatable value. The comment "// repeatable" also indicates a repeatable value.
//	// default: ijtag_so	The double slash indicates the text immediately following is a comment and tells the tool to ignore the text.

Core

In Tessent Shell, descriptions of main elements, like the memory library, the boundary scan information, or the fuse box interface, are presented to the tool in form of TCD files (Tessent Core Description files). After loading, they are hierarchically organized under the Core root entry, which is unique for a given module name.

Syntax

```
Core(module_name) {
    Memory {
    }
    MemoryCluster {
    }
    BoundaryScan {
    }
    FuseBoxInterface {
    }
}
```

Description

The Core wrapper collects all TCD data read into the tool. Such descriptions are automatically read in during module matching. See the set_design_sources -format tcd_memory command description for information about where they are looked for. See the read_core_descriptions command description to learn how to read them in explicitly.

You can also report on the loaded TCD information. You do this using the report_config_data command. An example is "report_config_data Core(ModuleName)/Memory -partition tcd", which report the contents of the Memory entries under Core. To see the supported syntax, use the report_config_syntax command, for example "report_config_syntax [get_config_value Core/Memory -partition meta:tcd -object]". Because the tool automatically looks up the metadata, you can specify only the wrapper names, such as "report_config_syntax Core/Memory".

Parameters

- *module_name*

  The name of the module, equivalent of the current design module name. You do not need to specify this when loading a memory TCD file. The tool with auto-generate and auto-configure the Core-level wrapper for you.

Related Topics

set_design_sources [Tessent Shell Reference Manual]

read_core_descriptions [Tessent Shell Reference Manual]

report_config_data [Tessent Shell Reference Manual]

report_config_syntax [Tessent Shell Reference Manual]

set_module_matching_options [Tessent Shell Reference Manual]

Memory

Specifies the memory behavior for the specific module_name.

Syntax

```
Core(module_name) {
  Memory {
    Algorithm                          : algo_name;
    ATD                                : on | off;
    BitGrouping                        : int | auto;
    ConcurrentRead                     : on | off;
    ConcurrentWrite                    : on | off;
    DataOutHoldWithInactiveReadEnable  : on | off;
    DataOutStage                       : none | StrobingFlop
    InternalScanLogic                  : on | off;
    LogicalPorts                       : nRnWnRW;
    MemoryHoldWithInactiveSelect       : on | off;
    MemoryType                         : rom | sram | dram;
    MilliWattsPerMegaHertz             : real | auto;
    MinHold                            : time; // default: 0
    NumberOfBits                       : int | auto;
    NumberOfWords                      : int | auto;
    ObservationLogic                   : on | off;
    OperationSet                       : operation_set_name;
    PipelineDepth                      : int; // default: 0
    ReadOutOfRangeOK                   : on | off;
    RetentionTimeMax                   : time | none;
    RomContentsFile                    : file_path_name;
    ShadowRead                         : on | off | auto;
    ShadowWrite                        : on | off;
    ShadowWriteOK                      : on | off;
    TransparentMode                    : syncmux | none | asyncmux;
    Port(port_name) {
    }
    AddressCounter {
    }
    PhysicalAddressMap {
    }
    PhysicalDataMap {
    }
    GroupWriteEnableMap {
    }
    RedundancyAnalysis {
    }
    IclPorts {
    }
  }
}
```

Description

The Memory TCD describes the memory behavior for the specified module_name and is automatically read in during module matching. See the set_design_sources -format tcd_memory command description for information about where they are looked for. See the read_core_descriptions command description to learn how to read them in explicitly. See the

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

set_module_matching_options command description for information about the name matching process.

_____ **Note** _____

The legacy LogicVision memory library format is supported natively and is automatically translated into this format when read. You only need to read one memory description. Note that Tessent Shell requires the LogicVision MemoryTemplate name to match the specified CellName as shown below:

```
MemoryTemplate(mydram) {
      MemoryType    : SRAM;
      CellName      : mydram;
       .
       .
       .
}
```

To see the content of a read-in Core(ModuleName)/Memory, use the "report_config_data Core(ModuleName)/Memory -partition tcd" command.

To see the supported syntax, use the "report_config_syntax Core/Memory" command.

Parameters

- Algorithm : *algo_name*;

  A property that identifies the type of the default algorithm that the memory BIST controller uses to test the memory. You can specify a library algorithm name or a custom algorithm name. The available library algorithms are shown below, with SMarchCHKBci as the default:

SMarchCHKBci	SMarch
SMarchCHKB	ReadOnly
SMarchCHKBcil	SMarchCHKBvcd
LVMarchX	LVMarchY
LVMarchCMinus	LVMarchLA
LVRowBar	LVColumnBar
LVGalPat	LVGalColumn
LVGalRow	LVCheckerboard1X1
LVCheckerboard4X4	LVWalkingPat
LVBitSurroundDisturb	LVAddressInterconnect
LVDataInterconnect	

  For the details on the Algorithm values, refer to "MemoryBIST Algorithms" in this manual.

These usage conditions apply:

- o The ReadOnly algorithm can only be used for MemoryType ROM.

- o A ROM memory type can only use the ReadOnly algorithm.

- o You can override the algorithm selection by setting values in DefaultsSpecification/ DftSpecification/MemoryBist/ControllerOptions, in DftSpecification/MemoryBist/ Controller/AdvancedOptions, or in DftSpecification/MemoryBist/Controller/Step.

- o A custom algorithm name might be specified. However, it is recommended that one of the following library algorithms is specified to enable the automatic generation of the parallel retention test: SMarchCHKB, SMarchCHKBci, SMarchCHKBcil or SMarchCHKBvcd.

- o The SMarchCHKBvcd algorithm performs specialized tests on the chip select and read enable ports. To use this algorithm, the memory data output value must be preserved when the chip select or read enable port is deasserted.

  _____ **Note** _____
  You cannot specify a different algorithm for two instances of the same memory. To specify different algorithms, create a new memory template and specify a different algorithm.

- • ATD : on | off;

  A property that supports address transition detection (ATD). The default value is off. A setting of on forces the address of the memory to change so that data can be read out of the memory being tested. Use this property for memories that require an address transition to initiate a read cycle.

  These usage conditions apply:

  - o The OperationSet property must specify an ATD waveform.

  - o If your address counter is not segmented, the value specified for NumberOfWords must be even.

  - o If your address counter is segmented into a row address (AddressCounter: Function (rowAddress)) and a column address (AddressCounter: Function (columnAddress)), you must specify an even value for lowRange and an odd value for highRange in Function (columnAddress): CountRange.

  The restrictions relating to the address counter prohibit an out-of-range address that is caused by inverting bit0 of the row address counter when the ATD waveform is on.

- • BitGrouping : *int* | auto;

  A property that specifies the distribution of the data bits in the memory array. The default value is auto. The specified integer value must be between 1 and the value specified for the NumberOfBits property. The auto setting resolves to 1 when the memory address port has at

least one column address bit, otherwise it will resolve to the value of the NumberOfBits property.

Modern memories are designed to form physical arrays of cells with minimum layout spacing design rules to reduce area. Memory designers also use different strategies to achieve higher performance by manipulating the way the memory bits are distributed in the physical array. Some memories might be designed to allow the data bit of a word be next to each other in the array. Other memories might have them distributed in the array in a systematic way.

The BitGrouping property enables you to specify the distribution of the data bits in the memory array. This information is required by some Tessent MemoryBIST algorithms to construct a contiguous checkerboard pattern within the bit arrays. Specifically, the following algorithms use this property during the checkerboard phases:

- o SMarchCHKB

- o SMarchCHKBci

- o SMarchCHKBcil

- o SMarchCHKBvcd

During the checkerboard phases of the algorithm, the BIST data written and read from the memory is modified at the memory interface to ensure the memory contains checkerboard patterns.

Refer to Example 2, Example 3, and Example 4 for examples of BitGrouping usage.

- ConcurrentRead : on | off;

A property that enables you to perform simultaneous read operations on inactive read ports during both the read and write cycles of the active port controlled from the selected algorithm. The default setting is off. Setting this property to on inserts logic in the memory interface circuit to support concurrent read operations controlled from the selected operation set. This is useful when multi-port memories exhibit defects related to inter-port interaction and require special test algorithms that enable the sensitization and detection of those defects. ConcurrentRead provides more flexibility than ShadowRead. ConcurrentRead enables modification of both the row and column address from the operation set, and is therefore preferred when creating custom operation sets for a programmable controller. ShadowRead only enables modification to the row address.

This property is not supported for ROM or 1RW memories.

- ConcurrentWrite : on | off;

A property that enables you to perform simultaneous write operations on an inactive write port during both the write and read cycles of the active port controlled from the selected algorithm. The default setting is off. Setting this property to on inserts logic in the memory interface circuit to support concurrent write. This is useful when multi-port memories exhibit defects related to inter-port interaction and require special test algorithms that enable the sensitization and detection of those defects. ConcurrentWrite provides more flexibility

than ShadowWrite. ConcurrentWrite enables modification of both the row and column address as well as the data pattern from the selected operation set, and is therefore preferred when creating custom operation sets for a programmable controller. ShadowWrite is only used by library algorithms and is not controllable from the operation set. ConcurrentWrite can be used on multi-port memories with up to two ports with write capability.

The memories must have ONE of the following combinations of ports:

- o   Two ReadWrite ports

- o   One Read-only port and one Write-only port

- o   Any number of Read-only ports and two Write-only ports

- DataOutHoldWithInactiveReadEnable : <u>on</u> | off;

A property that specifies whether the memory output is preserved when the read enable control signal is inactive. The default value is on.

If the memory output does not hold when the read enable control signal is inactive and you want to test the memory with the SMarchCHKBvcd algorithm, you must set the DataOutHoldWithInactiveReadEnable property to off.

- DataOutStage : <u>none</u> | StrobingFlop ;

A property that controls whether or Tessent MemoryBIST uses a strobing flip-flop on output data to meet the hold time in a memory that cannot safely write the data from a previous read operation. Specifying a strobing flip-flop effectively provides one stage of pipelining between the data output and the comparator logic.

Valid values are as follows:

none	Does not add a flip-flop to capture the output data
StrobingFlop	Inserts a flip-flop to strobe the memory output. This property is used when the data output of the memory is not latched or registered. For scan testing, the strobing flip-flop is reused to implement the bypass mode specified by the TransparentMode: syncmux setting.

- InternalScanLogic : on | <u>off</u>;

A property that specifies the memory module containing scan circuitry that is reused during scan test modes. The default value is off. The supported internal scan logic is the bypass logic between the data input and the data output of the memory. If the bypass logic includes flip-flops on the data input side, the flip-flops must be stitched into a scan chain.

When InternalScanLogic is set to on, Tessent MemoryBIST does not generate a scan model for the associated memory module. The memory scan model must be provided to the tool.

You must set TransparentMode to none when InternalScanLogic is set to on.

Set Core/Memory/ObservationLogic to off if the internal scan logic includes flip-flops that observe the address and control inputs and those flip-flops are part of the internal scan chain of the memory.

* LogicalPorts : nRnWnRW;

A property that identifies the configuration of read, write, and read/write logical ports for a memory. Logical ports require a unique address bus for reading, writing, or both.

Valid values are as follows:

nR	Identifies the number of Read logical ports.
nW	Identifies the number of Write logical ports.
nRW	Identifies the number of ReadWrite logical ports.

The default value for this property depends on the MemoryType property, as follows:

o If MemoryType is sram this property defaults to 1RW.

o If MemoryType is rom, this property defaults to 1R.

The following usage conditions apply:

o If MemoryType is rom, you cannot specify W or RW.

o If MemoryType is sram, you must specify at least one R and one W or one RW.

* MemoryHoldWithInactiveSelect : <u>on</u> | off;

The on setting indicates that, when the select control signal is inactive, the memory content is preserved for a write operation and the memory output is preserved for a read operation.

If the memory contents or output does not hold when the select control signal is inactive and you want to test the memory with the SMarchCHKBvcd algorithm, you must set MemoryHoldWithInactiveSelect to off.

* MemoryType : rom | <u>sram</u> | dram;

A property that specifies the type of memory. The default value is sram. Valid values are as follows:

rom	Specifies a read-only memory
sram	Specifies a static random access memory.
dram	Specifies a dynamic random access memory.

- MilliWattsPerMegaHertz : *real* | <u>auto</u>;

  A property that defines the amount of power consumed by the memory in relation to the operational frequency. This value represents the average between the read and write power consumption. Memory BIST performs approximately equal number of read and write operations. This value, which typically is from the data sheet provided by the memory supplier, enables you to manage the power distribution and consumption of the chip.

  By default, the auto option estimates the value as follows:

  b * (0.0004 + (c * 0.00008))

  Where:

  - *b* is the number of bits per word inferred from the NumberOfBits property.

  - *c* is the number of columns or column mux option, inferred from AddressCounter/Function/CountRange of the ColumnAddress, which in turn can be inferred from the column address segment width.

- MinHold : *time*; // default: 0

  The MinHold property provides delay on all address, data, and control input signals from the memory BIST controller with respect to the memory clock during RTL simulation. Specify MinHold when the minimum hold requirements of the memory models are greater than zero. The value can be either an integer or a real number with up to two digits of accuracy following the decimal point. A value of zero, which is the default, does not affect the simulation.

  If you use the MinHold property for a memory, keep in mind that delay is being added in the RTL code that does not translate through synthesis. The delays added in the RTL code to meet a hold-time requirement are ignored by the synthesis tool, and the inherent gate and wire delay in the circuit may or may not be sufficient to meet the hold-time requirement on the memory. The specified value must be greater than or equal to zero and less than the specified clock period. The time unit for MinHold is nanoseconds.

  _____ **Note** _____

  The specified MinHold value should not be equal to or greater than the clock period associated with the memory and its memory BIST Controller. Otherwise, the tool reports an error.
  _____

- NumberOfBits : *int* | <u>auto</u>;

  The NumberOfBits property specifies the number of bits per word in the memory. If the NumberOfBits property is not specified, the default value is the output data width of the first logical port defined in the memory library.

  For example, for a 32X8 memory (depth of 32 words and width of 8 bits), the entry for NumberOfBits would be 8.

- NumberOfWords : *int* | auto;

The NumberOfWords property specifies the number of words in the memory. The NumberOfWords property defaults to the product of the number of columns, rows, and banks defined in the AddressCounter wrapper. If an AddressCounter/CountRange is specified for a particular address segment, that AddressCounter/CountRange is used in the formula. If no AddressCounter/CountRange is specified, the address segment size is assumed to be 2^n, where *n* is the number of bits in that address segment.

> **Note** _____
>
> Siemens EDA recommends specifying the NumberOfWords property for non-segmented addresses and for memories in which the actual number of words is not a power of 2. Not specifying NumberOfWords might cause the simulation to fail. For example, if you do not specify NumberOfWords for a memory with 12 words and four address bits, the address counter counts to 16 although there are only 12 words.

Tessent MemoryBIST controller supports testing of memories with asymmetric banks containing a different number of rows per bank. In this case, you must specify an AddressCounter/CountRange for each existing memory address segment in the memory library file. Also, you must specify the NumberOfWords property indicating the valid memory address range is 0 to NumberOfWords -1.

For example, for a 32X8 memory (depth of 32 words and width of 8 bits), the entry for NumberOfWords is 32.

- ObservationLogic : on | off;

The ObservationLogic property specifies whether or not Tessent MemoryBIST adds scan observation points for address and control signals in the memory interface.

A value of on adds sample points by means of XOR gates and flip-flops to the address and control signals of the memory within the interface. Tessent MemoryBIST either uses existing flip-flops within the interface for these sample points, or if necessary, adds additional flip-flops with the XOR gates. A value of off omits sample points from the interface.

- OperationSet : Async | AsyncWR | ROM | (Sync) | SyncWR | SyncWRvcd | TessentSyncRamOps | TessentSyncRamOpsHR4 | TessentSyncRamOpsHR6 | *OperationSetName*;

The OperationSet property specifies the name of the operation set that the memory BIST controller uses to generate waveforms that drive the memory. The operation set that you specify must define the operations that are required by the algorithm testing this memory.

Valid values are as follows:

  o Async | AsyncWR | ROM | (Sync) | SyncWR | SyncWRvcd | TessentSyncRamOps | TessentSyncRamOpsHR4 | TessentSyncRamOpsHR6 — Reserved strings that specifies a library operation set. Refer to "Tessent MemoryBIST Library Operation Sets" for a description of each operation set.

> **Tip**
> Siemens EDA recommends using the library operation sets as templates to create your own operation sets. The library operation sets are generic and not optimized for one type of memory. Custom operation sets might be required to accommodate specific timing requirements or modes of operation.

  o *OperationSetName* — is a user-defined identifier that matches the name of an operation set defined by the OperationSet wrapper. If you specify a user-defined name for the OperationSet property, Tessent MemoryBIST searches the specified memory library files for an OperationSet wrapper with the same identifier.

- PipelineDepth : *int*; // default: 0

  The optional PipelineDepth property declares the number of cycles to delay the compare on the read data. The adjustment is memory specific and typically is used to handle memories with built-in pipelining. The compare on the read data is enabled by the strobe_data_out property in the OperationSet/Operation/Cycle wrapper.

  In the operation set used with this memory, the position of strobe_data_out is pipelined by the specified number of stages. Using the PipelineDepth property enables you to apply a common operation set to multiple memories having different pipelining and to customize the delay per memory type.

  Note that all memories that are grouped in the same step must have the same PipelineDepth value. Example 1 provides additional detail on the use of PipelineDepth.

- ReadOutOfRangeOK : on | off;

  The ReadOutOfRangeOK property suppresses the CountRange rule checking when ShadowRead is set to on. This enables you to use the shadow read cycle even when the low CountRange for your RowAddress is odd or the high CountRange for your RowAddress is even. The data from the out-of-range read does not contribute to the BIST results. Set this property to on only when your memory can tolerate an out-of-range read without damaging the memory logic or corrupting the memory data.

- RetentionTimeMax : time | none;

  The RetentionTimeMax property specifies the upper limit of retention time between two full refreshes of a DRAM array. Therefore, the real refresh interval between two consecutive refresh operations applied to the memory is computed as RetentionTimeMax divided by the maximum number of memory rows. The RetentionTimeMax value is also used to size the delay counter so that all the retention time values you want can be accommodated. The default value is none.

- RomContentsFile : *file_path_name*;

  The ROMContentsFile property specifies the name of the ROM contents file. The ROM contents file is a hexadecimal or a binary listing of the ROM contents. Each line in the ROM contents file represents a single address location within the ROM starting with address 0.

You can override the ROM content file name using the rom_content_file property in
DftSpecification/MemoryBist/Controller/Step/MemoryInterface wrapper.

The entire ROM contents file should be specified in hexadecimal or binary. The two formats
cannot be mixed within the same ROM contents file. The number of hexadecimal or binary
digits per line must be consistent with the width of the ROM locations. For example, each
entry for a 12-bit wide ROM must be specified exactly as three hexadecimal digits (000 to
3FF) or 12 binary digits (000000000000 to 111111111111).

The example ROM contents file shown below is in hexadecimal format for a 16-bit wide
ROM. Because the last entry in the ROM contents file is FFFF, all subsequent address
locations assume the FFFF data value.

```
0060
E896
0000
0000
0000
0000
0000
0000
C010
01C6
0000
0000
C010
0212
0000
0000
C010
025E
FFFF
```

The example ROM contents file shown below in binary format is for a 6-bit wide ROM.
Because the last entry in the ROM contents file is 101111, all subsequent address locations
assume the 101111 data value.

```
001000
011011
000110
000110
001111
101000
101011
101010
101111
```

- ShadowRead : on | off | auto;

  The ShadowRead property enables and disables the shadow read cycle. To detect shorts
  between multiple logical ports, the memory BIST controller performs a shadow read on
  inactive read ports during both the write and the read cycles of the active ports.
  ShadowRead does not allow to read a memory cell located in a different column when

running algorithms on a programmable controller using a custom operation set. Use ConcurrentRead for greater flexibility.

A value of on enables the shadow read by inverting bit0 of the row address counter. A value of off disables the shadow read. The default value is auto and its value (on or off) is determined by Tessent MemoryBIST as follows:

- ShadowRead is off, if the OperationSet section in the memory library file does not specify the ConcurrentPortSignals/read_row_address and ConcurrentPortSignals/read_enable waveforms.

- ShadowRead is off, if you do not specify a row address. For information on specifying a row address, refer to the AddressCounter wrapper.

- ShadowRead is off, if you use the default address counter and the value specified for the NumberOfWords property is odd.

- ShadowRead is off, if either lowRange for Function(RowAddress)/CountRange is an odd value or highRange for Function(RowAddress)/CountRange is an even value. You can suppress this rule by setting the ReadOutOfRangeOK property to on.

- ShadowRead is off, if you specify LogicalPorts: 1RW or 1R.

> **Note**
> When ShadowWrite is on the ShadowRead property is turned to on as well, if the requirements are met.

The restrictions relating to the address counter prohibit an out-of-range address, that is caused by inverting bit0 of the row address counter, when the ConcurrentPortSignals/read_enable waveform is On.

- ShadowWrite : on | off;

The ShadowWrite property enables and disables the shadow write operation. The shadow write operation is used to detect inter-port bitline coupling faults for multiple ReadWrite logical ports. The memory BIST controller performs a shadow write on inactive ReadWrite ports in specific phases of the SMarchCHKBci, SMarchCHKBcil, and SMarchCHKBvcd algorithms during the READ cycle of the active ReadWrite port. For memories with one Read and two Write ports, the SMarchCHKBci, SMarchCHKBcil, and SMarchCHKBvcd algorithms also support shadow write on inactive write ports during the READ and WRITE cycles of the active port.

A value of on enables the shadow write by forcing the row address to its full binary range. The default value of off disables the shadow write.

When ShadowWrite is on and ShadowWriteOk is on in the memory library file, concurrent operations are enabled in programmable controllers. The OperationSet SyncWRvcd is automatically selected if Algorithm: SMarchCHKBvcd is specified, and the user can then modify the way concurrent operations are performed in the SMarchCHKBvcd library algorithm, or in any custom algorithm. ShadowWrite operations are not controllable from

the operation set and are only used by library algorithms. ConcurrentWrite operations are fully controllable from the operation set.

If ShadowWrite is On, the following conditions must be satisfied:

- o The memories must be synchronous SRAM. Preferably, Siemens EDA Sync or SyncWR operation sets should be used. Other operation sets must take into account that the waveform used to perform shadow writes from inactive ports is the exact inverse of the waveform described for write enable signals. Also, the row address changes at the same time as the write enable signal. These waveforms might not be appropriate in all cases, especially for asynchronous memories.

- o If no memory port with Function ShadowAddressEnable is defined, the number of rows cannot be a power of two. The row address range must not use the full binary count. It might be required to generate a new memory block with at least one more row to comply with this restriction. Note that this restriction does not apply to memories that use the SMarchCHKBvcd algorithm.

- o The memories must have a BitGrouping of 1.

- o The memories must have ONE of the following combinations of ports:

  - Two ReadWrite ports

  - One Read-only port and one Write-only port

  - Any number of Read-only ports and two Write-only ports

- o The algorithm used must be SMarchCHKBci, SMarchCHKBcil, or SMarchCHKBvcd.

- o If ShadowWrite is On, you must specify a row address. For information on specifying a row address, refer to the AddressCounter wrapper.

- o The ShadowWriteOK property must be On.

- ShadowWriteOK : on | off;

  The ShadowWriteOK property is used to indicate that the memory can tolerate an out-of-range address during shadow write without damaging the memory logic or corrupting the data. If enabled it suppresses the count range rule checking. This property must be set to on to perform shadow writes on the memory. ShadowWrite operations are not controllable from the operation set and are only used by library algorithms. ConcurrentWrite operations are fully controllable from the operation set.

  A value of off indicates that the memory cannot tolerate out of range addresses and therefore shadow writes should not be used on this memory.

- TransparentMode : syncmux | none | asyncmux ;

  The TransparentMode property specifies when and how memories are bypassed during scan testing. Bypassing memories enables testing the user interface logic to and from these embedded memories as well as the memory BIST controller and interface circuitry.

The default value is syncmux. Valid values are as follows:

- o syncmux — Inserts an additional multiplexer and flip-flop on the memory output ports that enable combinational ATPG tools to test the interface between the user logic and the memory. When you specify this value, the interface also provides control values for scan chain testing. In addition, you can observe the memory input port by using the same flip-flop that provides the control values for memory data output ports.

    For ATPG tools that have the capability to generate test patterns through the specified memories, the additional multiplexer might not be necessary.

- o none — Specifies that no bypass logic is added in the memory interface. Specify none if you have a bidirectional data bus or the memory implements an internal bypass logic.

- o asyncmux — Inserts an additional multiplexer on the memory output ports so that data is directly transferred from the data input ports to the data output ports. This setting enables combinational ATPG tools to test the interface between the user logic and the memory. You can use this value only when DataOutStage is set to none.

_____ **Note** _____

To perform memory BIST, which requires controllability of the memory output, you must disable any internal bypass circuitry in your memory.

Examples

Example 1

The following example defines the read operation in the operation set for a typical synchronous memory. The read access is activated in the first cycle, and the output data is compared on the second cycle.

```
Operation (Read) {
  Cycle {
    read_enable: on;
  }
  Cycle {
    read_enable: off;
    strobe_data_out: on;
  }
}
```

Figure A-1illustrates the same memory having two stages of built-in pipelining on the output data. To account for the latency of the output data introduced by the internal pipelining stages, you can specify the PipelineDepth property and apply the same read operation. The strobe_data_out signal is pipelined by two cycles without any modifications to the operation set.

Figure A-1. Memory With 2 Stages of Built-In Pipelining on the Output Data
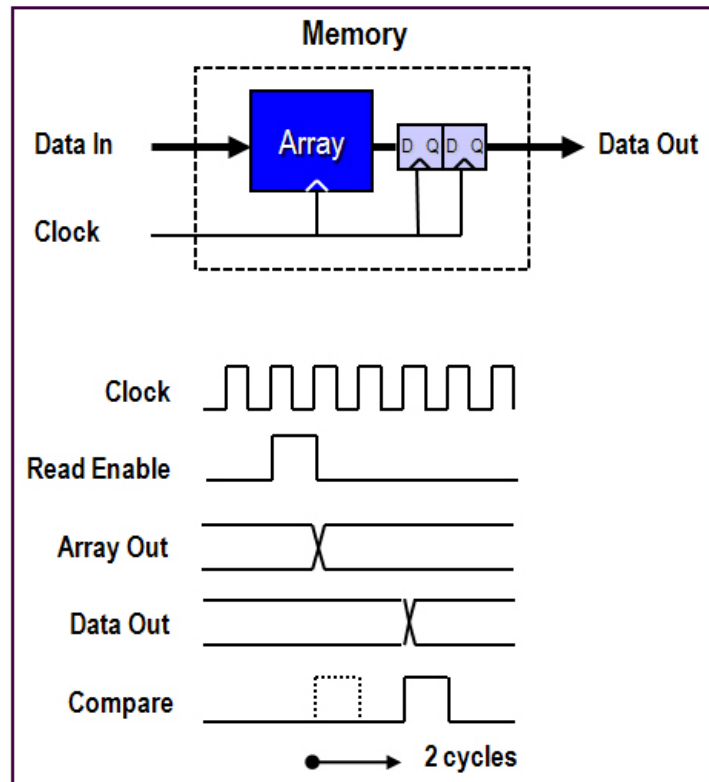
Example 2

The following outlines an example BitGrouping usage. Consider the physical layout of a 8-bit memory in which data is stored in columns. That is, data[0] is placed under column 0, data[1] is placed in column 1, data[2] is in column 2,.., data [7] is in column 7. Sometimes this type of memory is referred to as a *word-oriented memory*. The memory BIST controller applies a checkerboard pattern to the memory by alternating values of 0 and 1 in the address locations, as shown. For this memory, the BitGrouping is equal to the data width, which is 8.

Physical Layout   Checkerboard Pattern
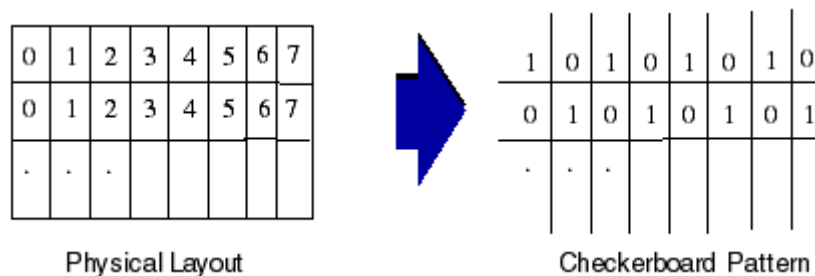
The numbers 0-7 in the figure above represent the physical data bits 0-7. Note that a large majority of memories with column address bits use a layout corresponding to this example, with a BitGrouping of 1.

Example 3

The following outlines another usage for BitGrouping. Many memories are designed to gather bits from different addresses with the same data reference into the same array. That is, the memory has separate arrays for each bit of the word. Here is an example of a memory with 4 data bits. There are four arrays with each array has the specific data bit reference. The BitGrouping property for this type of memory should be defined with a value of 1.
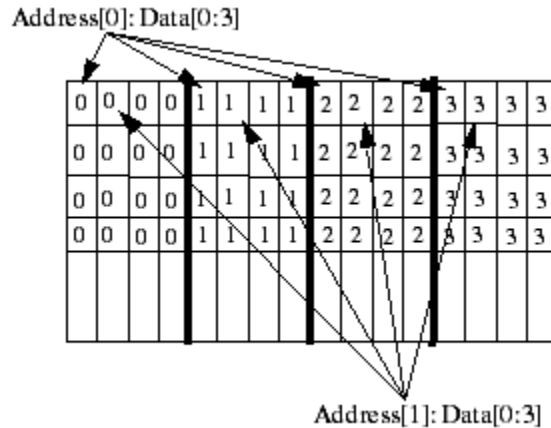
Example 4

In this example BitGrouping usage, consider a memory where only three of the consecutive data bits are physically gathered in the array. The physical layout of a 6-bit memory that is broken into two arrays. This example shows a bit grouping of 3.

Physical Layout
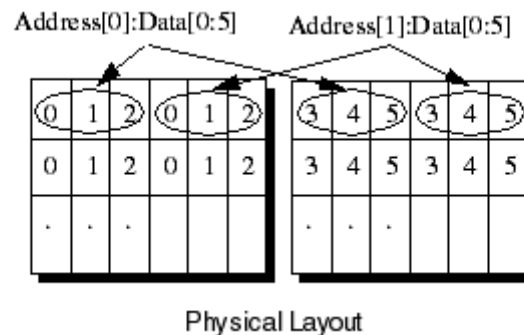
To access bit0 of the first row, specify the column address y0 to select the correct bit0. The checkerboard pattern for this grouping needs a 1 in the first bit0 and a 0 in the second bit0 of the same row. Specifying BitGrouping 3 for this example enables the hardware to make this correction.

Related Topics

set_design_sources [Tessent Shell Reference Manual]

read_core_descriptions [Tessent Shell Reference Manual]

set_module_matching_options [Tessent Shell Reference Manual]

report_config_data [Tessent Shell Reference Manual]

report_config_syntax [Tessent Shell Reference Manual]

Port

The Port wrapper is used to define signal port properties for a memory.

Syntax

```
Core(<core_name>) {
 Memory {
    Port(port_name) {
       BusRange           : range;
       BistOrTmActive     : on | off;
       BitsPerWriteEnable : int;
       Direction          : Input | Output | InOut;
       DisableDuringScan  : on | off | auto;
       Function           : None | Address | BistEn | BistOn |
                            Clock | Data | GroupWriteEnable | LogicHigh |
                            LogicLow | Open | OutputEnable |
                            ReadEnable | ScanTest | Select |
                            ShadoWaddressEnable |
                            WriteEnable | CAS | RAS | User0 | User1 |
                            User2 | User3 | User4 | User5 | User6 | User7 |
                            User8 | User9 | User10 | User11 | User12 |
                            User13 | User14 | User15 | User16 | User17 |
                            User18 | User19 | User20 | User21 | User22 |
                            User23 | Refresh | Activate | Precharge |
                            ValidData | BisrParallelData | BisrSerialData |
                            BisrClock | BisrReset | BisrScanenable
       LogicalPort        : port_id_string, ... ;
       Polarity           : activeLow | activeHigh;
       Retimed            : on | off;
       SafeValue          : binary; //Default: x
       EmbeddedTestLogic {
         TestInput        : port_name;
         TestOutput       : port_name;
       }
    }
  }
}
```

Description

The Port wrapper defines the direction, function, and bus parameters for a signal port. The memory TCD includes a Port wrapper for each port on the memory.

When you have multiple Port wrappers specified within the memory TCD of Function: Address or Function:Data, Tessent MemoryBIST concatenates and drives the ports of each Function type in the order they appear within the memory TCD. If the resulting order is not what you intended, you must re-arrange the appropriate Port wrappers in the memory TCD. An example demonstrating the concatenation is provided in Example 1.

Parameters

- port_name(*range*)

  The port_name property is the name of the port. Because Tessent MemoryBIST instantiates the memory into a memory BIST interface, port_name must match the actual port name of the memory module. The optional range element identifies that range for bused ports.

  The following example defines a Port wrapper of an 8-bit wide port named D1:

  ```
  Port (D1[7:0])
  ```

  If you include the range for a bused port in the Port wrapper, you do not need to specify the BusRange property.

  > **Note**
  > The use of an escaped identifier for port_name is not supported.

  If port_name contains a %d character identifier, Tessent MemoryBIST expands the port based on the bit range specified after %d or on the BusRange property. This enables the support of both bused and scalar memory ports.

  > **Note**
  > The %d scalar notation is case sensitive. If you use %D, the Port is treated as a bus with %D as part of the name. This makes it impossible to load the Tessent Shell output into other Siemens EDA tools or any simulator.

- BusRange : *range*;

  The BusRange property specifies the range for a bused port. You do not need to specify the range for single bit ports. The following syntax specifies this property:

  range is defined as [LeftIndex:RightIndex], where [LeftIndex:RightIndex] identifies the range for bused ports.

  This example specifies an 8-bit bused port named D1

  ```
  Port(D1) {
    BusRange: [7:0];
  }
  ```

- BistOrTmActive : on | off;

  The BistOrTMActive property specifies if the memory control port should be driven during both BIST and scan test.

  A value of on instructs Tessent MemoryBIST to make the following ports active during both BIST and scan test: Activate, Precharge, Refresh, Select, OutputEnable, RAS, CAS, WriteEnable or GroupWriteEnable.

  A value of off instructs the tool to handle the port as necessary for BIST.

- BitsPerWriteEnable : int; // default: 1

A parameter that indicates the number of data bits associated per group write enable signal. The default value is 1. Use in conjunction with the port function GroupWriteEnable.

- Direction : <u>Input</u> | Output | InOut;

The Direction property specifies the direction of the memory port. A value of Input identifies a port as an input to the memory. A value of Output identifies a port as an output from the memory. A value of InOut identifies a bidirectional data port. In this case, you must associate the bidirectional data port with an output enable port.

> **_____ Note _____**
> If Function is set to either Data or None, you must specify Direction.

- DisableDuringScan : on | off | <u>auto</u>;

The DisableDuringScan property specifies if certain memory ports are to be disabled during scan and logic test. The controller type determines the valid port types that can be disabled. The functional input is gated with the scan and logic test mode signal LV_TM within the memory interface. The gating differs based on the implementation of the memory_bypass_en DFT signal (refer to the add_dft_signals command for more information). If the signal is not implemented, the input is gated to its inactive value for the entire duration of the test. If the signal is implemented, the input is gated for the entire duration of the test when the memory is bypassed (memory_bypass_en = 1). When the memory is not bypassed (memory_bypass_en = 0), the input is gated only during shifting of the scan chains so the patterns can be applied during the capture phase.

Setting DisableDuringScan to on prevents the memory from being activated by functional logic during scan and logic test. Therefore, the power requirement is minimized. The value of off instructs Tessent MemoryBIST to not add the gating capability.

DisableDuringScan can only be set to on for the following functions: Activate, Precharge, Refresh, Select, RAS, CAS, WriteEnable, InterfaceReset, ReadEnable, GroupWriteEnable, and User0 through User23. For function OutputEnable, the value for this property is always automatically inferred. It is set to off if the port is tied to its active value, and set to on if driven by system logic. The property is ignored for any other function.

For the default value of auto, Tessent MemoryBIST determines automatically for the current port, if the value should be set to on or off. The value resolves to on for ports with these functions: Activate, CAS, Precharge, RAS, Refresh, Select, User0 through User23, and WriteEnable. For function OutputEnable, the value for this property is always automatically inferred. It is set to off if the port is tied to its active value, and on if it is driven by system logic. For ports with any other function, the value resolves to off.

- Function : function_type;

The Function property specifies the type of function of the signal port. The default value is none. Table A-3 describes the valid values for each Function value.

Table A-3. Valid Function Values

Function	Description
Activate	Specifies the activate signal for DRAM. A port of this function can be controlled by the activate property of the DramSignals wrapper in the operation set.
Address	Specifies a memory address port.
BisrClock	Specifies the port that controls the clock to the internal BISR chain registers. This port function is mandatory for memories with serial BISR interface.
BisrParallelData	Specifies the memory ports used for memory repair. This port function is mandatory for memories with parallel BISR interface.
BisrReset	Specifies the port that controls the internal BISR register chain asynchronous reset. This port function is mandatory for memories with serial BISR interface.
BisrScanEnable	Specifies the port that enables the shifting of the internal BISR chain. This port function is optional for memories with serial BISR interface.
BisrSerialData	Specifies the internal BISR chain serial input and output ports. The *BisrSerialData* port function is mandatory for memories with a serial BISR interface. You must specify the port direction using *Direction*: input | output | inout. In most cases, a memory with a serial BISR interface must have two *BisrSerialData* ports where one port has direction input and the other port has direction output. The exception is when a memory has serial repair access but no shift out port. In this case, you specify only the *BisrSerialData* input port.
BistEn	Specifies that the port is used to control a clock multiplexer in the memory. A port of this function is connected to the *BIST_EN* signal of the controller.
BistOn	Specifies that the port is used to control the signals (data/address/control but not a clock) selection in the memory. A port of this function is connected to the *BIST_ON* signal of the controller.
CAS	Specifies the column access strobe signal for DRAM. A port of this function can be controlled by the cas property of the DramSignals wrapper in the operation set.
Clock	Specifies a memory clock port.

Table A-3. Valid Function Values (cont.)

Function	Description
Data	Specifies the data port. When *Data* is specified for the Function property, Direction indicates if *Data* is an input, output, or bidirectional port
GroupWriteEnable	Specifies a write enable port that controls one or more bits in the data path. Group write enable ports must be associated with the data input port being controlled using the LogicalPort property in their Port wrappers.

The number of data bits in the group is defined using the BitsPerWriteEnable property.

The group write enable ports can be controlled using the even_group_write_enable and odd_group_write_enable properties of the Cycle wrapper. |
LogicHigh	Specifies that the associated port is to be tied to a logic high value.
LogicLow	Specifies that the associated port is to be tied to a logic low value.
None	Specifies a port that does not need to be controlled during memory BIST. This is the default. The functional connection to this port is preserved. Use SafeValue to control the memory port value during the memlibc certification flow. For pattern generation, a ProcedureStep in the PatternsSpecification may be required to set the proper value after you insert memory BIST into the design. Tessent MemoryBIST does not intercept the memory port.
Open	Specifies an unused output port of the memory that is to be left unconnected. The associated port must be defined with Direction output.
OutputEnable	Specifies the memory tri-state output enable that drives the data to the memory interface. A port of this function can be controlled by the output_enable property of the Cycle wrapper in the operation set.
Precharge	Specifies a port that controls the precharge circuitry in DRAM. A port of this function can be controlled by the precharge property of the DramSignals wrapper in the operation set.
RAS	Specifies the row access strobe signal for DRAM. A port of this function can be controlled by the ras property of the DramSignals wrapper in the operation set.

Table A-3. Valid Function Values (cont.)

Function	Description
ReadEnable	Specifies a memory read enable signal. A port of this function can be controlled by the read_enable property of the Cycle wrapper in the operation set.
Refresh	Specifies a port that controls the refresh circuitry in DRAM. A port of this function can be controlled by the refresh property of the DramSignals wrapper in the operation set.
ScanTest	Specifies the port that configures the embedded test logic to enable scan testing. Typically, the port disables the memory's tri-state outputs or enables the memory bypass mode. A port of this function is connected to the memory_bypass_en DFT signal, gated with the logic test ltest_en DFT signal.
Select	Specifies a memory (chip) select signal. A port of this function can be controlled by the select property of the Cycle wrapper in the operation set.
ShadowAddressEnable	Specifies that the port is used to enable the ShadowWriteAddress on the memory. This property is valid only for memories that support the ShadowWrite operation. A memory with a port of this function can have ShadowWrite: *on*, and the number of rows in the memory equal to a power of two. Asserting this port on the memory enables you to perform a write operation without corrupting the memory array content.
User0 - User23	Enable user-defined waveform to be applied to the associated memory port. These port functions can be assigned to any input port and can be controlled by the corresponding properties, user0 through user23, in the operation set.
ValidData	Specifies an output signal from the user logic that is asserted when valid data is available on the read data bus.
WriteEnable	Specifies a memory write enable signal. A port of this function can be controlled by the write_enable property of the Cycle wrapper in the operation set.

The following usage conditions apply:

o   Selection guidelines for Function values are as follows:

- If a port needs to have a different logic value in functional and Memory BIST mode, then a port function other than LogicHigh or LogicLow must be used.

When LogicHigh or LogicLow is specified, no mux is inserted so the port logic levels are the same in both modes.

- Most values of port Function cause a multiplexer to be inserted to select between the functional and test inputs. The following value selections assume a dedicated test port and no multiplexer is inserted: BistEn, BistOn, ScanTest, ShadowAddressEnable, BisrParallelData, BisrSerialData, BisrClock, BisrReset and BisrScanEnable.

- When you specify a port Function value of None, MemoryBIST does not insert a mux and you must specify a SafeValue, which the tool uses to control the memory port value during the memlibc certification flow. For pattern generation, a ProcedureStep in the PatternsSpecification may be required to set the proper value after you insert memory BIST into the design.

- Table A-4 provides additional Function value selection details and guidelines for various port operations in functional and memory BIST modes.

Table A-4. Function Value Details and Guidelines

Port Operation	Value Selection
Tied high or low in functional and memory BIST mode	LogicHigh or LogicLow
Port is driven in functional mode but needs to be static in memory BIST mode	One of User0 to User23 values, with the appropriate logic level specified with the corresponding property in the Port wrapper. For example, ActiveHigh (default) if the port should be driven to logic 0 during memory BIST, and ActiveLow for a logic 1 level.
Port needs to toggle in memory BIST mode	Any port Function controlled from the OperationSet wrapper, including User0-23.
Port is driven in functional mode, but cannot be gated.	None; set SafeValue to the appropriate level applied by the user circuit. For pattern generation, a ProcedureStep in the PatternsSpecification may be required to set the proper value when memory BIST is inserted into the design.

- LogicalPort : *port_id_string*, ...;

  A property with a repeatable string value. The LogicalPort property groups address, data, and control signals for memory BIST. Memory BIST requires a read port and a write port to perform the test. The read port and the write port can be either separate ports or shared ports.

  When LogicalPort is omitted from a Port wrapper, Tessent MemoryBIST treats the port as a global port and includes the port in all logical ports.

An example implementation is shown below:

```
Port (D1[7:0]) {
  Function: Data;
  LogicalPort: A;
}
Port (D2[7:0]) {
  Function: Data;
  LogicalPort: B;
}
Port (Clock) {
  Function: Clock; //This port belongs to both logical port A and B.
}
```

_____ **Note** _____
You can specify only one clock, one set of addresses, and one set of data terminals per logical port.

- Polarity : activeLow | activeHigh;

  The Polarity property specifies the active polarity of the port. Every operation begins with all signals at their inactive values.

  A value of activehigh specifies that a logic "1" on the port activates the function. This is the default. A value of activelow specifies that a logic "0" on the port activates the function.

  An example Polarity entry for a WriteEnable signal that activates when a logic "0" occurs on the port would be:

```
Function: WriteEnable;
Polarity: ActiveLow;
```

- Retimed : on | off;

  For a memory implementing the serial repair interface, the Retimed property specifies if the internal BISR chain has a negedge retiming flop on its scanout port. This property is used only when declaring the memory pin corresponding to the repair serial data output.

  A value of on specifies that a negedge retiming flop is present at the output of the internal BISR chain. A value of off specifies that no retiming flop is present on the internal BISR chain. This is the default.

  The following example specifies that the internal BISR chain has a negedge retiming flop on the SDOUT port:

```
Port(SDOUT) {
  Function: BisrSerialData;
  Direction: Output;
  Retimed: On;
}
```

- SafeValue : _binary_; // Default: x

  The SafeValue property specifies an assumed value that the tool applies to the memory input ports with Function:None, only during the memlibCertify (memlibc) memory library

certification flow. The memlibc certification flow instantiates the controller, interfaces, and memory modules in a testbench which you use to verify the operation of the BIST circuitry.

Tessent MemoryBIST does not add additional logic to a memory port for which you specify a SafeValue value. Once you insert the memory controller and interfaces into your design, the functional logic drives the memory inputs defined with Function:None.

> _____ **Note** _____
> 📄 The SafeValue property only has an effect during the memlibc certification flow.
> You need to ensure the functional logic properly drives the port after you insert the memory controller and interface into your design.

For single-bit ports, you specify the value as a binary value, or as a "1", "0", or "X" value. Setting this property to 1'bx or "X" sets the port to logic "X" in the memlibc certification flow testbench. "X" denotes the value as ambiguous and is the default value. The value of 1'b0 or "0" sets the port to logic "0" and the value of 1'b1 or "1" sets the port to logic "1" in the memlibc certification flow testbench.

For bussed ports, you must specify the value as a binary value, matching the bit width of the port, otherwise the tool reports an error. The exception is when specifying "X" for all bits of a bussed port. In this case, setting this property to "X" assigns "X" to all bits of the bussed port. You need to specify SafeValue only for ports that specify Direction: Input and Function:None in the definition.

- EmbeddedTestLogic

The EmbeddedTestLogic wrapper describes any embedded test interface feature that exist for the functional port being defined.

The following usage conditions apply:

  o EmbeddedTestLogic wrappers cannot be defined for ports with the following functions: BistOn, BistEn, ScanTest, LogicLow, LogicHigh, Open, None.

  o EmbeddedTestLogic wrappers cannot be defined for ports when the Direction property is set to the value inout.

- EmbeddedTestLogic/TestInput : _port_name_(range);

The TestInput property identifies the input test port associated with a functional port. When defined for a functional input port, this property specifies that embedded multiplexing logic exists within the memory to select between the functional port and the test port. This selection is controlled by an input port with Function BistOn.

For a functional output port, this property specifies that embedded multiplexing logic exists within the memory to bypass the memory output with test data. This selection is controlled by an input port with Function ScanTest.

For an input port of function Clock, this property specifies that embedded multiplexing logic exists within the memory to select between the functional clock and a BIST clock. This selection is controlled by an input port with Function BistEn.

The port_name value specifies the name of the input test port associated with the functional port being defined. The optional range value identifies the range for bused ports.

If port_name contains a %d character identifier, Tessent MemoryBIST expands the port based on the bit range specified after %d or on the BusRange property. This enables the support of both bused and scalar memory ports.

> _____ **Note** _____
> The %d scalar notation is case sensitive. If you use %D, the Port is treated as a bus with %D as part of the name. This makes it impossible to load the tool's output into other Siemens EDA tools or any simulator.

The following usage conditions apply:

- o The bus range must be the same as the range specified for the functional port being defined. The exceptions are the test data input port and test group write enable input port.

- o You can specify a test data input port or a test group write enable input port that is narrower than its functional signal. The test input signals are assumed to be repeated inside the memory module to form the internal data bus or the write mask controls to the memory array. The following limitations apply when testing such a memory design:

  - The valid test data width must range from 2 bits up to the functional port width.

  - The valid test group write enable width must range from 1 bit up to the functional port width.

  - Only data input and group write enable ports specified with the EmbeddedTestLogic/TestInput property are supported; all other test input ports must conform to the identical width requirement.

  - No data scrambling can be present within the memory array itself.

  - To support scalar or bit-blasted types for the data and group write enable ports, their definition in the memory library is restricted to the following syntax; the functional and test ports must be specified in one Port wrapper using the %d notation:

    ```
    Port (DIN%d[31:0]) { // functional port
      Function: Data;
      Direction: Input;
      EmbeddedTestLogic {
        TestInput: DFTDIN%d[31:0]; // test port
      }
    }
    ```

- o If this property is defined for an input functional port, then a port with Function BistOn must also be defined. If defined for an input clock port, then a port with

Function BistEn must also be defined. If defined for an output functional port, then a port with the Function ScanTest must also be defined.

o The existence of this property must be consistent among all functional ports having the same function and assigned to the same logical port.

o If this property is defined for a functional output port, then the Core/Memory/ TransparentMode property must have a value of SyncMux.

The following example specifies that the test input port TA[7:0] is associated with functional port A[7:0].

```
Port (A[7:0]) {
  Function: Address;
  EmbeddedTestLogic {
    TestInput: TA[7:0];
  }
}
Port (TESTSEL) {
  Function: BistOn;
}
```

- EmbeddedTestLogic/TestOuput : *port_name*(range);

The TestOutput property identifies the output test port associated with a functional port. When defined for a functional output port, the TestOutput property identifies the dedicated output port used to observe test responses. When defined for a functional input port, the property identifies the output port used to observe the multiplexed functional and test input data.

The port_name value specifies the name of the output test port associated with the functional port being defined. The optional range value identifies the range for bused ports.

If port_name contains a %d character identifier, Tessent MemoryBIST expands the port based on the BusRange property. This enables the support of both bused and scalar memory ports.

_____**Note**_____

The %d scalar notation is case sensitive. If you use %D, the Port is treated as a bus with %D as part of the name. This makes it impossible to load the tools' output into other Siemens EDA tools or any simulator.

The following usage conditions apply:

o If this property is used with an Input port, then the TestInput property must also appear.

o The bus range must be the same as the range specified for the functional port being defined.

o The existence of this property must be consistent among all functional ports having the same function and assigned to the same logical port.

This example specifies that the test output port TQ[15:0] is associated with functional data output port Q[15:0].

```
Port (Q[15:0]) {
  Function: Data;
  Direction: Output;
  EmbeddedTestLogic {
    TestOutput: TQ[15:0];
  }
}
Port (TESTSEL) {
  Function: BistOn;
}
```

Examples

Example 1

This example shows how to properly create a memory TCD for a memory that has an address composed of several ports. For this example, the memory address is composed of the following ports:

- BNK[1:0] — Bank Address port

- ROW[5:0] — Row Address port

- COL[1:0] — Column Address port

The memory TCD syntax accepts more than one Port wrapper with Function:Address. The order in which the Port wrappers appear in the memory TCD have an impact on how the MemoryBIST logic drives the memory address ports.

Consider the Port wrapper order in the memory TCD of Example A-1, which results in the MemoryBIST logic driving the memory address ports in an unintended fashion.

Note there is no Port Function to specify whether a port is a BankAddress, RowAddress, or ColumnAddress port. Therefore, each memory address port has their Function property defined as "Address".

Example A-1. Incorrect Memory TCD Port Wrapper Ordering

```
Port (ROW[5:0]){
  function: Address;
}
Port (COL[1:0]){
  function: Address;
}
Port (BNK[1:0]){
  function: Address;
}
```

```
AddressCounter {
  Function(BankAddress){
    LogicalAddressMap {
      BankAddress[1:0]:Address[9:8];
    }
    CountRange[0:3];
  }
  Function(ColumnAddress){
    LogicalAddressMap {
      ColumnAddress[1:0]:Address[1:0];
    }
    CountRange[0:3];
  }
  Function(RowAddress){
    LogicalAddressMap {
      RowAddress[5:0]:Address[7:2];
    }
    CountRange[0:55];
  }
}
```

The AddressCounter wrapper specifies how MemoryBIST constructs the address counter, as highlighted in red text. For this example, the address counter is 10 bits wide and constructed with the following segmentation:

```
address counter[9:0]={BankAddress[1:0],RowAddress[5:0],ColumnAddress[1:0]}
```

Note

The Port wrapper listing order in the memory TCD affects how the address counter drives the memory address ports.

The ROW[5:0] Port wrapper is the first address port specified in the memory TCD, for which MemoryBIST will use the first six bits of the address counter to drive the memory ROW port. The second Port wrapper is the COL[1:0] port, for which MemoryBIST uses the next two address counter bits to drive the COL port. The last Port wrapper is the BNK[1:0] port which the two remaining address counter bits drive. That is:

```
address counter[9:4]  drives the memory ROW[5:0]  port
address counter[3:2]  drives the memory COL[1:0]  port
address counter[1:0]  drives the memory BNK[1:0]  port
```

Notice this results in a misalignment between the address counter segmentation and the memory address ports. The address counter[9:4] bits drive the memory's ROW[5:0] port, however address counter[9:4] is composed of {BankAddress[1:0],RowAddress[5:2]}.

Simulations may not be able to detect this misalignment. If the number of words in your memory is equal to the full binary address count, then the Verilog simulations will likely pass. This address misalignment would affect the physical checkerboard pattern created in the memory. Since memory simulation models do not consider the physical arrangement of the memory array, no simulation issues would result.

However, if the number of words in your memory is not equal to the full binary address count, the Verilog simulations could fail. The misalignment could cause one or more operations to be outside of the memory's valid address range. In the Port and AddressCounter wrappers of Example A-1, the RowAddress has a limited count range as shown in the green highlighted text. The maximum value the memory ROW[5:0] port should drive is 6'b110111, or decimal 55. The misalignment because of the Port ordering will result with the ROW[5:0] port actually driving a maximum of 6'b111101, or decimal 61, which is outside of the valid row count range.

To correct the misalignment, the Port wrappers of Function:Address should appear in the same order as the AddressCounter segmentation, which is BankAddress[1:0], RowAddress[5:0], and finally ColumnAddress[1:0] as shown in Example A-2.

Example A-2. Correct Memory TCD Port Wrapper Ordering

```
port (BNK[1:0]){
  function: Address;
}
port (ROW[5:0]){
  function: Address;
}
port (COL[1:0]){
  function: Address;
}

AddressCounter {
  Function (BankAddress){
    LogicalAddressMap {
      BankAddress[1:0]:Address[9:8];
    }
    CountRange[0:3];
  }
  Function  (RowAddress){
    LogicalAddressMap {
      RowAddress[5:0]:Address[7:2];
    }
    CountRange[0:63];
  }
  Function (ColumnAddress){
    LogicalAddressMap {
      ColumnAddress[1:0]:Address[1:0];
    }
    CountRange[0:3];
  }
}
```

Related Topics

report_config_data [Tessent Shell Reference Manual]

report_config_syntax [Tessent Shell Reference Manual]

Memory

Core

AddressCounter

The AddressCounter wrapper can segment the address bits into bank, row or column address and enables you to specify the count range of each address segment.

Syntax

```
Core(core_name) {
  Memory {
    AddressCounter {
      Function(address|columnAddress|rowAddress|bankAddress) {
        LogicalAddressMap {
          RowAddress[x:y]    : Address[a:b]; // repeatable
          ColumnAddress[x:y] : Address[a:b]; // repeatable
          BankAddress[x:y]   : Address[a:b]; // repeatable
        }
        CountRange          [lowRange:highRange];
      }
    }
  }
}
```

Description

This wrapper is mandatory.

Typically, the lower address bits correspond to the column or row segment and the higher address bits correspond to the bank segment. Tessent MemoryBIST's controller supports testing of memories with asymmetric banks containing a different number of rows per bank. For more information about asymmetric banks, refer to the Core/Memory/NumberOfWords and Core/Memory/CountRange properties.

> **Note**
>
> Shadow read operations, memory templates with the PhysicalDataMap wrapper or the PhysicalAddressMap wrappers, and all checkerboard algorithms require segmented addresses. All checkerboard algorithms require a segmented address counter to build the correct physical one-by-one checkerboard pattern.

To ensure proper memory addressing, refer to Port wrapper Example 1 for information on the ordering of the memory TCD Ports of Function:Address and the AddressCounter/ LogicalAddressMap address segmentation elements.

Parameters

- Function(address | columnAddress | rowAddress | bankAddress)

  The required Function wrapper defines how the memory BIST controller drives memory address ports. Use the Function wrapper to segment the address port into column, row, and bank addresses as well as to specify the count range of each segment.

The following example shows an address port segmented into 4 row address bits and 7 column address bits. The row address range is defined as 0 to 15. The column address range is not specified and defaults to the full binary count of the column address size (0 to 127).

```
Function (address) {
  LogicalAddressMap {
    ColumnAddress[6:0]: Address[6:0];
    RowAddress[3:0]: Address[10:7];
  }
}
Function (rowAddress) {
  CountRange[0:15];
}
```

- Function/LogicalAddressMap

  The LogicalAddressMap wrapper maps the column, row, and bank address function bits to the logical address bits.

  The following usage conditions apply:

  o The values for ColumnAddress[x], RowAddress[x], and BankAddress[x] together must use the entire range specified for the address port (Core/Memory/Port/Function Address).

  o Each bit in Address[x] must map to either a column, row, or bank address counter bit or be multiplexed to row and column address bits. Multiplexing of bank address bits is not supported.

  o If a multiplexed row address segment and column address segment feature different sizes, you can define padding bit values for the shorter address segment with the multiplexed_address_padding of the Operation wrapper in the operation set.

  o If at least one address bit position is multiplexed, either all row address bits or all column address bits must be multiplexed as well.

  o Address multiplexing is supported only for single-port (1RW) memories.

  Mapping assignments, whether specified as either msb:lsb or lsb:msb, are equivalent and implemented in RTL the same way.

  For example, the following mappings:

```
ColumnAddress[0:2]  : Address[2:0]
ColumnAddress[2:0]  : Address[2:0]
```

  results in the same mapping in hardware:

```
ColumnAddress[0] — Address[0]
ColumnAddress[1] — Address[1]
ColumnAddress[2] — Address[2]
```

The following example shows a mapping for a 15-bit address bus with a non-contiguous 8-bit row address segment, which is composed of address bit 14, and from address bits 6 to 0:

```
Function (Address) {
  LogicalAddressMap {
    RowAddress[6:0]  : Address[6:0];
    ColumnAddress[6:0]  : Address[13:7];
    RowAddress[7:7]  : Address[14:14];
  }
}
```

- Function/LogicalAddressMap/BankAddress[*x:y*] : Address[*a:b*]

  A repeatable property and address pair. It maps parts or all of the bank address function counter bits to the logical address bits.

  For an example, see the Function/LogicalAddressMap wrapper above.

- Function/LogicalAddressMap/ColumnAddress[*x:y*] : Address[*a:b*]

  A repeatable property and address pair. It maps parts or all of the column address function counter bits to the logical address bits.

  For an example, see the Function/LogicalAddressMap wrapper above.

- Function/LogicalAddressMap/RowAddress[*x:y*] : Address[*a:b*]

  A repeatable property and address pair. It maps parts or all of the row address function counter bits to the logical address bits.

  For an example, see the Function/LogicalAddressMap wrapper above.

- Function/CountRange [*lowRange:highRange*]

  The CountRange property specifies the count range for the column, row, or bank address segments.

  The following example shows an address port segmented into 4 row address bits and 7 column address bits. The row address range is defined as 0 to 15. The column address range is not specified and defaults to the full binary count of the column address size (0 to 127).

```
Function (address) {
  LogicalAddressMap {
    ColumnAddress[6:0]: Address[6:0];
    RowAddress[3:0]: Address[10:7];
  }
}
Function (rowAddress) {
  CountRange[0:15];
}
```

Note _____

Asymmetric memory banks may exist with different numbers of rows per bank. For more information about asymmetric banks, refer to the NumberOfWords and CountRange properties.

Related Topics

Memory

Port

Core/Memory/NumberOfWords

Core/Memory/CountRange

PhysicalAddressMap

The PhysicalAddressMap wrapper describes the mapping between the logical address and the physical address in the memory array.

Syntax

```
Core(core_name) {
  Memory {
    PhysicalAddressMap {
      BankAddress[bit]   : expression; // repeatable
      ColumnAddress[bit] : expression; // repeatable
      RowAddress[bit]    : expression; // repeatable
    }
  }
}
```

Description

The PhysicalAddressMap wrapper enables Tessent MemoryBIST to construct a physical checkerboard pattern in the memory. The left column of the PhysicalAddressMap wrapper identifies the memory pins. The right column of the wrapper shows how the address counter bits drive the memory pins

If your memory TCD file(s) do not include a PhysicalAddressMap wrapper, Tessent MemoryBIST assumes a one-to-one mapping. For example, RowAddress[x]: r[x], ColumnAddress[x]: c[x], and BankAddress[x]: b[x].

In addition, if the number of equations is fewer than the number of address ports or if the mapping contains gaps, Tessent MemoryBIST assumes a one-to-one mapping for the missing addresses.

For example:

```
PhysicalAddressMap {
  ColumnAddress[0]: c[0] xor c[1];
  ColumnAddress[1]: c[1];
  ColumnAddress[3]: c[3];
}
```

Tessent MemoryBIST automatically fills in the missing address as follows:

```
ColumnAddress[2]: c[2];
```

When specifying the PhysicalAddressMap wrapper, ColumnAddress[x], RowAddress[x], and BankAddress[x] must be uniquely specified. In the example shown below, Tessent Memory BIST issues an error indicating ColumnAddress[1] is repeated:

```
PhysicalAddressMap {
  ColumnAddress[0]: c[0] xor c[1];
  ColumnAddress[1]: c[1];
  ColumnAddress[1]: c[2];
}
```

The same counter bit c[x], r[x], or b[x] may be specified in one or more equations. Omitting a counter bit from the PhysicalAddressMap wrapper is permitted. In the example shown below, Tessent MemoryBIST issues a warning indicating counter bit r[2] is not used in any equation:

```
PhysicalAddressMap {
  ColumnAddress[0]: c[0] xor c[1];
  ColumnAddress[1]: c[1];
  ColumnAddress[2]: c[2];
  RowAddress[0]: r[0] xor r[1];
  RowAddress[1]: r[1];
  RowAddress[2]: r[1];
}
```

For further information and detailed examples, refer to the "Memory BIST Physical Mapping Examples" appendix.

Parameters

- BankAddress[*bit*] ColumnAddress[*bit*] RowAddress[*bit*]

  These repeatable properties identify the signals that are mapped to the memory address port as described by the LogicalAddressMap wrapper in the AddressCounter wrapper. The index (*bit*) must match the LogicalAddressMap wrapper index values.

- *expression*

  This property represents a Boolean expression composed of Boolean operators, address counter bits, and optional parentheses. Table A-5 shows the valid operators and their precedence from highest to lowest. Operators on the same level are evaluated from left to right.

Table A-5. Operator Precedence

Operator	Precedence
not	highest
and, nand	
xor, xnor	
or, nor	lowest

The operands represent the counter bits used to generate the column, row, or bank addresses. Valid address counter symbols are as follows, where the integer x is a valid bit index of the address segment as defined in the AddressCounter wrapper:

- c[x] — column address

- r[x] — row address

- b[x] — bank address

The address counter bits r[x], c[x], or b[x] can be specified in any of the ColumnAddress, RowAddress, or BankAddress equations.

Optional parentheses define the operator precedence. Inserting parentheses is strongly recommended to avoid ambiguity in interpreting precedence and to improve readability.

The following are examples of supported mapping equations:

- RowAddress[0]: c[0];

- RowAddress[0]: not c[0] xor c[1] xor c[2];

- RowAddress[0]: (c[0] and not c[1]) or c[2];

- RowAddress[0]: not (r[11] and ((r[11] nor r[12]) nand not r[10]));

Examples

Example 1

The following example shows a typical PhysicalAddressMap wrapper.

```
PhysicalAddressMap {
  ColumnAddress[0]: c[0] xor c[1];
  ColumnAddress[1]: c[1];
  ColumnAddress[2]: c[2];
  ColumnAddress[3]: c[3];
  RowAddress[0]:    r[0] xor r[2];
  RowAddress[1]:    r[1] xor r[2];
  RowAddress[2]:    r[2];
  RowAddress[3]:    r[3];
  RowAddress[4]:    r[4];
  RowAddress[5]:    r[5];
  RowAddress[6]:    r[6];
  RowAddress[7]:    r[7];
  RowAddress[8]:    r[8];
  RowAddress[9]:    r[9];
}
```

Example 2

The following example shows the scrambling definition for a bussed address port. The port width is 4 bits; bit 0 represents the column address, and bits 1 to 3 represent the row address.

```
Memory {
  Port (A[3:0]) {
    Function: Address;
    Direction: Input;
  }
  ...
  AddressCounter {
    Function (Address) {
      LogicalAddressMap {
        ColumnAddress[0:0]: Address[0:0];
        RowAddress[2:0]: Address[3:1];
      }
    }
  }
  PhysicalAddressMap {
    ColumnAddress[0]: r[0] xor (c[0] or r[1]);
    RowAddress[0]: r[0];
    RowAddress[1]: r[1];
    RowAddress[2]: not (r[2] and not c[0]);
  }
}
```

Related Topics

Memory

AddressCounter

PhysicalDataMap

The PhysicalDataMap wrapper describes the mapping between the data input ports of the memory and the internal data lines.

Syntax

```
Core(core_name) {
  Memory {
    PhysicalDataMap {
      Data[bit] : [not] d[bit] [xor expression] ... ;
    }
  }
}
```

Description

Because physical memory cells are arranged in a particular order to ease layout, you must provide this information so Tessent MemoryBIST can correctly generate the checkerboard patterns. Tessent MemoryBIST does not support the PhysicalDataMap wrapper for ROM memories, as specified by the "Memory/MemoryType : rom;" setting in the memory TCD file. However, if you do specify the PhysicalDataMap wrapper for a ROM memory, Tessent MemoryBIST will ignore the wrapper with no impact to the generated hardware or to the default ROM MISR signature calculation.

If your memory TCD file does not include a physical data map wrapper, Tessent MemoryBIST assumes a one-to-one mapping, Data[n]: d[n].

The address counter bit specified in the physical data map equation must be defined in the Core/Memory/AddressCounter wrapper of your memory TCD file.

The data bus must be an even multiple of the number of the physical data map equations. Tessent MemoryBIST repeats the entire physical data map wrapper to produce a map wide enough for the actual data bus.

For further information and detailed examples, refer to the "Memory BIST Physical Mapping Examples" appendix.

Parameters

- Data[*bit*]

  This property represents the data port of the memory.

- d[*bit*]

  This property identifies the physical data bit controlled by the memory BIST controller.

- *expression*

  The *expression* value represents a Boolean expression specified within the physical data map equation of the form:

  ```
  <counterBit> | ([not] <counterBit> and [not] <counterBit>)
  ```

  The parentheses in the preceding expression are literal characters.

  <counterBit> is of the form c[bit] | r[bit], where c[bit], r[bit] represent the counter bits used to generate the column addresses (c[bit]), and row addresses (r[bit]), respectively.

  The following table shows the valid physical data map equation operators and their precedence. Operators on the same level of precedence are evaluated from left to right.

  Table A-6. Operator Precedence

Operator	Precedence
not	highest
and	
xor	lowest

  When specifying the mapping between the data input ports of the memory and the internal data lines, Data[bit] must be contiguous:

  ```
  PhysicalDataMap {
    Data[0]: d[0];
    //ERROR: missing Data[1]
    Data[2]: d[2];
    Data[3]: d[3];
  }
  ```

  Further, d[bit] must use every Data[bit] exactly once:

  ```
  PhysicalDataMap {
    Data[0]: d[0];
    Data[1]: d[0];
    //ERROR: d[0] used more than once. d[1] missing.
  }
  ```

Examples

Example 1

The first example is a typical PhysicalDataMap wrapper for an 8-bit wide memory.

```
PhysicalDataMap {
  Data[0]: not d[1] xor c[0];
  Data[1]: d[0] xor c[0];
  Data[2]: d[2];
  Data[3]: not d[3] xor (c[1] and not r[2]) xor r[4];
  Data[4]: d[4] xor (not c[3] and r[5]) xor (c[3] and not r[5]);
  Data[5]: d[5] xor c[0];
  Data[6]: d[6];
  Data[7]: not d[7] xor c[1] xor c[8];
}
```

Example 2

The second example shows two equivalent versions of the PhysicalDataMap wrapper for a 4-bit wide memory. Tessent MemoryBIST interprets the first PhysicalDataMap wrapper as shown in the second PhysicalDataMap wrapper.

```
PhysicalDataMap {
  Data[0]: not d[0];
  Data[1]: d[1];
}
PhysicalDataMap {
  Data[0]: not d[0];
  Data[1]: d[1];
  Data[2]: not d[2];
  Data[3]: d[3];
}
```

Related Topics

Memory

AddressCounter

GroupWriteEnableMap

The GroupWriteEnableMap wrapper describes the mapping of a memory's group write enable bit to the one or more data bits that it controls.

Syntax

```
Core(core_name) {
  Memory {
    GroupWriteEnableMap {
      GroupWriteEnable[gwe_bit] : d[bit_or_range], ... ; // repeatable
    }
  }
}
```

Description

The GroupWriteEnableMap wrapper is an optional wrapper that describes the mapping of a memory's group write enable (GWE) bit to the one or more data bits that it controls. If this wrapper is not specified, a uniform distribution of the GWE bits to memory data bits is assumed and all memory bits must be controlled by a GWE bit. This wrapper is only allowed if the memory has at least one Port wrapper with the Function GroupWriteEnable present in the library.

If the ports with Function GroupWriteEnable also feature EmbeddedTestLogic inputs, the number of GroupWriteEnable properties must match the size of the test GWE port. Otherwise, the number of GroupWriteEnable properties must match the size of the functional GWE port.

Parameters

- GroupWriteEnable[*gwe_bit*] : d[*bit_or_range*], ... ;

  A repeatable property and repeatable data bit or data bit range pairing that specifies which memory data bits are controlled by the GWE bit specified by *gwe_bit*.

  The *gwe_bit* indices for GroupWriteEnable must count from 0 to the size of the test input or functional GWE port, minus 1.

  The data *bit_or_range* indices must be within the range of functional memory data input ports, regardless of whether the data port has EmbeddedTestLogic or not. The bit range of vector Data Input ports must always start with "0". The indices cannot be duplicated within the GroupWriteEnableMap wrapper, however not all indices need to be used. If all indices are not used, a warning is issued.

Examples

This example shows a mapping of a 4 bit GWE port that covers 9 of 10 bits of memory data input ports:

```
GroupWriteEnableMap {
  GroupWriteEnable[3] : d[9];
  GroupWriteEnable[2] : d[8:6];
  GroupWriteEnable[1] : d[4];
  GroupWriteEnable[0] : d[3:0];
}
```

Given this mapping, data input bits 0-3 and 6-8 are controlled by GWE bits 0 and 2 with the Cycle/even_group_write_enable property and data input bits 4 and 9 are controlled by GWE bits 1 and 3 with the Cycle/odd_group_write_enable property. Note that data input bit 5 is not controlled by any GWE bit.

RedundancyAnalysis

The RedundancyAnalysis wrapper indicates that the built-in repair analysis (BIRA) feature is required for the memory described by the memory library.

Syntax

```
Core(core_name) {
  Memory {
    RedundancyAnalysis {
      RowSegmentRange {
        SegmentAddress[bit]: AddressPort(name);//repeatable
      }
      RowSegment(string) {  //repeatable
      }
      ColumnSegmentRange {
        SegmentAddress[bit] : AddressPort(name); // repeatable
      }
      ColumnSegment(string) {  //repeatable
      }
    }
  }
}
```

Description

The properties and wrappers within the RedundancyAnalysis wrapper contain information about the repairable memory segments, the number of spare elements within a segment, and the addresses to be logged for the spare fuses.

> **Note**
> For multi-port memories, you only need to define the repair information for one port. Tessent MemoryBIST tests the ports in sequence re-using the same comparators between ports. The comparator results are cumulative and capture all defects for all ports at the end of the test.

> **Note**
> The use of an escaped identifier for AddressPort(*name*) is not supported.

Parameters

- RowSegmentRange/SegmentAddress[*bit*] : AddressPort(*name*) ; // Repeatable

  The RowSegmentRange wrapper enables you to specify the memory address bits used to define the address space for the specified row segment.

  The RowSegmentRange defines the significant row address bits that encode the RowSegmentCountRange limits in the Core/Memory/RedundancyAnalysis/ColumnSegment and Core/Memory/RedundancyAnalysis/RowSegment wrappers.

  You do not need to specify this wrapper if only one RowSegment or ColumnSegment wrapper is defined within the RedundancyAnalysis wrapper. If only one RowSegment or

ColumnSegment wrapper is present, the segment encompasses the entire memory address space.

If more than one RowSegment or ColumnSegment wrapper is defined for a same ShiftedIORange, the RowSegmentRange wrapper is required and the combined count ranges of all RowSegmentCountRange properties must encompass all possible codes defined by the SegmentAddress properties of the RowSegmentRange wrapper. Any unused codes must be explicitly indicated within the range values of the RowSegmentCountRange property. Refer to the second Example provided in the RowSegmentCountRange property description.

- ColumnSegmentRange/SegmentAddress[*bit*] : AddressPort(*name*) ; // Repeatable

  The ColumnSegmentRange wrapper enables you to define a portion of the memory address space where spare element can replace a defective element. The ColumnSegmentRange defines the significant column address bits that encode the ColumnSegmentCountRange limits inside the ColumnSegment wrapper.

  You do not need to specify this wrapper if only one ColumnSegment or RowSegment wrapper is defined within the RedundancyAnalysis wrapper. If only one ColumnSegment or RowSegment wrapper is present, the segment encompasses the entire memory address space.

  If more than one ColumnSegment or RowSegment wrapper is defined for a same ShiftedIORange, the ColumnSegmentRange wrapper is required and the combined count ranges of all ColumnSegmentCountRange properties must encompass all possible codes defined by the SegmentAddress properties of the ColumnSegmentRange wrapper. Any unused codes must be explicitly indicated within the range values of the ColumnSegmentCountRange property. Refer to the second Example provided in the ColumnSegmentCountRange property description.

  This example divides the memory address space into 2 column segments called BANK0 and BANK1. The address range of each column segment is defined by address port AD[7].

  ```
  ColumnSegmentRange{
    SegmentAddress[0]: AddressPort(AD[7]);
  }
  ColumnSegment (BANK0) {
    ColumnSegmentCountRange [1'b0:1'b0];
  }
  ColumnSegment (BANK1) {
    ColumnSegmentCountRange [1'b1:1'b1];
  }
  ```

Examples

The following example specifies four RowSegment wrappers.

- Each segment consists of two spare elements and is located in the address space defined by the address ports AD[12:10].

- The RowSegment(Bank0) is defined within the address space, whereby AD[12:10] is within 3'b000 and 3'b001.

- The RowSegment(Bank1) is defined within the address space, whereby AD[12:10] is within 3'b010 and 3'b011.

- The RowSegment(Bank2) is defined within the address space, whereby AD[12:10] is within 3'b100 and 3'b101.

- The RowSegment(Bank3) is defined within the address space, whereby AD[12:10] is within 3'b110 and 3'b111.

```
RedundancyAnalysis {
  RowSegmentRange {
    SegmentAddress[0]: AddressPort(AD[10]);
    SegmentAddress[1]: AddressPort(AD[11]);
    SegmentAddress[2]: AddressPort(AD[12]);
  }
  RowSegment (Bank0){
    NumberOfSpareElements: 2;
    RowSegmentCountRange [3'b000:3'b001];
    FuseSet {
      Fuse[2]: AddressPort(AD[9]);
      Fuse[1]: AddressPort(AD[8]);
      Fuse[0]: AddressPort(AD[7]);
    }
  }
  RowSegment (Bank1){
    NumberOfSpareElements: 2;
    RowSegmentCountRange [3'b010:3'b011];
    FuseSet {
      Fuse[2]: AddressPort(AD[9]);
      Fuse[1]: AddressPort(AD[8]);
      Fuse[0]: AddressPort(AD[7]);
    }
  }
  RowSegment (Bank2){
    NumberOfSpareElements: 2;
    RowSegmentCountRange [3'b100:3'b101];
    FuseSet {
      Fuse[2]: AddressPort(AD[9]);
      Fuse[1]: AddressPort(AD[8]);
      Fuse[0]: AddressPort(AD[7]);
    }
  }
  RowSegment (Bank3){
    NumberOfSpareElements: 2;
    RowSegmentCountRange [3'b110:3'b111];
    FuseSet {
      Fuse[2]: AddressPort(AD[9]);
      Fuse[1]: AddressPort(AD[8]);
      Fuse[0]: AddressPort(AD[7]);
    }
  }
}
```

Related Topics

RedundancyAnalysis/ColumnSegment

RedundancyAnalysis/RowSegment

Memory

RedundancyAnalysis/RowSegment

The RowSegment wrapper enables you to identify a segment of the memory address space that contains spare row elements.

Syntax

```
Core(core_name) {
  Memory {
    RedundancyAnalysis {
      RowSegment(string) { // repeatable
        NumberOfSpareElements       : int;  // default: 1
        RowSegmentCountRange range;
        FuseSet {
          Fuse[bit] :  AddressPort(name) | not AddressPort(name) |
                       LogicHigh | LogicLow; // repeatable
        }
        PinMap {
        }
      }
    }
  }
}
```

Description

The row memory segment specified by the RowSegment wrapper can cover the whole memory address space or a subset of the address space. Each segment covers all IOs. If a single segment is used, it must cover the whole address space. The wrapper is repeatable and when multiple segments are used, each segment covers a subset of the address space and the union of all segments must cover the entire memory without overlap. Specify this wrapper in the RedundancyAnalysis wrapper to implement row repair analysis.

Note

The use of an escaped identifier for AddressPort(*name*) is not supported.

Parameters

- RowSegment(*string*)

  The string value names the row segment. It must be unique identifier for the memory segment.

- NumberOfSpareElements: *int* ; // default: 1

  The NumberOfSpareElements property enables you to define the number of spare rows. The default value is 1.

  The following example specifies there are 2 spare elements in the row segment Bank0.

```
RowSegment (Bank0){
  NumberOfSpareElements: 2;
   .
   .
}
```

- RowSegmentCountRange *range* ;

  The RowSegmentCountRange property defines the portion of the row address space where the segment's spare elements can replace a defective IO/Column element.

  range is defined as [<lowRange>:<highRange>], where valid values are as follows:

  - o   lowRange— specifies the low address value in terms of the defined segment address bits used to enable the repair analysis for this segment.

  - o   highRange— specifies the high address value in terms of the defined segment address bits used to enable the repair analysis for this segment.

Valid data types for lowRange and highRange are integers or BitsValues. If any SegmentAddress bits are specified in the RowSegmentRange wrapper, the RowSegmentCountRange property defaults to a lowRange of zero to a highRange of 2^n -1, where *n* is the number of SegmentAddress bits specified.

The RowSegmentCountRange property can be specified only when at least one SegmentAddress bit is defined. When more than one RowSegment wrapper is specified, the combined count ranges of all RowSegmentCountRange properties must encompass all possible codes defined by the SegmentAddress properties of the RowSegmentRange wrapper. Any unused codes must be explicitly indicated within the range values of the RowSegmentCountRange property.

In the following example, row segment Bank0 is enabled when address port AD[11] is logic 1 and AD[10] is logic 0. Row segment Bank1 is enabled for all remaining AD[11] and AD[10] combinations.

```
RowSegmentRange {
    SegmentAddress[1]: AddressPort(AD[11]);
    SegmentAddress[0]: AddressPort(AD[10]);
}
RowSegment(Bank0) {
    RowSegmentCountRange[2'b00:2'b00];
     .
     .
     .
}
RowSegment(Bank1) {
    RowSegmentCountRange[2'b01:2'b11];
     .
     .
     .
}
```

In the example below, addr[9:8] are bank address bits, but there are only three banks, each with a ColumnSegment. In this case, there are only three useful decoded values of the SegmentAddress bits, however the remaining decoded value is added to the last segment.

```
RedundancyAnalysis {
  RowSegmentRange {
      SegmentAddress[0] : AddressPort(addr[8]);
      SegmentAddress[1] : AddressPort(addr[9]);
  }
  ColumnSegment (ALL0) {
      RowSegmentCountRange  [2'b00:2'b00];
        ...
  }
  ColumnSegment (ALL1) {
      RowSegmentCountRange  [2'b01:2'b01];
        ...
  }
  ColumnSegment (ALL2) {
      // value 2'b11 specified but is never exercised
      RowSegmentCountRange  [2'b10:2'b11];
        ...
  }
}
```

- FuseSet/Fuse[*bit*] : AddressPort(*name*) | not AddressPort(*name*) | LogicHigh | LogicLow ;

  The Fuse property in the FuseSet wrapper defines which address bits are required for the fuses to replace a defective element with a spare element. These fuse bits are defined per row segment. This is a repeatable property.

  The following example specifies two segments that consist of two spare elements each. Each spare element has a fuse register that logs the specified address bits as defined by the FuseSet wrapper. The fuse register bits are as follows:

    o Fuse[3] — logs the address driven on the port AD[9] for the defective element

    o Fuse[2] — logs the address driven on the port AD[8] for the defective element

    o Fuse[1] — logs the address driven on the port AD[7] for the defective element

    o Fuse[0] — logs the address driven on the port AD[0] for the defective element

```
RowSegment (Bank0){
  NumberOfSpareElements: 2;
  RowSegmentCountRange [1'b0:1'b0];
  FuseSet {
    Fuse[3]: AddressPort(AD[9]);
    Fuse[2]: AddressPort(AD[8]);
    Fuse[1]: AddressPort(AD[7]);
    Fuse[0]: AddressPort(AD[0]);
  }
}
RowSegment (Bank1){
  NumberOfSpareElements: 2;
  RowSegmentCountRange [1'b1:1'b1];
  FuseSet {
    Fuse[3]: AddressPort(AD[9]);
    Fuse[2]: AddressPort(AD[8]);
    Fuse[1]: AddressPort(AD[7]);
    Fuse[0]: AddressPort(AD[0]);
  }
}
```

Related Topics

RedundancyAnalysis

Memory

RedundancyAnalysis/ColumnSegment

RedundancyAnalysis/ColumnSegment

The ColumnSegment wrapper enables you to identify a segment of the memory address space that contains spare IO/Column elements.

Syntax

```
Core(core_name) {
  Memory {
    RedundancyAnalysis {
      ColumnSegment(string) { // repeatable
        RowSegmentCountRange range;
        ColumnSegmentCountRange range;
        NumberOfSpareElements   : int; // default: 1
        ShiftedIORange          : port_name, ... ;
        FuseSet {
          Fuse[bit]  : AddressPort(name) | not AddressPort(name) |
                       LogicHigh | LogicLow;
          FuseMap[HighBitRange:LowBitRange] {
            NotAllocated        : binary; // default: 0
            ShiftedIO(io)       : binary; // repeatable
          }
        }
        PinMap {
        }
      }
    }
  }
}
```

Description

The column segment defined by the ColumnSegment wrapper can cover the whole address space or a subset of the address space. It can also cover all IOs or a subset of the IOs. The wrapper is repeatable and all segments must cover the entire memory without overlapping. Specify this wrapper in the RedundancyAnalysis wrapper to implement IO/Column repair analysis.

> **Note**
> The use of an escaped identifier for ShiftedIORange:*port_name* or AddressPort(*name*) is not supported.

Parameters

- ColumnsSegment(*string*)

  The string value names the column segment. It must be unique identifier for the memory segment.

- RowSegmentCountRange *range* ;

  The RowSegmentCountRange property defines the portion of the row address space where the segment's spare elements can replace a defective IO/Column element.

range is defined as [<lowRange>:<highRange>], where valid values are as follows:

- o lowRange— specifies the low address value in terms of the defined segment address bits used to enable the repair analysis for this segment.

- o highRange— specifies the high address value in terms of the defined segment address bits used to enable the repair analysis for this segment.

Valid data types for lowRange and highRange are integers or BitsValues. If any SegmentAddress bits are specified in the RowSegmentRange wrapper, the

RowSegmentCountRange property defaults to a lowRange of zero to a highRange of 2^n -1, where *n* is the number of SegmentAddress bits specified.

The RowSegmentCountRange property can be specified only when at least one SegmentAddress bit is defined. When more than one ColumnSegment wrapper is specified, the combined count ranges of all RowSegmentCountRange properties must encompass all possible codes defined by the SegmentAddress properties of the RowSegmentRange wrapper. Any unused codes must be explicitly indicated within the range values of the RowSegmentCountRange property.

In the following example, column segment Bank0 is enabled when address port AD[10] is logic 0. Bank1 is enabled when AD[10] is logic 1.

```
RowSegmentRange {
    SegmentAddress[1]: AddressPort(AD[10]);
}
ColumnSegment(Bank0) {
    RowSegmentCountRange[1'b0:1'b0];
    .
    .
    .
}
ColumnSegment(Bank1) {
    RowSegmentCountRange[1'b1:1'b1];
    .
    .
    .
}
```

In the example below, addr[9:8] are bank address bits, but there are only three banks, each with a ColumnSegment. In this case, there are only three useful decoded values of the SegmentAddress bits, however the remaining decoded value is added to the last segment.

```
RedundancyAnalysis {
  RowSegmentRange {
      SegmentAddress[0] : AddressPort(addr[8]);
      SegmentAddress[1] : AddressPort(addr[9]);
  }
  ColumnSegment (ALL0) {
      RowSegmentCountRange  [2'b00:2'b00];
        ...
  }
  ColumnSegment (ALL1) {
      RowSegmentCountRange  [2'b01:2'b01];
        ...
  }
  ColumnSegment (ALL2) {
      // value 2'b11 specified but is never exercised
      RowSegmentCountRange  [2'b10:2'b11];
        ...
  }
}
```

- ColumnSegmentCountRange *range*

  The ColumnSegmentCountRange property defines the portion of the column address space where the segment's spare elements can replace a defective IO/Column element.

  range is defined as [<lowRange>:<highRange>], where valid values are as follows:

  - lowRange— specifies the low address value in terms of the defined segment address bits used to enable the repair analysis for this segment.

  - highRange— specifies the high address value in terms of the defined segment address bits used to enable the repair analysis for this segment.

  Valid data types for lowRange and highRange are integers or BitsValues. If any SegmentAddress bits are specified in the ColumnSegmentRange wrapper, the ColumnSegmentCountRange property defaults to a lowRange of zero to a highRange of 2^n - 1, where *n* is the number of SegmentAddress bits specified.

  The ColumnSegmentCountRange property can be specified only when at least one SegmentAddress bit is defined.

  When more than one ColumnSegment wrapper is specified, the combined count ranges of all ColumnSegmentCountRange properties must encompass all possible codes defined by the SegmentAddress properties of the ColumnSegmentRange wrapper. Any unused codes must be explicitly indicated within the range values of the ColumnSegmentCountRange property.

  In the following example, column segment Bank0 is enabled when address port AD[11] is logic 1 and AD[10] is logic 0. Column segment Bank1 is enabled for all remaining AD[11] and AD[10] combinations.

```
ColumnSegmentRange {
    SegmentAddress[1]: AddressPort(AD[11]);
    SegmentAddress[0]: AddressPort(AD[10]);
}
ColumnSegment(Bank0) {
    ColumnSegmentCountRange[2'b00:2'b00];
     .
     .
     .
}
ColumnSegment(Bank1) {
    ColumnSegmentCountRange[2'b01:2'b11];
     .
     .
     .
}
```

In the example below, addr[9:8] are bank address bits, but there are only three banks, each with a ColumnSegment. In this case, there are only three useful decoded values of the SegmentAddress bits, however the remaining decoded value is added to the last segment.

```
RedundancyAnalysis {
  ColumnSegmentRange {
      SegmentAddress[0] : AddressPort(addr[8]);
      SegmentAddress[1] : AddressPort(addr[9]);
  }
  ColumnSegment (ALL0) {
      ColumnSegmentCountRange  [2'b00:2'b00];
       ...
  }
  ColumnSegment (ALL1) {
      ColumnSegmentCountRange  [2'b01:2'b01];
       ...
  }
  ColumnSegment (ALL2) {
      // value 2'b11 specified but is never exercised
      ColumnSegmentCountRange  [2'b10:2'b11];
       ...
  }
}
```

- NumberOfSpareElements: *int* ; // default: 1

  The NumberOfSpareElements property enables you to define the number of spare columns. The default value is 1.

- ShiftedIORange: *port_name*, ... ;

  The ShiftedIORange property enables you to define a group of IO bits where spare elements can replace a faulty IO. When a defective element is within this group, a spare element is allocated to this segment. The *port_name* value is a valid data port name that does not contain escaped identifiers. Each port name can be a bused port or scalar ports separated by comma. The default range includes all IO bits of the data port.

- FuseSet/Fuse[*bit*] : AddressPort(*name*) | not AddressPort(*name*) | LogicHigh | LogicLow ;

  The repeatable FuseSet wrapper contains two possible definitions of the fuse register bits:

  o The first definition is used to map fuse register bits to address ports on the memory. In this case, the fuse register bit logs the value of this address port for the defective IO element. The mapping information is passed through the Fuse property. The use of an escaped identifier for AddressPort(*name*) is not supported.

  o The second definition is used for IO shifting and enables you to define fuse register values for identifying defective IO/columns. The defined fuse value is loaded into the fuse register for a defective IO. The mapping information for the shifted IO is passed to the tool through the FuseMap wrapper.

  _____ **Note** _____
  The FuseSet wrapper is used in the ColumnSegment wrapper and is specified only once per ColumnSegment wrapper.

- FuseSet/FuseMap[*HighBitRange:LowBitRange*]

  The FuseMap wrapper enables you to define a bit range for the fuse register. This wrapper is used to map IO ports to the fuse register.

  This example specifies the binary codes that are logged to identify each failing data IO port.

```
FuseMap[3:0] {
  ShiftedIO(Data[0]): 4'b0000;
  ShiftedIO(Data[1]): 4'b0001;
  ShiftedIO(Data[2]): 4'b0010;
  ShiftedIO(Data[3]): 4'b0011;
  ShiftedIO(Data[4]): 4'b0100;
  ShiftedIO(Data[5]): 4'b0101;
  ShiftedIO(Data[6]): 4'b0110;
  ShiftedIO(Data[7]): 4'b0111;
  ShiftedIO(Data[8]): 4'b1000;
  ShiftedIO(Data[9]): 4'b1001;
}
```

- FuseSet/FuseMap[*HighBitRange:LowBitRange*]/NotAllocated : binary;

  The NotAllocated property specifies the fuse register decode value to indicate the spare element in the column segment is not used. If this property is not present, a one bit register called the Allocation Bit is added as the MSB of the fuse register.

  These usage conditions apply:

  o This property must be specified for all or none of ColumnSegment.

  o If this property is defined, the specified value must be all zeroes.

  o If this property is defined, the same value must be specified for all ColumnSegments.

o   If this property is defined, the specified value cannot be repeated as a ShiftedIO value within the same FuseMap wrapper.

- FuseSet/FuseMap[*HighBitRange:LowBitRange*]/ShiftedIO(*io*) : binary; // repeatable

    The repeatable ShiftedIO property enables you to specify the values to be logged in the fuse register, which identifies each defective IO. *io* is a defined data port name.

    You should specify a ShiftedIO property for all memory data outputs. Otherwise, an error message is generated. You can specify the same bitString value for ShiftedIO properties of different memory outputs. However, a limitation applies.

Related Topics

RedundancyAnalysis

Memory

RedundancyAnalysis/RowSegment

PinMap

The contents of the PinMap wrapper enables you to instruct Tessent MemoryBIST to apply the repair solution, computed by the BIRA engine, to the memory repair ports or the serial repair register.

Syntax

```
Core(core_name) {
  Memory {
    RedundancyAnalysis {
      RowSegment (string) {
        PinMap {
          SpareElement {
            RepairEnable : repairPortName | RepairRegister[bit];
            Fuse[bit]    : repairPortName | RepairRegister[bit];
            LogicLow     : repairPortName | RepairRegister[bit];
          }
        }
      }
      ColumnSegment (string) {
        PinMap {
          SpareElement {
            RepairEnable : repairPortName | RepairRegister[bit];
            Fuse[bit]    : repairPortName | RepairRegister[bit];
            FuseMap[bit] : repairPortName | RepairRegister[bit];
            LogicLow     : repairPortName | RepairRegister[bit];
          }
        }
      }
    }
  }
}
```

Description

Using the SpareElement wrapper, you can specify mappings from the BISR fuse registers to the corresponding memory repair ports. These pin mappings are used to connect the BISR fuse register ports to the memory repair ports. The number of SpareElement wrappers allowed within the PinMap wrapper is equal to the NumberOfSpareElements value specified in the respective RedundancyAnalysis/ColumnSegment and RedundancyAnalysis/RowSegment wrappers.

For a memory with serial repair interface, you can specify the sequence of the internal BISR chain register within the memory.

> **Note**
> The use of an escaped identifier for *repairPortName* is not supported.

Parameters

The parameter descriptions are identical for the RowSegment(string)/PinMap/SpareElement wrapper and the ColumnSegment(string)/PinMap/SpareElement wrapper. The property path leading to the elements in SpareElement is omitted in the following.

RepairRegister[*bit*] enables you to describe the order of the internal BISR chain register for the serial BISR interface. The total BISR chain length (N) for a given memory is equal to the number of the RepairRegister[bit] properties specified in the RowSegment and ColumnSegment wrappers combined inside a memory TCD file.

The RepairRegister[0] specifies the BISR chain register that is closest to the output port with function BisrSerialData, and the RepairRegister[N-1] specifies the BISR chain register that is closest to the input port with function BisrSerialData. A single BISR chain register is generated for each memory TCD file. The RepairRegister[x] indexes must be contiguous from 0 to N-1 within a memory TCD file. All indexes between 0 and N-1 must be used, and each index can only be used once.

You must take care to describe the RepairRegister[x] indexes to match the internal memory BISR chain order specified in the memory data sheet. Any mismatch between the external and internal BISR chains ordering results in the incorrect repair data scanned inside the memory.

- RepairEnable : *repairPortName* | RepairRegister[*bit*];

  The RepairEnable property is used to identify the memory pin name or the bit of the internal BISR chain register that is used to control the memory repair function.

  repairPortName specifies memory port name used to activate the spare element.

- Fuse[*bit*] : *repairPortName* | RepairRegister[*bit*];

  The Fuse property enables you to specify the memory port of the internal BISR chain bit that controls the address of a spare row or column element. You can use this property multiple times. This property maps each bit in FuseSet:Fuse property to the corresponding repair ports or bits in the internal BISR chain of the memory. Each SpareElement:Fuse[<bitIndex>] index must match a FuseSet:Fuse [<bitIndex>] index.

  bit represents the fuse bit number. This number must start from zero and incrementally count by one up to n-1 where n is the number of Fuse bits specified.

  repairPortName represents the memory port with function BisrParallelData that logs the failing address for the spare element.

  Each ColumnSegment/FuseSet/Fuse property must have a corresponding PinMap/SpareElement/Fuse property.

- FuseMap[*bit*] : repairPortName | RepairRegister[*bit*];

  The FuseMap property is only available for the ColumnSegment/PinMap/SpareElement wrapper.

  The FuseMap property enables you to specify the memory port or internal BISR chain bit that controls the IO shifting circuit. This property maps the shifted IO fuse map bits to the corresponding repair ports or bits in the internal BISR chain of the memory. Each

SpareElement/FuseMap[*bit*] index must be within the FuseSet/FuseMap[*range*] index range.

bit represents the fuse bit number. This number must start from zero and incrementally count by one up to n-1 where n is the number of Fuse bits specified.

repairPortName represents the memory port with function BisrParallelData that logs the failing IO for the spare element.

- LogicLow : *repairPortName* | RepairRegister[*bit*];

The LogicLow property is used to instantiate a BISR register that is not associated to any BIRA register. This instantiates a BISR register that initializes to a logiclow when the asynchronous BISR clear signal is enabled. This register holds its value during a BIRA to BISR transfer.

repairPortName identifies the memory port with function BisrParallelData that is driven to the logic low value.

RepairRegister[*bit*] identifies the index of the internal BISR chain that is set to the logic low value.

This example specifies that the third bit of the BISR chain initializes to a constant logic low value during the BISR chain asynchronous reset. This register holds its value during a BIRA to BISR transfer. This register is connected to the SRowAddress[2] port of the memory with parallel BISR interface.

```
PinMap{
  SpareElement{
    LogicLow: SRowAddress[2];
    Fuse[2]:  SRowAddress[1];
    Fuse[1]:  SRowAddress[0];
    RepairEnable: SRowEn;
  }
}
```

Examples

The following example shows spare elements for implementing built-in self-repair feature.

```
PinMap {
  SpareElement {
    RepairEnable: B0_REN0;
    Fuse[0]: B0_RR0[0];
    Fuse[1]: B0_RR0[1];
    Fuse[2]: B0_RR0[2];
    Fuse[3]: B0_RR0[3];
  }
  SpareElement {
    RepairEnable: B0_REN1;
    Fuse[0]: B0_RR1[0];
    Fuse[1]: B0_RR1[1];
    Fuse[2]: B0_RR1[2];
    Fuse[3]: B0_RR1[3];
  }
}
```

Related Topics

RedundancyAnalysis

Memory

RedundancyAnalysis/RowSegment

RedundancyAnalysis/ColumnSegment

IclPorts

Identifies memory ports to be controlled or observed using TDRs of the IJTAG network.

Usage

```
IclPorts {
  DataInPort (port_name[range]) {                    // repeatable
    Attribute(attribute_name) : attribute_value ; // repeatable
  }
  DataOutPort (port_name[range]){                    // repeatable
    Attribute(attribute_name) : attribute_value ; // repeatable
  }
}
```

Description

The IclPorts wrapper identifies memory ports to be controlled or observed using TDRs of the IJTAG network. The TDRs are automatically added to the DftSpecification wrapper generated by create_dft_specification, then inserted and connected to the memory during the DFT Specification processing.

A common usage is to provide static control to the memory, such as read/write margin and power mode, prior to applying the memory test. The TDR can be accessed with an iProc applied through a ProcedureStep in the Patterns Specification.

Use the DataInPort and DataOutPort wrappers to assign attributes on the associated memory ports. The attributes instruct the create_dft_specification command to configure the TDR connections in the IjtagNetwork wrapper of the generated DFT Specification.

_____ **Note** _____
The use of an escaped identifier for *port_name* is not supported.
_____

The create_dft_specification command considers the following ICL attributes. Other valid attributes may be specified, but are ignored by create_dft_specification.

- default_load_value

- tessent_use_in_dft_specification

- tessent_enable_group

- tessent_common_tdr_source

Tessent MemoryBIST generates an ICL view of the memory module. The attributes defined in the DataInPort and DataOutPort wrappers are populated into the memory ICL module.

The DataInPort and DataOutPort wrappers are repeatable. Each wrapper may specify one or more attributes. The specified attributes are cumulative and if duplicated, the last value specified is retained.

Arguments
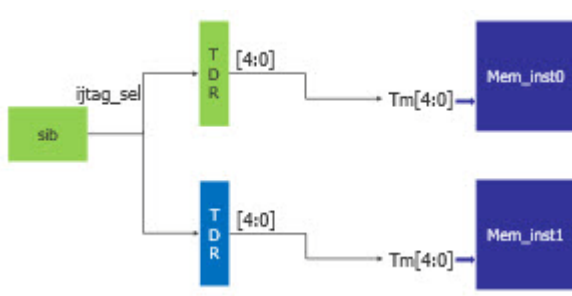
- DataInPort(*port_name[range]*)

  The port_name property must refer to a Port wrapper defined with Function:None and Direction:Input. The optional *range* element identifies the bit range for bussed ports.

- DataOutPort(*port_name[range]*)

  The port_name property must refer to a Port wrapper defined with Function:None and Direction:Output. The optional *range* element identifies the bit range for bussed ports.

- Attribute(*attribute_name*) : *attribute_value* ;

  A repeatable property and value pair that is specified in the DataInPort or DataOutPort wrapper. A port or bus bit range specified in many DataInPort or DataOutPort wrappers collects all of the attributes specified within each wrapper for those ports. If an attribute is duplicated, the value that is specified last is retained.

Examples
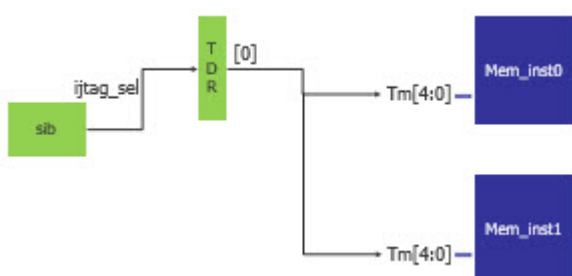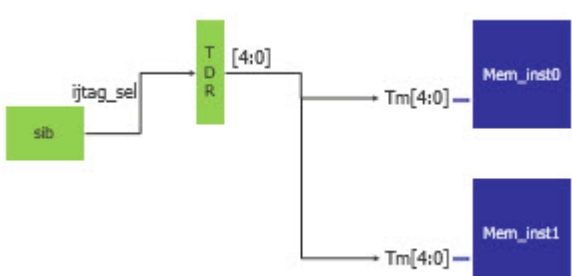
Example TDR DFT Configurations

Table A-7 shows the TDR configurations that are created with the IclPorts specifications provided, using the tessent_enable_group and tessent_common_tdr_source attributes. Mem_inst0 and Mem_inst1 are different instances of the same memory module. Where indicated, Mem_inst2 is an instance of a different memory module.

Table A-7. Example TDR Configurations Using tessent_enable_group and tessent_common_tdr_source
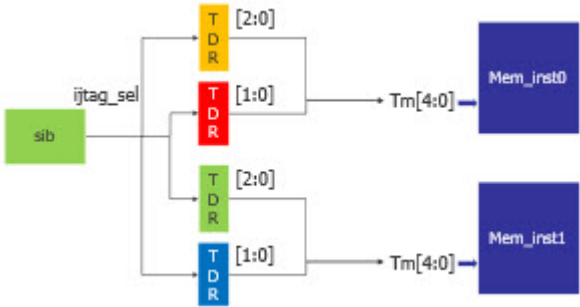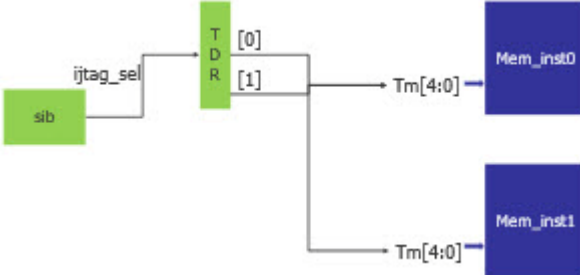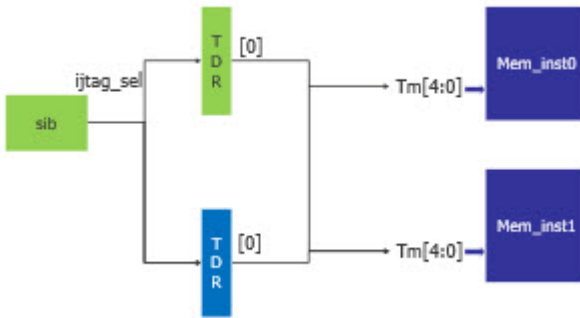
Specification	Resulting DFT
Example Implementations Controlling the Full Bus	

Bussed port with tessent_enable_group:
 	
``
DataInPort(Tm[4:0]) {
  Attribute(tessent_enable_group):group0;
}
``
	
	Green TDR[4:0] drives Tm[4:0] of Mem_inst0.
	Blue TDR[4:0] drives Tm[4:0] of Mem_inst1.

**Table A-7. Example TDR Configurations Using tessent_enable_group and
tessent_common_tdr_source (cont.)**
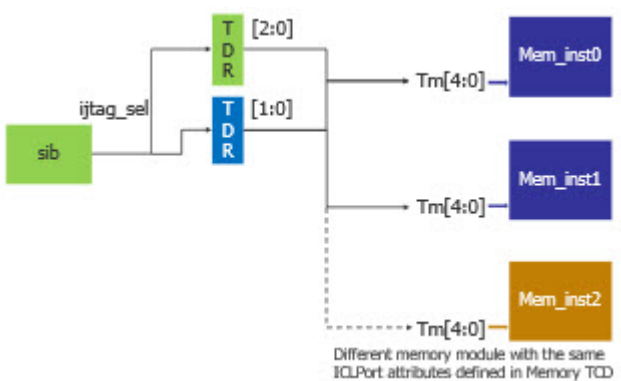
Specification	Resulting DFT

Bussed port with tessent_common_tdr_source:
 	
``DataInPort(Tm[4:0]) {
  Attribute(tessent_common_tdr_source):tm_all;
}```
 	
Bussed port with both attributes:
 	
``DataInPort(Tm[4:0]) {
  Attribute(tessent_common_tdr_source):tm_all;
  Attribute(tessent_enable_group):group0;
}```
	
	Green TDR[0] drives Tm[4:0] of all memory instances.

Bit-wise port with tessent_common_tdr_source:
 	
``DataInPort(Tm[4]) {
  Attribute(tessent_common_tdr_source):tm_4;
}
...
DataInPort(Tm[0]) {
  Attribute(tessent_common_tdr_source):tm_0;
}```
 	
Bit-wise port with both attributes:
 	
``DataInPort(Tm[4]) {
  Attribute(tessent_common_tdr_source):tm_4;
  Attribute(tessent_enable_group):group0;
}...
DataInPort(Tm[0]) {
  Attribute(tessent_common_tdr_source):tm_0;
  Attribute(tessent_enable_group):group0;
}```
	
	Green TDR[4:0] drives Tm[4:0] of all memory instances.

Example Implementations Controlling Portions of a Bus	

Table A-7. Example TDR Configurations Using tessent_enable_group and tessent_common_tdr_source (cont.)

Specification	Resulting DFT

Split Bus with tessent_enable_group:
 	
``
DataInPort(Tm[4:2]) {
  Attribute(tessent_enable_group):group0;
}
DataInPort(Tm[1:0]) {
  Attribute(tessent_enable_group):group1;
  Attribute(default_load_value) : 2'b11;
}
``
	
	Orange TDR[2:0] drives Tm[4:2] of Mem_inst0.
	Red TDR[1:0] drives Tm[1:0] of Mem_inst0.
	Green TDR[2:0] drives Tm[4:2] of Mem_inst1.
	Blue TDR[1:0] drives Tm[1:0] of Mem_inst1.

Split Bus with tessent_common_tdr_source:
 	
``
DataInPort(Tm[4:2]) {
  Attribute(tessent_common_tdr_source):tm_4_2;
}
DataInPort(Tm[1:0]) {
  Attribute(tessent_common_tdr_source):tm_1_0;
}
``
	
	TDR[0] drives Tm[4:2] of all memory instances.
	TDR[1] drives Tm[1:0] of all memory instances.

Split Bus with both attributes:
 	
``
DataInPort(Tm[4:2]) {
  Attribute(tessent_common_tdr_source):tm_4_2;
  Attribute(tessent_enable_group) : group0;
}
DataInPort(Tm[1:0]) {
  Attribute(tessent_common_tdr_source):tm_1_0;
  Attribute(tessent_enable_group) : group1;
}
``
	
	Green TDR[0] drives Tm[4:2] of all memory instances.
	Blue TDR[0] drives Tm[1:0] of all memory instances.

Table A-7. Example TDR Configurations Using tessent_enable_group and tessent_common_tdr_source (cont.)

Specification	Resulting DFT

Bit-wise split bus with both attributes:
 	
``
DataInPort(Tm[4]) {
  Attribute(tessent_common_tdr_source):tm_4;
  Attribute(tessent_enable_group):group0;
}
DataInPort(Tm[3]) {
  Attribute(tessent_common_tdr_source):tm_3;
  Attribute(tessent_enable_group):group0;
}
DataInPort(Tm[2]) {
  Attribute(tessent_common_tdr_source):tm_2;
  Attribute(tessent_enable_group):group0;
}
DataInPort(Tm[1]) {
  Attribute(tessent_common_tdr_source):tm_1;
  Attribute(tessent_enable_group):group1;
}
DataInPort(Tm[0]) {
  Attribute(tessent_common_tdr_source):tm_0;
  Attribute(tessent_enable_group):group1;
}
``
	
	Green TDR[2:0] drives Tm[4:2] of all memory instances.
	
	Blue TDR[1:0] drives Tm[1:0] of all memory instances.

Example Implementation of TDR Sharing Between Different Memory Modules With Partially Shared Attribute Values	

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Table A-7. Example TDR Configurations Using tessent_enable_group and tessent_common_tdr_source (cont.)

Specification	Resulting DFT

Mem_inst0/Mem_inst1 IclPorts wrapper:
 	
``
DataInPort(Tm[4]) {
  Attribute(tessent_common_tdr_source):tm_4;
  Attribute(tessent_enable_group):group0;
}
DataInPort(Tm[3]) {
  Attribute(tessent_common_tdr_source):tm_3;
  Attribute(tessent_enable_group):group0;
}
DataInPort(Tm[2]) {
  Attribute(tessent_common_tdr_source):tm_2;
  Attribute(tessent_enable_group):group0;
}
DataInPort(Tm[1]) {
  Attribute(tessent_common_tdr_source):tm_1;
  Attribute(tessent_enable_group):group1;
}
DataInPort(Tm[0]) {
  Attribute(tessent_common_tdr_source):tm_0;
  Attribute(tessent_enable_group):group1;
}
``
 	
Mem_inst2 IclPorts wrapper:
 	
``
DataInPort(Tm[4]) {
  Attribute(tessent_common_tdr_source):tm_4;
  Attribute(tessent_enable_group):group3;
}
DataInPort(Tm[3]) {
  Attribute(tessent_common_tdr_source):tm_3;
  Attribute(tessent_enable_group):group3;
}
DataInPort(Tm[2]) {
  Attribute(tessent_common_tdr_source):tm_2;
  Attribute(tessent_enable_group):group3;
}
DataInPort(Tm[1]) {
  Attribute(tessent_common_tdr_source):tm_1;
  Attribute(tessent_enable_group):group1;
}
DataInPort(Tm[0]) {
  Attribute(tessent_common_tdr_source):tm_0;
  Attribute(tessent_enable_group):group1;
}
``
	
	Green TDR[2:0] drives Tm[4:2] of Mem_inst0 and Mem_inst1.
	
	Blue TDR[1:0] drives Tm[1:0] of all memory instances.
	
	Red TDR[2:0] drives Tm[4:2] of Mem_inst2.

Example 2

The Memory TCD specifies the input port *in1* to be sourced by one TDR. The first DataInPort wrapper includes all port bits. The second DataInPort wrapper subsequently excludes bit 0.

```
IclPorts {
    DataInPort(in1[4:0]) {
      Attribute(tessent_use_in_dft_specification) : auto;
    }
    DataInPort(in1[0:0]) {
      Attribute(tessent_use_in_dft_specification) : false;
    }
  }
```

The generated DftSpecification configures a 4-bit TDR controlling the 4 MSBs of the memory input port.

```
Tdr(sri_tdr1) {
    DataOutPorts {
      connection(0) : dpmem1/in1[1];
      connection(1) : dpmem1/in1[2];
      connection(2) : dpmem1/in1[3];
      connection(3) : dpmem1/in1[4];
    }
    reset_value : 4'b0000;
  }
```

Example 3

The Memory TCD specifies the input port *in1* to be sourced by separate TDRs. The first DataInPort wrapper assigns the 3 MSBs to enable group *mode1*. The second DataInPort assigns the 2 LSBs to enable group *mode2* and sets the default TDR value.

```
IclPorts {
  DataInPort(in1[4:2]) {
    Attribute(tessent_enable_group) : mode1;
  }
  DataInPort(in1[1:0]) {
    Attribute(tessent_enable_group) : mode2;
    Attribute(default_load_value) : 2'b11;
  }
}
```

The generated DftSpecification configures two TDRs to be connected to the memory port. The default value of the 2-bit TDR is 2'b11.

```
      // Enable group "mode1"
    Tdr(sri_tdr2) {
      DataOutPorts {
        connection(0) : dpmem1/in1[2];
        connection(1) : dpmem1/in1[3];
        connection(2) : dpmem1/in1[4];
      }
      reset_value : 3'b000;
    }
    // Enable group "mode2"
    Tdr(sri_tdr3) {
      DataOutPorts {
        connection(0) : dpmem1/in1[0];
        connection(1) : dpmem1/in1[1];
      }
      reset_value : 2'b11;
    }
```

Example 4

The Memory TCD specifies that each bit of the *in1* bus, for all instances of this memory, should be driven by the same TDR.

```
IclPorts {
  DataInPort(in1[0]) {
    Attribute(tessent_common_tdr_source): config_mem0;
  }
  DataInPort(in1[1]) {
    Attribute(tessent_common_tdr_source): config_mem1;
  }
}
```

The generated DftSpecification configures one TDR to be connected to the memory ports of all instances:

```
Tdr(sri_tdr1) {
  DataOutPorts {
    port_naming : config_mem1,config_mem0;
    connection(1): dpmem1/in1[1];
    connection(1): dpmem2/in1[1];
    connection(1): dpmem3/in1[1];
    connection(0): dpmem1/in1[0];
    connection(0): dpmem2/in1[0];
    connection(0): dpmem3/in1[0];
  }
}
```

MemoryCluster

Specifies the memory cluster behavior for the specific module_name.

Syntax

```
Core(core_name) {
  MemoryCluster {
     Port(port_name) {
     }
     MemoryBistInterface(id) {
     }
     IclPorts {
     }
  }
}
```

Description

Specifies the memory cluster behavior for the specific module_name. Such descriptions are automatically read in during module matching. See the set_design_sources -format tcd_memory command description for information about where they are looked for. See the read_core_descriptions command description to learn how to read them in explicitly. See the set_module_matching_options command description for information about the name matching process.

_____ **Note** _____

The legacy LogicVision MemoryClusterTemplate library format is supported natively and is automatically translated into this format when read. You only need to read one memory description for a given MemoryCluster.

To see the content of a read-in Core(ModuleName)/Memory, use the "report_config_data Core(ModuleName)/MemoryCluster -partition tcd" command.

To see the supported syntax, use the "report_config_syntax Core/MemoryCluster" command.

Parameters

There are no parameters specified for this wrapper

MemoryCluster/Port

The Shared Bus memory cluster TCD Port wrapper defines the direction, function, and bus parameters for a signal port that is global to all Shared Bus memory interfaces within a Shared Bus memory cluster.

Usage

```
Core(<core_name>) {
  MemoryCluster {
    Port(port_name) {
      Direction          : Input | Output | InOut;
      Function           : None | Clock | ScanTest | BistOn |
                           InterfaceReset ;
      SafeValue          : X | 1 | 0 ;
    }
  }
}
```

Description

The Shared Bus memory cluster TCD Port wrapper defines the direction, function, and bus parameters for a signal port that is global to all the Shared Bus memory interfaces, as defined by a MemoryBistInterface wrapper, within a Shared Bus memory cluster TCD. The MemoryCluster/Port wrapper is repeatable for each global signal port needed.

The Arguments section defines properties that are unique to the Shared Bus memory cluster TCD syntax for the Port wrapper.

Note

The use of an escaped identifier for *port_name* is not supported.

Arguments

- Function : *function_type*;

  The Function property specifies the function of the signal port. The default value is None. Table A-8 describes the valid *function_type* values that can be specified.

Table A-8. Valid Port Function Values for Shared Bus Memory Cluster TCD

Function	Description
None	Specifies a port that does not need to be controlled during memory BIST. This is the default. The functional connection to this port are preserved. Use SafeValue to control the memory port value during the controller assembly simulation. For pattern generation, a ProcedureStep in the PatternsSpecification may be required to set the proper value when memory BIST is inserted into the design. Tessent MemoryBIST does not intercept the memory port.
Clock	Specifies a memory clock port.

Table A-8. Valid Port Function Values for Shared Bus Memory Cluster TCD

Function	Description
ScanTest	Specifies the port that configures the embedded test logic to enable scan testing. Typically, the port disables the memory's tri-state outputs or enables the memory bypass mode.
InterfaceReset	Specifies a signal that is used to reset all Shared Bus memory interfaces in the cluster.
BistOn	Specifies that the port is used to control the signals (data/address/control, but not clock) selection in the memory. A port of this function is connected to the BIST_ON signal of the controller.

MemoryBistInterface

The MemoryBistInterface wrapper contains information about ports, logical memories, and access codes associated with the specified Shared Bus interface.

Syntax

```
Core(core_name) {
  MemoryCluster {
    MemoryBistInterface(id) {
      Port(port_name) {
      }
      MemoryGroupAddressDecoding(GroupAddress) | (Address[x:y]) {
      }
      LogicalMemoryToInterfaceMapping(logical_memory_id) {
      }
    }
  }
}
```

Description

The MemoryBistInterface wrapper contains information about ports, logical memories, and access codes associated with the specified Shared Bus MemoryCluster.

Parameters

- *id*

  A unique identifier for the MemoryBistInterface within the MemoryCluster.

MemoryBistInterface/Port

The MemoryBistInterface Port wrapper defines the direction, function, and bus parameters for a signal port on a Shared Bus interface.

Usage

```
Core(<core_name>) {
  MemoryCluster {
    MemoryBistInterface(id) {
      Port(port_name) {
        Direction          : Input | Output | InOut;
        Function           : None | MemoryGroupAddress | WriteAddress |
                             Address | ReadAddress | InterfaceReset |
                             ConfigurationData | TestPortSelect |
                             Data | Clock | WriteEnable | ReadEnable |
                             GroupWriteEnable ;
        SafeValue          : X | 1 | 0 ;
        LogicalPort        : cluster_interface_logical_port_id, ... ;
      }
    }
  }
}
```

Description

The MemoryBistInterface Port wrapper defines the direction, function, and bus parameters for a signal port on a Shared Bus interface. The wrapper is repeatable and used within the Shared Bus memory cluster TCD.

The Arguments section defines properties that are unique to the Shared Bus memory cluster TCD syntax for the interface Port wrapper. Refer to the memory TCD Port wrapper for descriptions of the common properties.

> **Note**
> The use of an escaped identifier for *port_name* is not supported.

Arguments

- Function : *function_type* ;

  The Function property specifies the function of the signal port. The default value is None. Table A-9 describes the valid values that are unique to the Shared Bus memory cluster TCD interface ports. Refer to the memory TCD Port wrapper for descriptions of the remaining Function properties that are common.

Table A-9. Valid Port Function Values Unique to the Shared Bus Memory Cluster TCD Interface

Function	Description
MemoryGroupAddress	Specifies a port that is used to select a memory.

Table A-9. Valid Port Function Values Unique to the Shared Bus Memory Cluster TCD Interface (cont.)

Function	Description
Address	Specifies an address port shared by the cluster's logical memories.[2]
WriteAddress	Specifies a signal that provides the write address for a single-port logical memory.[1] This Function is not allowed if there are any memory cluster TCD PinMappings/LogicalMemoryLogicalPort properties to indicate mapping between logical ports.[2]
ReadAddress	Specifies signal that provides the read address for a single-port logical memory.[1] This Function is not allowed if there are any memory cluster TCD PinMappings/LogicalMemoryLogicalPort properties to indicate mapping between logical ports.[2]
WriteEnable	Specifies a write enable port shared by the cluster's logical memories.
GroupWriteEnable	Specifies a write enable port that controls one or more bits in the data path to the logical memories.
ReadEnable	Specifies a read enable port shared by the cluster's logical memories.
Clock	Specifies a clock port shared by the cluster's logical memories.
Data	Specifies a signal that provides data to the logical memories. When Data is specified for the Function property, the Direction property indicates if Data is an input, output, or bidirectional port.
InterfaceReset	Specifies a signal that is used to reset the Shared Bus memory interface.
ConfigurationData	Specifies a port that is used to configure access to a logical memory.
TestPortSelect	Specifies a port that is used to select a multi-port logical memory port that is multiplexed to a cluster interface port. This select port can be declared as a separate cluster interface port or share the bus port name with the MemoryGroupAddress port. In this case, the TestPortSelect bits must be contiguously placed on either the LSB or MSB side of the joint physical port.

1. When you have both 1R1W and 1RW memories on the same Shared Bus memory cluster and there are Address and WriteAddress ports functions defined on the cluster MemoryBistInterface for driving the address ports of the 1R1W memory, the 1RW memories should connect to the address port with function Address and use *LogicalMemoryAddress[]: InterfaceAddress[]* property to define the address port mapping.

2. The use of Function:Address in combination with LogicalPort is preferred over Function:WriteAddress or Function:ReadAddress. The former combination is consistent with the memory library file usage for memories that are not part of a Shared Bus memory cluster.

- LogicalPort : *cluster_interface_logical_port_id, ...* ;

  A property with a repeatable string value that groups address, data, and control signals of the memory cluster interface for logical memories that contain multiple ports. When specified, port functions of WriteAddress and ReadAddress are not allowed. Logical memories with multiple ports are associated to a particular cluster interface logical port by associating the specified *cluster_interface_logical_port_id* in the LogicalMemoryToInterfaceMapping/ PinMappings/LogicalMemoryLogicalPort() : InterfaceLogicalPort(*cluster_interface_logical_port_id*) property assignment in the Memory Cluster TCD.

MemoryBistInterface/ LogicalMemoryToInterfaceMapping

The LogicalMemoryToInterfaceMapping wrapper specifies the associations between the logical memory and the shared bus interface ports

Syntax

```
Core(core_name) {
  MemoryCluster {
    MemoryBistInterface(id) {
      LogicalMemoryToInterfaceMapping(logical_memory_id) {
        MemoryInstanceName : instance_name ;
        ConfigurationData  : binary  ; // default: 0
        MemoryTemplate     : template_name ;
        PipelineDepth      : integer        ; // default: 0
        PinMappings {       // repeatable for multi-port logical memory
          }
      }
    }
  }
}
```

Description

The LogicalMemoryToInterfaceMapping wrapper is used in the MemoryCluster/ MemoryBistInterface wrapper and is part of the Shared Bus memory cluster TCD.

Parameters

- LogicalMemoryToInterfaceMapping/MemoryInstanceName : *instance_name* ;

  The MemoryInstanceName property identifies the hierarchical path to the logical memory in the MemoryCluster module. Tessent MemoryBIST automatically constructs the absolute path to the memory instance by concatenating the memory cluster instance path with the specified MemoryInstanceName.

- LogicalMemoryToInterfaceMapping/ConfigurationData : *binary* ;

  The ConfigurationData property in the MemoryCluster core library wrapper specifies a binary value that must be applied on the port with the ConfigurationData port function when the logical memory is selected.

- LogicalMemoryToInterfaceMapping/MemoryTemplate : *template_name* ;

  Defines the logical memory core library name.

- LogicalMemoryToInterfaceMapping/PipelineDepth : *integer* ;

  The PipelineDepth property specifies the total number of pipeline stages that surround the logical memory. For example, if a memory has one stage of pipeline registers on the data inputs and one stage of pipeline registers on the data outputs, the pipeline depth for this memory is 2. In this example, the assumption is that the logical memory itself is a synchronous memory and only introduces a delay of one clock cycle. The value specified for PipelineDepth in the memory cluster library file for a given logical memory overrides

any value specified in the corresponding logical (or physical) memory library file. Therefore, the PipelineDepth value must include the contribution of all pipeline stages inside the memory cluster.

MemoryGroupAddressDecoding

The MemoryGroupAddressDecoding wrapper specifies the access method and decode value to access a given memory.

This wrapper has two usages, one within the Shared Bus memory cluster TCD and another within the Shared Bus logical memory TCD.

Syntax

Usage 1: Memory Cluster TCD

The MemoryGroupAddressDecoding wrapper in the Memory Cluster Tessent Core Description specifies the access method and decode value to access each logical memory.

```
Core(core_name) {
  MemoryCluster {
    MemoryBistInterface(id) {
      MemoryGroupAddressDecoding(GroupAddress) | (Address[x:y]) {
        Code(binaryValue) : logical_memory_id [,logical_memory_id...] ;
      }
    }
  }
}
```

Usage 2: Logical Memory TCD

The MemoryGroupAddressDecoding wrapper in the Logical Memory Tessent Core Description contains the information about one or more physical memories that form the address space of the logical memory.

```
Core(core_name) {
  Memory {
    MemoryGroupAddressDecoding (Address [x:y]) {
      Code(binaryValue) : physical_memory_id [,physical_memory_id...] ;
    }
  }
}
```

Description

Usage 1: The MemoryGroupAddressDecoding wrapper is used in the MemoryCluster/MemoryBistInterface wrapper of the Shared Bus memory cluster TCD as shown in Figure 6-5.

Usage 2: The MemoryGroupAddressDecoding wrapper is used in the Shared Bus logical memory TCD as shown in Figure 6-6.

Usage Conditions:

- Use the GroupAddress decoding method if the memory cluster module has a port that is used to select logical memories.

- Use the Address[$x{:}y$] decoding method if the memory cluster module selects logical memories based on address bus ranges.

- The specified decoding method indicates which port to use to enable the logical memories that the Code property specifies. If the GroupAddress decoding method is specified, the size of the Code property's binary value must match the width of the port specified with the Function type of MemoryGroupAddress.

- If you have a logical memory that includes multiple physical memories and you are using BISR, additional code bits are required. See "Memory Cluster Library File Preparation for BIRA and BISR" for more information.

- If you use the Address[*x:y*] decoding method, you cannot specify address bits that map to the logical memory address in the MemoryBistInterface/ LogicalMemoryToInterfaceMapping wrapper.

- **Usage 2**: You cannot specify address bits that map to the physical memory address in the PhysicalToLogicalMapping wrapper.

Parameters

Usage 1

- Code(*binaryValue*) : *logical_memory_id* [,*logical_memory_id*...] ;

  A required, repeatable property that specifies the name for each logical memory that is enabled with a particular port value or address as defined by *binaryValue*.

Usage 2

- Code(*binaryValue*) : *physical_memory_id* [,*physical_memory_id*...] ;

  A required, repeatable property that specifies one or more physical memories, which form the address space of the logical memory, that are enabled with a particular port value or address as defined by *binaryValue*.

Examples

The following example shows how to use the Address[*x:y*] decoding method:

```
MemoryGroupAddressDecoding(Address[4:3]){
  code(2'b00): MEM_0;
  code(2'b01): MEM_1;
  code(2'b10): MEM_2;
  code(2'b11): MEM_3;
}
```

PhysicalToLogicalMapping

The PhysicalToLogicalMapping wrapper specifies the associations between the physical memory ports and the logical memory ports, and is used in the Shared Bus logical memory TCD. One PhysicalToLogicalMapping wrapper is required for each physical memory that forms the logical memory.

Syntax

```
Core(core_name) {
  Memory {
    PhysicalToLogicalMapping (physical_memory_id) { // Repeatable
      MemoryTemplate: physicalTemplateName;
      MemoryInstanceName : instance_name ;
      PinMappings {       // repeatable for multi-port logical memory
      }
    }
  }
}
```

Description

The PhysicalToLogicalMapping wrapper is used in the logical memory core library file to map the ports of the physical memory to the ports of the logical memory. This wrapper is used in the Shared Bus logical memory TCD.

Parameters

- MemoryTemplate : *physicalTemplateName*;

  Defines the logical memory core library name.

- MemoryInstanceName : *instance_name* ;

  The MemoryInstanceName property identifies the hierarchical path to the physical memory in the MemoryCluster module. Tessent MemoryBIST automatically constructs the absolute path to the memory instance by concatenating the memory cluster instance path with the specified MemoryInstanceName.

PinMappings

The PinMappings wrapper is used within the Memory Cluster Library and the Logical Memory Library.

Usage

Usage 1: Memory Cluster Library

```
Core(core_name) {
  MemoryCluster {
    MemoryBistInterface(id) {
      LogicalMemoryToInterfaceMapping(logical_memory_id) {
        PinMappings {  // repeatable for multi-port logical memory
          TestPortSelect : binary ; // default: 1'b0
          LogicalMemoryLogicalPort(lm_logical_port_id):
              InterfaceLogicalPort(cluster_interface_logical_port_id);
          LogicalMemoryDataInput[indexList]:
              InterfaceDataInput[indexList];
          LogicalMemoryDataOutput[indexList]:
              InterfaceDataOutput[indexList];
          LogicalMemoryAddress[indexList]:
              InterfaceAddress[indexList];
          LogicalMemoryWriteAddress[indexList]:
              InterfaceWriteAddress[indexList];
          LogicalMemoryReadAddress[indexList]:
              InterfaceReadAddress[indexList];
          LogicalMemoryGroupWriteEnable[indexList]:
              InterfaceGroupWriteEnable[indexList];
        }
      }
    }
  }
}
```

Usage 2: Logical Memory Library

```
Core(core_name) {
  Memory {
    PhysicalToLogicalMapping(physical_memory_id) { // Repeatable
      MemoryTemplate: physicalTemplateName;
      PinMappings { // repeatable for multi-port logical memory
        PhysicalMemoryLogicalPort(pm_logical_port_id) :
            LogicalMemoryLogicalPort(lm_logical_port_id);
        PhysicalMemoryDataInput[indexList] :
            LogicalMemoryDataInput[indexList];
        PhysicalMemoryDataOutput[indexList] :
            LogicalMemoryDataOutput[indexList];
        PhysicalMemoryAddress[indexList] :
            LogicalMemoryAddress[indexList];
        PhysicalMemoryWriteAddress[indexList] :
            LogicalMemoryWriteAddress[indexList];
        PhysicalMemoryReadAddress[indexList] :
            LogicalMemoryReadAddress[indexList];
        PhysicalMemoryGroupWriteEnable[indexList] :
            LogicalMemoryGroupWriteEnable[indexList];
      }
    }
  }
}
```

Description

The PinMappings wrapper in the memory cluster library specifies the mappings between the logical memory ports and the shared bus interface ports. Each port declared with the MemoryBistInterface/Port wrapper must have a mapping specified. The PinMappings wrapper is repeated and specified for each logical port for logical memories with multiple ports. The PinMappings wrapper in the logical memory library specifies the mappings of the physical memory ports to the logical memory ports.

If there are any PinMappings/LogicalMemoryLogicalPort properties present in the memory cluster library, the presence of LogicalPort properties in the memory cluster library and logical memory library is implied for mapping between logical ports. In this case, the port functions ReadAddress and WriteAddress are not allowed; however multiple ports with function Address are allowed and are distinguished by the LogicalPort label for connectivity.

The presence of the PinMappings/LogicalMemoryLogicalPort property in the memory cluster library also requires the presence of the PinMappings/PhysicalMemoryLogicalPort property for completeness. An exception can occur in the case where a shared bus cluster contains a mixture of multi-port and single port memories. Because of the presence of multi-port logical memories, the LogicalPort and PinMappings/LogicalMemoryLogicalPort properties have to be used versus the use of ReadAddress and WriteAddress port functions for single port implementations. However, the memory vendor may not have the LogicalPort property specified in the single port physical memory library since there is a single logical port and the LogicalPort property is redundant. In this case, you have to specify a dash or NULL in the PinMappings/LogicalMemoryLogicalPort and PinMappings/PhysicalMemoryLogicalPort properties, as demonstrated in the following example.

```
MemoryClusterTemplate(C) {
 MemoryBistInterface(I1) {
  LogicalMemoryToInterfaceMapping(lm1) {
    PinMappings {
     LogicalMemoryLogicalPort(-) : InterfaceLogicalPort(A);
    }
   }
  }
 }
 MemoryTemplate(logical_mem) {
  PhysicalToLogicalMapping {
   PinMappings {
    PhysicalMemoryLogicalPort(-): LogicalMemoryLogicalPort(RW1);
    // Use this property when the LogicalPort property is defined in
    // this logical memory, don't use it otherwise
   }
  }
 }
```

If there are no memory cluster PinMappings/LogicalMemoryLogicalPort properties specified, there can only be one PinMappings wrapper present in either the memory cluster library or logical memory library. The ReadAddress or WriteAddress port functions should be specified in addition to Address port functions in the memory cluster library, as well as the PinMappings/ LogicalMemory{Read | Write}Address and PinMappings/PhysicalMemory{Read | Write}Address properties in the memory cluster library and logical memory library, respectively.

For clusters implementing multi-port logical memories where you specify multiple PinMappings wrappers, the following usage conditions apply:

1. All memory cluster LogicalMemoryToInterfaceMapping/PinMappings wrappers define the LogicalMemoryLogicalPort property.

2. All logical memory PhysicalToLogicalMapping/PinMappings wrappers define the PhysicalMemoryLogicalPort property.

3. All LogicalMemoryToInterfaceMapping/PinMappings/LogicalMemoryLogicalPort property ids (for example, "RW1") are unique within a LogicalMemoryToInterfaceMapping wrapper.

4. All the PhysicalToLogicalMapping/PinMappings/PhysicalMemoryLogicalPort property ids are unique within a PhysicalToLogicalMapping wrapper.

5. All the logical memory TCD Port/LogicalPort identifiers have the associated logical memory cluster LogicalMemoryToInterfaceMapping/PinMappings wrapper defined. This ensures all the logical ports of a given logical memory instance have an explicit association to a cluster interface logical port defined.

6. All the physical memory TCD Port/LogicalPort identifiers have the associated logical memory PhysicalToLogicalMapping/PinMappings wrapper defined. This ensures all the logical ports of a given physical memory instance have an explicit association to the logical memory logical port.

7. There are no memory cluster MemoryBistInterface/Port wrappers defining ports with function ReadAddress or WriteAddress. All address ports must be of the Address function type.

8. All the memory cluster MemoryBistInterface/Port wrappers defining ports that are LogicalPort-specific (for example, functions Address, Data (Input/Output), ReadEnable, WriteEnable, and GroupWriteEnable) have a LogicalPort property defined.

9. All the logical memory TCD, LogicalPort-specific ports have an associated port of the same function and LogicalPort id on the memory cluster MemoryBistInterface.

Arguments

Usage 1

- LogicalMemoryLogicalPort(*lm_logical_port_id*) : InterfaceLogicalPort(*cluster_interface_logical_port_id*);

  Specifies the association of the PinMappings wrapper for the logical port of a multi-port logical memory, specified by *lm_logical_port_id*, to the MemoryBistInterface/Port/ LogicalPort:*cluster_interface_logical_port_id* of the memory cluster interface. The *lm_logical_port_id* identifier must correspond to a unique LogicalPort label defined in the logical memory TCD.

- TestPortSelect : *binary* ;

  This property is used together with the port specified in the memory cluster TCD with the MemoryBistInterface/Port/Function of type TestPortSelect. The value *binary* must match the size of the port defined in the memory cluster interface. The use of TestPortSelect decreases route congestion at the cluster interface by multiplexing logical ports of a logical memory to the memory cluster interface. The TestPortSelect : *binary* property is used to drive the select inputs of the logical port multiplexers. In cases where there is only one single-direction port, such as the write port of a 2R1W memory, the value of the TestPortSelect property may be "x".

- *LogicalMemoryDataInput*[]: *InterfaceDataInput*[] ;

  Maps the data input port of the logical memory to the data input port of the memory cluster interface.

- *LogicalMemoryDataOutput*[]: *InterfaceDataOutput*[] ;

  Maps the data output port of the logical memory to the data output port of the memory cluster interface.

- *LogicalMemoryAddress*[]: *InterfaceAddress*[] ;

  Maps the address port of the logical memory to the address port of the memory cluster interface. Only use this mapping with single-port memories (1RW).

- *LogicalMemoryWriteAddress*[]: *InterfaceWriteAddress*[] ;

  Maps the write address port of the logical memory to the write address port of the memory cluster interface. Use this mapping with dual-port memories (1R1W).

- *LogicalMemoryReadAddress*[]: *InterfaceReadAddress*[] ;

  Maps the read address port of the logical memory to the read address port of the memory cluster interface. Use this mapping with dual-port memories (1R1W).

- *LogicalMemoryGroupWriteEnable*[]: *InterfaceGroupWriteEnable*[] ;

  Maps the group write enable port of the logical memory to the group write enable port of the memory cluster interface.

  > **Note**
  >
  > When you have both 1R1W and 1RW memories on the same Shared Bus memory cluster and there are Address and WriteAddress ports functions defined on the cluster MemoryBistInterface for driving the address ports of the 1R1W memory, the 1RW memories should connect to the address port with function Address and use *LogicalMemoryAddress[]: InterfaceAddress[]* property to define the address port mapping.

Usage 2

- PhysicalMemoryLogicalPort(*pm_logical_port_id*) : LogicalMemoryLogicalPort(*lm_logical_port_id*) ;

  Specifies the association of the PinMappings wrapper for the logical port of a physical memory, specified by *pm_logical_port_id*, to a unique Port/ LogicalPort:*cluster_interface_logical_port_id* of the logical memory TCD. The *pm_logical_port_id* identifier must correspond to a unique LogicalPort label defined in the physical memory TCD.

- *PhysicalMemoryDataInput*[]: *LogicalMemoryDataInput*[] ;

  Maps the data input port of the physical memory to the data input port of the logical memory.

- *PhysicalMemoryDataOutput*[]: *LogicalMemoryDataOutput*[] ;

  Maps the data output port of the physical memory to the data output port of the logical memory.

- *PhysicalMemoryAddress*[]*: LogicalMemoryAddress*[] ;

  Maps the address port of the physical memory to the address port of the logical memory.

- *PhysicalMemoryWriteAddress*[]*: LogicalMemoryWriteAddress*[] ;

  Maps the write address port of the physical memory to the write address port of the logical memory.

- *PhysicalMemoryReadAddress*[]*: LogicalMemoryReadAddress*[] ;

  Maps the read address port of the physical memory to the read address port of the logical memory.

- *PhysicalMemoryGroupWriteEnable*[]: *LogicalMemoryGroupWriteEnable*[] ;

  Maps the group write enable port of the physical memory to the group write enable port of the logical memory.

Examples

The following example cluster template demonstrates the mapping of logical memory logical ports to cluster interface logical ports. Also shown is the use of TestPortSelect to multiplex the logical ports W1/W2, and R1/R2 of logical memory LM_2R2W_inst2 to the cluster interface logical ports A and B.

```
MemoryClusterTemplate(CLUSTER_MULTIPORT) { // {{{
 MemoryBistInterface(I1) { // {{{
 // [start] : Interface port functions {{{
 // Observe some ports have multiple LogicalPort labels
 Port(clk) {Function: Clock;}
 Port(confdata[1:0]) {Function: ConfigurationData; Direction : Input;}
 Port(I1_A1[5:0]) {Function: Address; Direction: Input; LogicalPort: A,E;}
 Port(I1_A2[5:0]) {Function: Address; Direction: Input; LogicalPort: B;}
 Port(I1_A3[5:0]) {Function: Address; Direction: Input; LogicalPort: C;}
 Port(I1_A4[5:0]) {Function: Address; Direction: Input; LogicalPort: D;}
 Port(I1_DI1[15:0]) {Function: Data; Direction: Input; LogicalPort: A,E;}
 Port(I1_DO1[15:0]) {Function: Data; Direction: Output;LogicalPort: B;}
 Port(I1_DI2[15:0]) {Function: Data; Direction: Input; LogicalPort: C;}
 Port(I1_DO2[15:0]) {Function: Data; Direction: Output;LogicalPort: D;}
 Port(I1_WE1) {Function: WriteEnable; Direction: Input; LogicalPort: A,E;}
 Port(I1_RE1) {Function: ReadEnable; Direction: Input; LogicalPort: B;}
 Port(I1_WE2) {Function: WriteEnable; Direction: Input; LogicalPort: C;}
 Port(I1_RE2) {Function: ReadEnable; Direction: Input; LogicalPort: D;}
 Port(I1_SEL[2:0]) {Function: MemoryGroupAddress; Direction: Input;}
 Port(I1_SEL[3:3]) {Function: TestPortSelect; Direction: Input;}
 Port(nrst) {Function: InterfaceReset; Direction: Input; Polarity : ActiveLow;}
 Port(MbistOn1) {Function: BistOn; Direction: Input;}
 // [end] : Interface port functions }}}
 MemoryGroupAddressDecoding(GroupAddress) { // {{{
  code(3'b000) : LM_2R2W_inst1;
  code(3'b001) : LM_2R2W_inst2;
  code(3'b011) : LM_2R1W_inst3;
 } // }}}
```

```
    // The following memory does not multiplex physical ports to the
    // Shared Bus interface. For example, it has 4 logical ports that
    // are connected to four cluster logical ports A, B, C, and D.
  LogicalMemoryToInterfaceMapping(LM_2R2W_inst1) { // {{{
   MemoryTemplate : LM_2R2W;
   MemoryInstanceName : LM_2R2W_inst1;
   PipelineDepth : 9;
   PinMappings {
    TestPortSelect: 1'b0;
    LogicalMemoryLogicalPort(W1) : InterfaceLogicalPort(A);
    LogicalMemoryDataInput[7:0] : InterfaceDataInput[7:0];
    LogicalMemoryAddress[3:0] : InterfaceAddress[3:0];
   }
   PinMappings {
    TestPortSelect: 1'b0;
    LogicalMemoryLogicalPort(R1) : InterfaceLogicalPort(B);
    LogicalMemoryDataOutput[7:0] : InterfaceDataOutput[7:0];
    LogicalMemoryAddress[3:0] : InterfaceAddress[3:0];
   }
   PinMappings {
    TestPortSelect: 1'b1;
    LogicalMemoryLogicalPort(W2) : InterfaceLogicalPort(C);
    LogicalMemoryDataInput[7:0] : InterfaceDataInput[7:0];
    LogicalMemoryAddress[3:0] : InterfaceAddress[3:0];
   }
   PinMappings {
    TestPortSelect: 1'b1;
    LogicalMemoryLogicalPort(R2) : InterfaceLogicalPort(D);
    LogicalMemoryDataOutput[7:0] : InterfaceDataOutput[7:0];
    LogicalMemoryAddress[3:0] : InterfaceAddress[3:0];
   }
  } // }}}
```

```
   // The following memory uses internal multiplexing of its logical ports
   // to the Shared Bus cluster interface logical ports A and B, each of
   // which are used twice.
 LogicalMemoryToInterfaceMapping(LM_2R2W_inst2) { // {{{
  MemoryTemplate : LM_2R2W;
  MemoryInstanceName : LM_2R2W_inst2;
  PipelineDepth : 9;
  PinMappings {
   TestPortSelect: 1'b0;
   LogicalMemoryLogicalPort(W1) : InterfaceLogicalPort(A);
   LogicalMemoryDataInput[7:0] : InterfaceDataInput[7:0];
   LogicalMemoryAddress[3:0] : InterfaceAddress[3:0];
  }
  PinMappings {
   TestPortSelect: 1'b0;
   LogicalMemoryLogicalPort(R1) : InterfaceLogicalPort(B);
   LogicalMemoryDataOutput[7:0] : InterfaceDataOutput[7:0];
   LogicalMemoryAddress[3:0] : InterfaceAddress[3:0];
  }
  PinMappings {
   TestPortSelect: 1'b1;
   LogicalMemoryLogicalPort(W2) : InterfaceLogicalPort(A);
   LogicalMemoryDataInput[7:0] : InterfaceDataInput[7:0];
   LogicalMemoryAddress[3:0] : InterfaceAddress[3:0];
  }
  PinMappings {
   TestPortSelect: 1'b1;
   LogicalMemoryLogicalPort(R2) : InterfaceLogicalPort(B);
   LogicalMemoryDataOutput[7:0] : InterfaceDataOutput[7:0];
   LogicalMemoryAddress[3:0] : InterfaceAddress[3:0];
  }
 } // }}}
```

```
LogicalMemoryToInterfaceMapping(LM_2R1W_inst3) { // {{{
 MemoryTemplate : LM_2R1W;
 MemoryInstanceName : LM_2R1W_inst3;
 PipelineDepth : 9;
 PinMappings {
  // TestPortSelect is irrelevant for the Write port of
  // this logical memory
  TestPortSelect: 1'bx;
  LogicalMemoryLogicalPort(W1) : InterfaceLogicalPort(A);
  LogicalMemoryDataInput[7:0] : InterfaceDataInput[7:0];
  LogicalMemoryAddress[3:0] : InterfaceAddress[3:0];
 }
 PinMappings {
  TestPortSelect: 1'b0;
  LogicalMemoryLogicalPort(R1) : InterfaceLogicalPort(B);
  LogicalMemoryDataOutput[7:0] : InterfaceDataOutput[7:0];
  LogicalMemoryAddress[3:0] : InterfaceAddress[3:0];
 }
 PinMappings {
  TestPortSelect: 1'b1;
  LogicalMemoryLogicalPort(R2) : InterfaceLogicalPort(B);
  LogicalMemoryDataOutput[7:0] : InterfaceDataOutput[7:0];
  LogicalMemoryAddress[3:0] : InterfaceAddress[3:0];
 }
 } // }}}
 } // }}}
} // }}}
```

FuseBoxInterface

Describes a fuse box interface module with all its associated interface ports and the characteristics of the fuse box it uses.

Usage

```
Core(module_name) {
  FuseBoxInterface {

    write_duration               : time ; // Default: 8us
    read_duration                : time ; // Default: 200ns
    init_duration                : time ; // Default: 1us
    read_word_size               : int ;  // Default: 1
    align_access_en_with_address : on | off | auto ;
    read_pipeline_depth          : int ; // Default: 0
    programming_method           : buffered | unbuffered ;
    number_of_fuses              : int | auto ;

    Interface {
    }
  }
}
```

Description

A configuration file format used to describe a fuse box interface module with all its associated interface ports and the characteristics of the fuse box it uses. These descriptions are automatically read in during module matching. See the set_design_sources -format tcd_fusebox command description for information about where they are looked for. See the read_core_descriptions command description to learn how to read them in explicitly. See the set_module_matching_options command description for information about the name matching process.

To see the content of a read-in Core(ModuleName)/FuseBoxInterface, use the "report_config_data "Core(ModuleName)/FuseBoxInterface -partition tcd" command. To see the supported syntax, use the "report_config_syntax Core/FuseBoxInterface" command.

The described fuse box interface module must only have type of the ports mentioned in the Interface wrapper when it is instantiated inside the bisr controller as controlled by the fuse_box_location property of the DefaultsSpecification/DftSpecification/MemoryBisr wrapper or of the DftSpecification/MemoryBisr/Controller wrapper.

When describing a module that is external to the bisr controller, all ports described in the Interface wrapper must exist on the actual design module instance. The design module instance is allowed to have additional ports not described in the library module Interface wrapper, as long as they are specified in the DftSpecification ExternalFuseBoxOptions wrapper before running process_dft_specification. Note however, when defining connections in the ConnectionOverrides wrapper, it is mandatory to specify the following properties:

- done

- read_data

- write_en

- select

- access_en

- address

- write_duration_count

It is recommended that the FuseBoxInterface module is as complete as possible.

Arguments

- write_duration : *time* ;

  A property that defines the guaranteed minimum amount of time it takes to program a fuse. When unspecified, the default value is 8 microseconds.

- read_duration : *time* ;

  A property that defines the guaranteed maximum amount of time it takes to read a fuse value. When unspecified, the default value is 200ns.

- init_duration : *time* ;

  A property that defines the maximum initialization time needed by the fuse box after it has been selected. A delay of the specified duration is inserted before the first read access to the fuse box after it has been selected. When unspecified, the default value is 1us.

- read_word_size : *int* ;

  A property that defines how many fuses are read in one read access to the fuse box. Even though the fuse box interface module supplies only one read bit at a time, the time it takes to return a fuse value when it is part of the same read word that was previously read is much faster. The read_word_size number is used by the tool to calculate the worst case time it take to unload all the fuses when generation a power_up_emulation pattern. When unspecified, the default value is 1. Specifying a number that is smaller than reality only has the effect that the power_up_emulation pattern waits longer than it needs to.

- align_access_en_with_address : on | off | auto ;

  A property that specifies if the pulse triggering a fuse box access and generated by the BISR controller is expected to be aligned with the address to be read or written or is one cycle ahead. The value "on" means that the fuse box interface can only accept a pulse that is aligned with the address. The value "off" means that the fuse box interface can also accept a pulse occurring one cycle ahead of the address enabling a slightly faster fuse box access.

  For example, a read access can take as little as two clock cycles instead of three for some fuse box interfaces. When set to auto, it defaults to on if fuse_box_location is external, and to off if fuse_box_location is internal.

- read_pipeline_depth : *int* ;

  A property that specifies how many additional pipeline stages are to be added for fuse box read cycles. The default setting is 0, for which a read access takes two cycles when accessing an internal fuse box and three cycles for an external fuse box.

- programming_method : buffered | <u>unbuffered</u> ;

  A property used to specify that it is not allowed to program individual fuses at randomly specified addresses. In some fuseboxes, individual fuse bit cannot be addressed directly, all fuse bits must be read or programmed as a group.

  When this property is set to buffered, a signal called programFB is generated by the BISR controller and connected to the fuse box interface. This signal is used to initiate the final fuse box programming. Extra steps are also added to the patterns to transfer and program the fuses during the "self_fuse_box_program" autonomous operation and in the "program" and "read" fuse box access modes.

- number_of_fuses : *int* | <u>auto</u> ;

  A property used to define the number of fuses present in the fuse box. You need to reserve the bottom part of the address space for repair but you may use the upper part for other purposes. By default, the value is computed as 2 to the power of the number of address ports specified with the Core/FuseBoxInterface/Interface/address property. The number of fuses reserved for repair is described in the MemoryBisr/Controller wrapper of the DftSpecification wrapper.

Examples

The following example defines a fuse box interface module call fuse_2k where the write duration is 10us, the read word size is 8 and the number of fuses available for repair is 2000.

```
FuseBoxInterface {

  write_duration           : 10us;
  read_word_size           : 8
  number_of_fuses          : 2000;

}
```

Related Topics

DftSpecification/MemoryBisr/Controller [Tessent Shell Reference Manual]

DefaultsSpecification/DftSpecification/MemoryBisr [Tessent Shell Reference Manual]

report_config_data [Tessent Shell Reference Manual]

report_config_syntax [Tessent Shell Reference Manual]

Interface

Lists all the port functions and their associated port name.

Syntax

```
Core(module_name) {
  FuseBoxInterface {

    Interface {
      // inputs
      bisr_en                : port_name;
      clock                  : port_name;
      select                 : port_name;
      reset                  : port_name;
      access_en              : port_name;
      write_en               : port_name;
      address                : port_name;   // n-bit
      write_buffer_transfer  : port_name;
      read_buffer_select     : port_name;
      programming_voltage    : port_name;

      write_duration_count   : port_name;   // n-bit

      logictest_en           : port_name;

      // outputs
      done                   : port_name;
      read_data              : port_name;
      read_buffer_output     : port_name;   // n-bit
    }
  }
}
```

Description

A wrapper that list all the port functions and their associated port name. All functions except for bisr_en and programming_voltage are required.

The programming_voltage port is required if your are going to have the module instantiated inside the bisr controller, as controlled by the fuse_box_location property of the DefaultsSpecification/DftSpecification/MemoryBisr wrapper or of the DftSpecification/MemoryBisr/Controller wrapper, and it does not have a local charge pump to generate its programing voltage internally.

Parameters

- bisr_en : *port_name*;

  A property that specifies one or many connections to create from the bisr_en port to any pins in the circuit. This signal is typically used as the select for multiplexers when the fuse box is shared with other circuitry, and the multiplexer is already inside the interface.

- clock : *port_name*;

  A required property that identifies the name of the "clock" input port on the fuse box interface module. The specified clock input port is driven by the clock output port of the BISR controller.

- select : *port_name*;

  A required property that identifies the name of the "select" input port on the fuse box interface module that selects the fuse box for an operation. The specified select input port is connected to the select output port of the BISR controller.

- reset : *port_name*;

  A required property that identifies the name of the "reset" input port on the fuse box interface module.

- access_en : *port_name*;

  A required property that identifies the name of the "access_en" input port on the fuse box interface module that initiates an access to the fuse box. The specified access_en input port is connected to the access_en output port of the BISR controller.

- write_en : *port_name*;

  A required property that identifies the name of the "write_en" input port on the fuse box interface module that configures the fuse box in write mode. The specified write_en input port is connected to the write_en output port of the BISR controller.

- address : *port_name*;

  A required property that identifies the name of the "address" input ports on the fuse box interface module. The bus range must be specified as part of the *port_name*. You can use the %<integer>d[msb:lsb] symbol to define a group of scalar ports.

  For example, Add%2d[12:0] would define port "Add12" to "Add00" as the address ports.

  The specified input address port is connected to the address output ports of the BISR controller.

- write_buffer_transfer : *port_name*;

  A property that identifies the name of the "write_buffer_transfer" input port on the fuse box interface module that initiates the final fuse box programming. This property is required if FuseBoxInterface/programming_method is set to buffered. Refer to the programming_method property description for additional information. The specified write_buffer_transfer input port is connected to the programFB output port of the BISR controller.

- read_buffer_select : *port_name*;

  A property that identifies the name of the "read_buffer_select" input port on the fuse box interface module. The read_buffer_select property is mandatory if the DftSpecification/ MemoryBisr/memory_repair_loading_method property is set to "from_read_buffer", and for this setting the control of read_buffer_select is handled by Tessent Shell. When

memory_repair_loading_method is set to the default setting of "serial", the control of the read_buffer_select input on the fuse box interface is accomplished by the user through assigning a TDR register inside the IjtagNetwork, then manually controlling it in the patterns specification. If read_buffer_select is specified, the port names and bus range are validated against the design module.

- programming_voltage : *port_name*;

  A property that identifies the name of the "programming_voltage" input port on the fuse box interface module.

  The programming_voltage port is required if your are going to have the module instantiated inside the bisr controller, as controlled by the fuse_box_location property of the DefaultsSpecification/DftSpecification/MemoryBisr wrapper or of the DftSpecification/MemoryBisr/Controller wrapper, and it does not have a local charge pump to generate its programing voltage internally.

- write_duration_count : *port_name*;

  A required property that identifies the name of the "write_duration_count" input bus on the fuse box interface module, which indicates the number of clock cycles required to perform a write operation. The bus range must be specified as part of the *port_name*. You can use the %<integer>d[msb:lsb] symbol to define a group of scalar ports.

  For example, WD%2d[12:0] would define port "WD12" to "WD00" as the write_duration_count ports.

  The specified input write_duration_count ports are connected to the write_duration_count output ports of the BISR controller.

- logictest_en : *port_name*;

  An optional property that identifies the name of the "logictest_en" input port on the fuse box interface module. The specified input port is connected to the Sib(sti) ltest_to_en output port, which also connects to the BISR controller TM input port. This signal is asserted high during scan test.

- done : *port_name*;

  A required property that identifies the name of the "done" output port on the fuse box interface module. This port indicates when the fuse box access has been completed. The specified done output port is connected to the done input port of the BISR controller.

- read_data : *port_name*;

  A required property that identifies the name of the "read_data" output port on the fuse box interface module. The specified read_data output port is connected to the read_data input port of the BISR controller.

- read_buffer_output : *port_name*;

  A property that identifies the name of the read buffer output port on the fuse box interface module. You can use the %<integer>d[msb:lsb] symbol to define a group of scalar ports.

For example, RB%2d[12:0] would define port "RB12" to "RB00" as the read_buffer_output ports.

The read_buffer_output property is mandatory if the MemoryBisr/ memory_repair_loading_method parameter is set to "from_read_buffer", otherwise it is optional. If read_buffer_output is specified, the port names and bus range are validated against the design module.

Examples

The following example defines the port names on a fuse box interface in a module called generic_fuse_box.

```
Core(generic_fuse_box) {
 FuseBoxInterface {

  Interface {
    // inputs

    clock               : clock;
    select              : selectFB;
    reset               : FBreset;
    access_en           : FBAccess;
    write_en            : writeFB;
    address             : Address[8:0];
    write_buffer_transfer : programFB;
    read_buffer_select  : port_name;

    programing_voltage  : vddq;

    write_duration_count : strobeCntVal[31:0];

    logictest_en        : TM;

    // outputs
    done                : doneFB;
    read_data           : fuseValue;
  }

 }
}
```

Related Topics

DftSpecification/MemoryBisr/Controller [Tessent Shell Reference Manual]

DefaultsSpecification/DftSpecification/MemoryBisr [Tessent Shell Reference Manual]

FuseBoxInterface

This chapter documents the configuration data syntax used to encode the MemoryOperationsSpecification information. This information describes custom memory test algorithms and memory operation sets.

The configuration data syntax is composed of nested wrappers and properties that fully describe the MemoryOperationsSpecification Algorithm and OperationSet information. These specifications can be automatically created using the create_dft_specification and create_patterns_specification commands, and they can be processed using the process_dft_specification and process_patterns_specification commands. The specifications can be edited and introspected using the commands listed in Table B-1.

> **Note**
> The legacy LogicVision Algorithm and OperationSet wrapper formats are supported natively and are automatically translated into the format described in this Appendix when read.

Table B-1. Configuration Data Editing and Introspection Commands

Command	Description
add_config_element	Adds a configuration element in the configuration data.
add_config_message	Adds error, warning, or information messages to configuration elements.
add_config_tab	Adds one configuration tree tab to the Config Data Browser window.
delete_config_element	Deletes one or more configuration elements. Only elements that can be added can be deleted.
delete_config_messages	Deletes error, warning or info message that were added to configuration elements using the add_config_message command.
delete_config_tabs	Deletes one or many configuration tree tabs from the Config Data Browser window that was previously added using the add_config_tab or display_specification command.
get_config_elements	Returns a collection of configuration elements or a count of configuration elements when the -count option is used.
get_config_messages	Returns a list of message strings attached to a configuration element.

Table B-1. Configuration Data Editing and Introspection Commands (cont.)

Command	Description
get_config_value	Returns a value associated with a configuration element based on the specified option.
move_config_element	Moves a configuration element from one location to another.
read_config_data	Reads the content of configuration data files or a string into the Tessent Shell environment.
report_config_data	Reports the content of configuration wrappers in the transcript as it displays inside the file when the configuration data is written to a file using the write_config_data command.
report_config_messages	Reports messages associated to a configuration element. When the -hierarchical option is used, it also reports the messages associated to elements below the specified wrapper.
report_config_syntax	Reports the legal configuration syntax for a specified configuration object.
set_config_value	Sets the value of an element in the configuration data.
write_config_data	Writes the configuration data presently in memory into a file. When using the -wrappers option, the data written to the file can be limited to some specific wrappers.

This chapter uses the syntax conventions in Table B-2 when documenting wrappers and properties used in configuration data.

Table B-2. Syntax Conventions for Configuration Files

Convention	Example	Usage
Italic	***scan_in :*** *port_pin_name*;	An italic font indicates a user-supplied value.
Underline	wgl_type : generic	lsi;	An underlined item indicates the default value.
|	logic_level : both	high	low;	The vertical bar separates a list of values from which you must choose one. Do not include the bar in the configuration file.
...	port_naming : *port_naming*, ...;	Ellipses indicate a repeatable value. The comment "// repeatable" also indicates a repeatable value.
//	// default: ijtag_so	The double slash indicates the text immediately following is a comment and tells the tool to ignore the text.

This appendix covers the following topics:

MemoryOperationsSpecification

This section describes the syntax available to describe custom memory test algorithms and memory access waveforms. Memory test algorithms are described in the Algorithm wrapper and memory access operations are described in the OperationSet wrapper.

Syntax

```
MemoryOperationsSpecification {
   Algorithm(algorithm_name) {
   }
   OperationSet(operation_set_name) {
   }
}
```

Description

A wrapper used to hold all user-defined Algorithm and OperationSet wrappers. The content of multiple files containing each a MemoryOperationsSpecification wrapper with different Algorithm and OperationSet wrappers can be read into the tool simultaneously using the read_config_data command. The contents of multiple MemoryOperationsSpecification wrappers are merged automatically. You can use the write_config_data command to write them back out into a single file or separated into multiple files.

Parameters

No parameters are required for this wrapper.

Related Topics

Algorithm

read_config_data [Tessent Shell Reference Manual]

OperationSet

write_config_data [Tessent Shell Reference Manual]

Algorithm

This section discusses the syntax for the Algorithm wrapper used in the memory TCD for Tessent MemoryBIST. The Algorithm wrapper is used to describe the memory test algorithm that can be hard-coded into the controller or scanned into the controller for execution.

Syntax

```
MemoryOperationsSpecification {
  Algorithm(algorithm_name) {
    TestRegisterSetup {
      AddressGenerator {
      }
      DataGenerator {
      }
      allow_multi_size_memory_parallel_test : on | off;
      load_counter_a_end_count      : int | min_bank | max_bank |
                                      min_column | max_column |
                                      min_row | max_row |
                                      num_address_bits |
                                      num_bank_address_bits |
                                      num_column_address_bits |
                                      num_row_address_bits |
                                      // default: 0
      load_delay_counter_end_count : int | min_bank | max_bank |
                                      min_column | max_column |
                                      min_row | max_row |
                                      num_address_bits |
                                      num_bank_address_bits;
                                      // default: 0
      data_polarity_enable          : on | off | auto;
      operation_set_select          : operation_set_name;
      target_memory                 : all_compatible | row_only;
      treat_bank_as_row_msb         : on | off;
    }
    MicroProgram {
    }
  }
}
```

Description

This section discusses the syntax for the Algorithm wrapper used in the memory TCD for Tessent MemoryBIST. All algorithms defined in the memory TCD file are hard-coded into the controller.

You can define one or more algorithms in the Tessent MemoryBIST Algorithm file. There are multiple ways in which you can specify the algorithms you want to be available in a controller:

- You can refer to algorithms by specifying the name in the property of the Algorithm property of the Core/Memory TCD configuration file.

- You can set a global default by defining the algorithm and extra_algorithm properties in the DefaultsSpecification/DftSpecification/MemoryBist/ControllerOptions wrapper.

The global default specifies controller step algorithm and added algorithms that you want built into the controller if you do not specify them as indicated in the next list item.

- You can edit the algorithm property and the extra_algorithm property of DftSpecification/MemoryBist/Controller/AdvancedOptions wrapper to specify the default controller step algorithm and added algorithms that you want built into the controller that can be selected at run time.

The algorithm wrapper outline provided in Figure B-1 shows the required sub-wrappers that need to be present, in any included algorithm, for the process_dft_specification command to complete without error. Mandatory wrappers need to be present even if there are no properties assigned within them.

Figure B-1. Algorithm Wrapper Requirements

```
MemoryOperationsSpecification {
  Algorithm(algorithm_name) {
    TestRegisterSetup {                          // Mandatory
      AddressGenerator {                         // Mandatory
        AddressRegisterA | AddressRegisterB {    // Mandatory; can be empty
        }
      }
      DataGenerator {                            // Mandatory; can be empty
      }
    }
    MicroProgram {                               // Mandatory
      Instruction {                              // Mandatory
        NextConditions {                         // Mandatory; can be empty
          RepeatLoopA | RepeatLoopB {            // Optional
            Repeat1 | Repeat2 | Repeat3 {        // Optional
            }
          }
        }
      }
      AdvancedOptions {                          // Optional
      }
      DataCommands {                             // Optional
      }
      AddressCommands {                          // Optional
      }
      CounterCommands {                          // Optional
      }
    }
  }
}
```

Parameters

- TestRegisterSetup/allow_multi_size_memory_parallel_test : <u>on</u> | off

This property indicates whether the current algorithm is compatible with parallel testing of different size memories. The compatibility is dependent on the algorithm coding.

The value of on specifies the algorithm can be applied to any BIST step without restrictions on the memory sizes. This is the default. The value of off specifies the algorithm can only be

applied to a BIST step that is testing memories with the same number of rows, columns and banks.

- TestRegisterSetup/load_counter_a_end_count : *int* | min_bank | max_bank | min_column | max_column | min_row | max_row | num_address_bits | num_bank_address_bits | num_column_address_bits | num_row_address_bits | num_address_bits_minus_one | num_data_bits_minus_one;

  The optional property enables you to specify the end count value for the general purpose module CounterA. This end count value is the target count value for Counter A. When CounterA is instructed to count using the Microprogram/Instruction/CounterCommands/ counter_a property, the CounterA increments once per operation execution. When the CounterA module reaches the specified load_counter_a_end_count, the counter is reset to zero.

  The default value of this property is 0.

    o integer — decimal number specifying the end count value. The integer specified for the property must be in the range [0:2^n-1]. n is the number of bits in CounterA .

    o min_bank — specifies the starting bank address of the controller step to which the algorithm is applied and is the lowest starting bank address of all memories tested in the controller step.

    o max_bank — specifies the ending bank address of the controller step to which the algorithm is applied and is the highest ending bank address of all memories tested in the controller step.

    o min_column — specifies the starting column address of the controller step to which the algorithm is applied and is the lowest starting column address of all memories tested in the controller step.

    o max_column — specifies the ending column address of the controller step to which the algorithm is applied and is the highest ending column address of all memories tested in the controller step.

    o min_row — specifies the starting row address of the controller step to which the algorithm is applied and is the lowest starting row address of all memories tested in the controller step.

    o max_row — specifies the ending row address of the controller step to which the algorithm is applied and is the highest ending row address of all memories tested in the controller step.

    o num_address_bits — specifies the total number of bank, column and row address bits used in the controller step to which the algorithm is applied.

    o num_bank_address_bits — specifies the total number of bank address bits used in the controller step to which the algorithm is applied.

- num_column_address_bits — specifies the total number of column address bits used in the controller step to which the algorithm is applied.

- num_row_address_bits — specifies the total number of row address bits used in the controller step to which the algorithm is applied.

• TestRegisterSetup/load_delay_counter_end_count : *int* | min_bank | max_bank | min_column | max_column | min_row | max_row | num_address_bits | num_bank_address_bits | num_column_address_bits | num_row_address_bits;

The optional property enables you to specify the end count value for the general purpose module DelayCounter. This end count value is the target count value for the DelayCounter. When the DelayCounter is instructed to count using the Microprogram/Instruction/ CounterCommands/delay_counter property, the DelayCounter increments once per operation execution. When the DelayCounter module reaches the specified load_delay_counter_end_count, the counter is reset to zero.

The default value of this property is 0.

- integer — decimal number specifying the end count value. The integer must be in the range $[0:2^n-1]$. *n* is the number of bits in DelayCounter.

- min_bank — specifies the starting bank address of the controller step to which the algorithm is applied and is the lowest starting bank address of all memories tested in the controller step.

- max_bank — specifies the ending bank address of the controller step to which the algorithm is applied and is the highest ending bank address of all memories tested in the controller step.

- min_column — specifies the starting column address of the controller step to which the algorithm is applied and is the lowest starting column address of all memories tested in the controller step.

- max_column — specifies the ending column address of the controller step to which the algorithm is applied and is the highest ending column address of all memories tested in the controller step.

- min_row — specifies the starting row address of the controller step to which the algorithm is applied and is the lowest starting row address of all memories tested in the controller step.

- max_row — specifies the ending row address of the controller step to which the algorithm is applied and is the highest ending row address of all memories tested in the controller step.

- num_address_bits — specifies the total number of bank, column and row address bits used in the controller step to which the algorithm is applied.

- num_bank_address_bits — specifies the total number of bank address bits used in the controller step to which the algorithm is applied.

- o num_column_address_bits — specifies the total number of column address bits used in the controller step to which the algorithm is applied.

- o num_row_address_bits — specifies the total number of row address bits used in the controller step to which the algorithm is applied.

- TestRegisterSetup/data_polarity_enable : on | off | <u>auto</u>;

The optional property enables you to apply the physical data map equations, or apply a logical data pattern to the memory when executing custom algorithms. The value of on activates the data_polarity and performs the physical data mapping equations when performing operations on the memory. The value of off deactivates the data_polarity, which removes the equation polarity from the physical data map equations.

This property is ignored if the PhysicalDataMap wrapper of the memory library file does not contain any not or xor terms.

This property is not applicable to library algorithms. Data mapping is automatically applied as defined by these algorithms. For SMarchCHKB, SMarchCHKBci, SMarchCHKBcil and SMarchCHKBvcd, the physical data map is only applied during specific phases of the algorithm. For all other library algorithms, the physical data map is applied during all phases of the algorithm.

The value of auto is set by Tessent MemoryBIST as follows:

- o on — when an equation of the Core/Memory/PhysicalDataMap wrapper in the memory TCD file contains a not or xor term.

- o off — for the following algorithms: SMarchCHKB, SMarchCHKBci, SMarchCHKBcil, and SMarchCHKBvcd.

> **___ Note ___**
> Any crossing of bit lines or remapping of logical data bits to different physical data bits is still performed when data_polarity_enable is set to off.

- TestRegisterSetup/operation_set_select : *operation_set_name*;

The mandatory property enables you to identify an OperationSet from which the microprogram select Operations to be performed on the memory.

- TestRegisterSetup/target_memory : <u>all_compatible</u> | row_only;

This property indicates whether the current algorithm is to be used with row_only memories or all memories.

An algorithm that operates on the column address counter cannot be applied to memory BIST steps having zero column address bits. For this reason, you might have to create a second copy of your custom algorithms that only operates on the row address counter. For such algorithm, set this property to row_only. Those algorithms are not included into the memory BIST controller unless it has at least one step where it is testing exclusively row_only memories.

When this property is set to row_only, the algorithm is not hard-coded into the controller if no step exists with only row_only memories.

Algorithms making use of the Column Address counter are automatically excluded from the memory BIST step having exclusively row_only memories. If the controller exclusively tests row_only memories, all algorithms making use of the Column Address counter are discarded.

- TestRegisterSetup/treat_bank_as_row_msb : <u>on</u> | off ;

This property indicates whether the bank address counter is automatically configured as an extension of the row address counter. The Z address segment is linked to the X1 address segment. The Z address segment counts when instructed by the Microprogram/Instruction/AddressCommands/x1_address property and a carry-out from the X1 address segment is generated.

An application of this property is testing a memory having bank address bits with an algorithm that operates only on the row and column address counters.

A value of on specifies the Z address segment is linked to the X1 address segment. A value of off specifies the Z address segment is not linked to the X1 address segment.

Further, this property is set to off if the algorithm specifies the z_carry_in property, or if the algorithm links the carry-out from the Z address segment to any of the row address or column address segments. One of the properties x0_carry_in, x1_carry_in, y0_carry_in, y1_carry_in specifies z_carry_out.

AddressGenerator

The mandatory AddressGenerator wrapper groups the properties of the address generator that require initialization prior to execution of the microprogram.

Syntax

```
MemoryOperationsSpecification {
  Algorithm(algorithm_name) {
    TestRegisterSetup {
      AddressGenerator {
        load_bank_address_max   : bits | auto;
        load_bank_address_min   : bits | auto;
        load_column_address_max : bits | auto;
        load_column_address_min : bits | auto;
        load_row_address_max    : bits | auto;
        load_row_address_min    : bits | auto;
        AddressRegisterA | AddressRegisterB {
        }
      }
    }
  }
}
```

Description

The AddressGenerator wrapper groups the properties of the address generator that require initialization prior to execution of the microprogram. The wrapper is mandatory and contains two possible wrappers, AddressRegisterA and AddressRegisterB, of which one must be present.

Parameters

- load_bank_address_max : *bits* | auto;

  The optional property enables you to specify a maximum value for both the A and B address registers. This value is used by the Instruction/NextConditions wrapper when checking if z_end_count is true. It is also used by the Instruction/AddressCommands wrapper when a load_max command is in effect. This property defaults to a *bits* value equivalent to the maximum highRange of the CountRange(BankAddress) specified in the Core/Memory/AddressCounter wrapper of the memory TCD file for all memories tested in the controller step.

  These usage conditions apply:

  - This property is rarely used for hard algorithms. In general, the property is not specified and the tool selects the appropriate default value. Using this method, the algorithm can be applied to any memory.

  - This property is sometimes used in soft algorithms to limit the address range during diagnosis. The DftSpecification soft_algorithm_address_min_max property must be set to on for this usage.

o Specify the property only when the number of bank address bits specified in any Core/Memory/AddressCounter wrapper of the memory library file is greater than zero.

o The width of the binary value *bits* must be equivalent to the maximum address bit width specified in the AddressCounter wrapper of the memory library file for all memories tested in the controller step.

o The specified property cannot be greater than the maximum highRange specified for CountRange(BankAddress) in the AddressCounter wrapper of the memory library file for all memories tested in the controller step.

o The specified property is applied to both AddressRegisterA and AddressRegisterB.

The following is a sample load_bank_address_max value for AddressGenerator.

```
TestRegisterSetup {
  AddressGenerator {
    load_bank_address_max: 2'b01;
      .
      .
  } // end of AddressGenerator wrapper
} // end of TestRegisterSetup wrapper
```

- load_bank_address_min : *bits* | <u>auto</u>;

  The optional property enables you to specify a minimum value for both the A and B address registers. This value is used by the Instruction/NextConditions wrapper when checking if z_end_count is true. It is also used by the Instruction/AddressCommands wrapper when a load_min command is in effect. This property defaults to a *bits* value equivalent to the minimum lowRange of the CountRange(BankAddress) specified in the Core/Memory/AddressCounter wrapper of the memory TCD file for all memories tested in the controller step.

  These usage conditions apply:

  o This property is rarely used for hard algorithms. In general, the property is not specified and the tool selects the appropriate default value. Using this method, the algorithm can be applied to any memory.

  o This property is sometimes used in soft algorithms to limit the address range during diagnosis. The DftSpecification soft_algorithm_address_min_max property must be set to on for this usage.

  o Specify the property only when the number of bank address bits specified in any Core/Memory/AddressCounter wrapper of the memory library file is greater than zero.

  o The width of the binary value *bits* must be equivalent to the maximum address bit width specified in the AddressCounter wrapper of the memory library file for all memories tested in the controller step.

o The specified property cannot be less than the minimum lowRange specified for CountRange(BankAddress) in the AddressCounter wrapper of the memory library file for all memories tested in the controller step.

o The specified property is applied to both AddressRegisterA and AddressRegisterB.

The following is a sample load_bank_address_min value for AddressGenerator.

```
TestRegisterSetup {
  AddressGenerator {
    Load_bank_address_min: 2'b01;
.
  } // end of AddressGenerator wrapper
} // end of TestRegisterSetup wrapper
```

- load_column_address_max : *bits* | auto;

  The optional property enables you to specify a maximum value for both the A and B address registers. This value is used by the Instruction/NextConditions wrapper when checking if y0_end_count or y1_end_count is true. It is also used by the Instruction/AddressCommands wrapper when a load_max command is in effect. This property defaults to a *bits* value equivalent to the maximum highRange of the CountRange(ColumnAddress) specified in the Core/Memory/AddressCounter wrapper of the memory TCD file for all memories tested in the controller step.

  These usage conditions apply:

    o This property is rarely used for hard algorithms. In general, the property is not specified and the tool selects the appropriate default value. Using this method, the algorithm can be applied to any memory.

    o This property is sometimes used in soft algorithms to limit the address range during diagnosis. The DftSpecification soft_algorithm_address_min_max property must be set to on for this usage.

    o Specify the property only when the number of Column address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

    o The width of the binary value *bits* must be equivalent to the maximum address bit width specified in the AddressCounter wrapper of the memory library file for all memories tested in the controller step.

    o The specified property cannot be greater than the maximum highRange specified for CountRange(ColumnAddress) in the AddressCounter wrapper of the memory library file for all memories tested in the controller step.

    o The specified property is applied to both AddressRegisterA and AddressRegisterB.

  The following is a sample load_column_address_max value for AddressGenerator.

```
TestRegisterSetup {
  AddressGenerator {
    load_column_address_max: 2'b01;
    .
  } // end of AddressGenerator wrapper
} // end of TestRegisterSetup wrapper
```

- load_column_address_min : *bits* | <u>auto</u>;

  The optional property enables you to specify a minimum value for both the A and B address registers. This value is used by the Instruction/NextConditions wrapper when checking if y0_end_count or y1_end_count is true. It is also used by the Instruction/AddressCommands wrapper when a load_min command is in effect. This property defaults to a *bits* value equivalent to the minimum lowRange of the CountRange(ColumnAddress) specified in the Core/Memory/AddressCounter wrapper of the memory TCD file for all memories tested in the controller step.

  These usage conditions apply:

    o This property is rarely used for hard algorithms. In general, the property is not specified and the tool selects the appropriate default value. Using this method, the algorithm can be applied to any memory.

    o This property is sometimes used in soft algorithms to limit the address range during diagnosis. The DftSpecification soft_algorithm_address_min_max property must be set to on for this usage.

    o Specify the property only when the number of Column address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

    o The width of the binary value bits must be equivalent to the maximum address bit width specified in the AddressCounter wrapper of the memory library file for all memories tested in the controller step.

    o The specified property cannot be less than the minimum lowRange specified for CountRange(ColumnAddress) in the AddressCounter wrapper of the memory library file for all memories tested in the controller step.

    o The specified property is applied to both AddressRegisterA and AddressRegisterB.

  The following is a sample load_column_address_min value for AddressGenerator.

```
TestRegisterSetup {
  AddressGenerator {
    load_column_address_min: 2'b01;
    .
  } // end of AddressGenerator wrapper
} // end of TestRegisterSetup wrapper
```

- load_row_address_max : *bits* | <u>auto</u>;

  The optional property enables you to specify a maximum value for both the A and B address registers. This value is used by the Instruction/NextConditions wrapper when checking if

x0_end_count or x1_end_count is true. It is also used by the Instruction/AddressCommands wrapper when a load_max command is in effect. This property defaults to a *bits* value equivalent to the maximum highRange of the CountRange(RowAddress) specified in the Core/Memory/AddressCounter wrapper of the memory TCD file for all memories tested in the controller step.

These usage conditions apply:

- o This property is rarely used for hard algorithms. In general, the property is not specified and the tool selects the appropriate default value. Using this method, the algorithm can be applied to any memory.

- o This property is sometimes used in soft algorithms to limit the address range during diagnosis. The DftSpecification soft_algorithm_address_min_max property must be set to on for this usage.

- o Specify the property only when the number of Row address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

- o The width of the binary value bits must be equivalent to the maximum address bit width specified in the AddressCounter wrapper of the memory library file for all memories tested in the controller step.

- o The specified property cannot be greater than the maximum highRange specified for CountRange(RowAddress) in the AddressCounter wrapper of the memory library file for all memories tested in the controller step.

- o The specified property is applied to both AddressRegisterA and AddressRegisterB.

The following is a sample LoadRowAddressMax value for AddressGenerator.

```
TestRegisterSetup {
  AddressGenerator {
    load_row_address_max: 2'b01;
    .
  } // end of AddressGenerator wrapper
} // end of TestRegisterSetup wrapper
```

- load_row_address_min : *bits* | <u>auto</u>;

The optional property enables you to specify a minimum value for both the A and B address registers. This value is used by the Instruction/NextConditions wrapper when checking if x0_end_count or x1_end_count is true. It is also used by the Instruction/AddressCommands wrapper when a load_min command is in effect. This property defaults to a bits value equivalent to the minimum lowRange of the CountRange(RowAddress) specified in the Core/Memory/AddressCounter wrapper of the memory TCD file for all memories tested in the controller step.

These usage conditions apply:

o This property is rarely used for hard algorithms. In general, the property is not specified and the tool selects the appropriate default value. Using this method, the algorithm can be applied to any memory.

o This property is sometimes used in soft algorithms to limit the address range during diagnosis. The DftSpecification soft_algorithm_address_min_max property must be set to on for this usage.

o Specify the property only when the number of Row address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

o The width of the binary value bits must be equivalent to the maximum address bit width specified in the AddressCounter wrapper of the memory library file for all memories tested in the controller step.

o The specified property cannot be less than the minimum lowRange specified for CountRange(RowAddress) in the AddressCounter wrapper of the memory library file for all memories tested in the controller step.

o The specified property is applied to both AddressRegisterA and AddressRegisterB.

The following is a sample load_row_address_min value for AddressGenerator.

```
TestRegisterSetup {
  AddressGenerator {
    load_row_address_min: 2'b01;
      .
  } // end of AddressGenerator wrapper
} // end of TestRegisterSetup wrapper
```

Related Topics

MemoryOperationsSpecification

Algorithm

AddressRegisterA | AddressRegisterB

The AddressRegisterA and AddressRegisterB wrappers group the properties of the named address register.

Syntax

```
MemoryOperationsSpecification {
  Algorithm(algorithm_name) {
    TestRegisterSetup {
      AddressGenerator {
        AddressRegisterA | AddressRegisterB {
          load_bank_address    : binary | min_bank | max_bank;
          load_column_address : binary | min_column | max_column;
          load_row_address     : binary | min_row | max_row;
          number_x0_bits       : int; // default: 0
          number_y0_bits       : int; // default: 0
          x0_carry_in          : none | x1_carry_out |z_carry_out |
                                  y1_carry_out | y0_carry_out;
          x1_carry_in          : none | x0_carry_out | z_carry_out |
                                  y1_carry_out | y0_carry_out;
          y0_carry_in          : none | x1_carry_out | z_carry_out |
                                  y1_carry_out | x0_carry_out;
          y1_carry_in          : none | x1_carry_out | z_carry_out |
                                  y0_carry_out | x0_carry_out;
          z_carry_in           : none | x1_carry_out | x0_carry_out |
                                  y1_carry_out | y0_carry_out;
        }
      }
    }
  }
}
```

Description

The AddressRegisterA and AddressRegisterB wrappers group the properties for the two address registers available in the Memory BIST architecture for programmable controllers. It is mandatory that at least one wrapper is present in the AddressGenerator wrapper and the wrapper can be empty. Each address register is a counter generating the addresses during execution of the Memory BIST microprogram. These registers require initialization prior to execution of the microprogram.

The AddressRegisterA and AddressRegisterB wrappers consist of properties that enable you to specify the following:

- Initialization of the BankAddress

- Initialization of the RowAddress

- Initialization of the ColumnAddress

- Row address register segmentation

- Column address register segmentation

- Linking of address register segments

One, or both, of AddressRegisterA and AddressRegisterB must be specified in the AddressGenerator wrapper.

Parameters

- AddressRegisterA | AddressRegisterB/load_bank_address : *binary* | <u>min_bank</u> | max_bank;

  The optional property enables you to specify a binary value to be loaded into the BankAddress for the named A or B AddressRegister. The value loaded into the BankAddress is the initial value of the bank address prior to execution of the microprogram. You should specify the property only when the number of bank address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

  The value of min_bank specifies the starting bank address of the controller step to which the algorithm is applied and is the lowest starting bank address of all memories tested in the controller step. This is the default. The value of max_bank specifies the ending bank address of the controller step to which the algorithm is applied and is the highest ending bank address of all memories tested in the controller step.

- AddressRegisterA | AddressRegisterB/load_column_address : *binary* | <u>min_column</u> | max_column;

  The optional property enables you to specify a binary value to be loaded into the ColumnAddress for the named A or B AddressRegister. The value loaded into the ColumnAddress is the initial value of the column address prior to execution of the microprogram. You should specify the property only when the number of column address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

  The value of min_column specifies the starting column address of the controller step to which the algorithm is applied and is the lowest starting column address of all memories tested in the controller step. This is the default. The value max_column specifies the ending column address of the controller step to which the algorithm is applied and is the highest ending column address of all memories tested in the controller step.

- AddressRegisterA | AddressRegisterB/load_row_address : *binary* | <u>min_row</u> | max_row |

  The optional LoadRowAddress property enables you to specify a binary value to be loaded into the RowAddress for the named A or B AddressRegister. The value loaded into the RowAddress is the initial value of the row address prior to execution of the microprogram. You should specify the property only when the number of row address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

  The value min_row specifies the starting row address of the controller step to which the algorithm is applied and is the lowest starting row address of all memories tested in the controller step. This is the default. The value max_row specifies the ending row address of the controller step to which the algorithm is applied and is the highest ending row address of all memories tested in the controller step.

- AddressRegisterA | AddressRegisterB/number_x0_bits : *int*; // default: 0

The optional property enables you to specify the number of bits in the X0 segment of the RowAddress. The RowAddress counter may be separated into a X1 address counter and a X0 address counter.

int is an integer number specifying the number of bits in the X0 address counter. It defaults to 0. The number of bits in the X1 address counter is the number of row address bits less the specified number_x0_bits ;

These usage conditions apply:

   o The number of row address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD must be greater than zero.

   o The minimum value for *int* is zero.

   o The maximum value for *int* is the smallest of:

      - The number of row address bits minus one.

      - *n*, where *n* is the number_x0_bits, with a CountRange of $[0:(2^n-1)]$. That is the row address CountRange for the X0 address segment must be a full binary count from all zeros to all ones.

   o number_x0_bits must be set to 1 when ALL of the following conditions apply:

      - max_x0_segment_bits is set to 1.

      - The selected operation set has an operation with Cycle/AdvancedSignals/ row_address_count_enable set to on.

      - No x1_address is set to Hold in any Microprogram/Instruction/ AddressCommands/ wrapper.

- AddressRegisterA | AddressRegisterB/number_y0_bits : *int*; // default: 0

The optional property enables you to specify the number of bits in the Y0 segment of the ColumnAddress. The ColumnAddress counter may be separated into a Y1 address counter and a Y0 address counter.

int is a number of bits in the Y0 address counter. It defaults to 0. The number of bits in the Y1 address counter is the number of ColumnAddress bits less the specified number_y0_bits.

These usage conditions apply:

   o The number of column address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD must be greater than zero.

   o The minimum value for *int* is zero.

   o The maximum value for *int* is the smallest of:

      - The number of column address bits minus one.

- *n*, where *n* is the number_y0_bits with a CountRange of $[0:(2^n-1)]$. That is the column address CountRange for the Y0 address segment must be a full binary count from all zeros to all ones.

  o number_y0_bits must be set to 1 when ALL of the following conditions apply:

    - max_y0_segment_bits is set to 1.

    - The selected operation set has an operation with column_address_count_enable set to On.

    - No y1_address is set to Hold in any Microprogram/Instruction/ AddressCommands wrapper.

- AddressRegisterA | AddressRegisterB/x0_carry_in : <u>none</u> | x1_carry_out | z_carry_out | y1_carry_out | y0_carry_out;

  The optional property is one of several CarryIn properties used to configure the segments of the address counter. You have to specify this property when either the number of row address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero, or when number_y0_bits is greater than zero.

  Valid values are as follows:

    o none — specifies that there is no carry in required for this counter segment. This is the default. This segment counts when instructed to count by the x0_address property of the Microprogram/Instruction/AddressCommands wrapper.

    o z_carry_out — specifies that there is a carry out from the Z address segment is required for this segment. This counter segment counts when instructed to count by the x0_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the Z address segment is generated. z_carry_out can only be specified when the number of bank address bits specified in any Core/ Memory/AddressCounter wrapper of the memory TCD is greater than zero.

    o x1_carry_out — specifies that there is a carry out from the X1 address segment is required for this segment. This counter segment counts when instructed to count by the x0_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the X1 address segment is generated. x1_carry_out can only be specified when the number of row address bits specified in any Core/ Memory/AddressCounter wrapper of the memory TCD is greater than zero.

    o y1_carry_out — specifies that there is a carry out from the Y1 address segment is required for this segment. This counter segment counts when instructed to count by the x0_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the Y1 address segment is generated. y1_carry_out can only be specified when the number of columns address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD is greater than zero.

    o y0_carry_out — specifies that there is a carry out from the Y0 address segment is required for this segment. This counter segment counts when instructed to count by

the x0_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the Y0 address segment is generated. y0_carry_out can only be specified when the number of column address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD is greater than zero and number_y0_bits is greater and zero.

- AddressRegisterA | AddressRegisterB/y0_carry_in : <u>none</u> | x1_carry_out | z_carry_out | y1_carry_out | x0_carry_out;

The optional y0_carry_in property is one of several CarryIn properties used to configure the segments of the address counter. You have to specify y0_carry_in when either the number of column address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero, or number_y0_bits is greater than zero.

Valid values are as follows:

- o none — specifies that there is no carry in required for this counter segment. This segment counts when instructed to count by the y0_address property of the Microprogram/Instruction/AddressCommands.

- o z_carry_out — specifies that there is a carry out from the Z address segment is required for this segment. This counter segment counts when instructed to count by the y0_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the Z address segment is generated. z_carry_out can only be specified when the number of bank address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero.

- o x1_carry_out — specifies that there is a carry out from the X1 address segment is required for this segment. This counter segment counts when instructed to count by the y0_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the X1 address segment is generated. x1_carry_out can only be specified when the number of row address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero.

- o x0_carry_out — specifies that there is a carry out from the X0 address segment is required for this segment. This counter segment counts when instructed to count by the y0_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the X0 address segment is generated. x0_carry_out may only be specified when wither the number of row address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero, or number_x0_bits is greater than zero.

- o y1_carry_out — specifies that there is a carry out from the Y1 address segment is required for this segment. This counter segment counts when instructed to count by the y0_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the Y1 address segment is generated. y1_carry_out can only be specified when the number of column address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero.

The following is a sample linking of address segments for AddressRegisterA. In this example, assume that there are eight row (X) address bits, six column (Y) address bits, and 2 bank (Z) address bits specified in the memory templates. The resulting linked address counter is Y1<-Y0<-Z<-X1<-X0 where Y1 is the most significant address segment and X0 is the least significant address segment.

```
TestRegisterSetup {
  AddressGenerator {
    AddressRegisterA {
      number_x0_bits: 7;
      number_y0_bits: 4;
      z_carry_in: x1_carry_out;
      x1_carry_in: x0_carry_out;
      x0_carry_in: None;
      y1_carry_in: y0_carry_out;
      y0_carry_in: z_carry_out;
  } // end of AddressRegisterA wrapper
} // end of AddressGenerator wrapper
```

- AddressRegisterA | AddressRegisterB/y1_carry_in : <u>none</u> | x1_carry_out | z_carry_out | y0_carry_out | x0_carry_out;

  The optional y1_carry_in property is one of several CarryIn properties used to configure the segments of the address counter. Specify y1_carry_in when the number of column address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero.

  Valid values are as follows:

  • none — specifies that there is no carry in required for this counter segment. This segment counts when instructed to count by the y1_address property of the Microprogram/ Instruction/AddressCommands wrapper. y1_carry_in must be a value other than none (none is not allowed) when ALL of the following conditions apply:

    o max_y0_segment_bits set to 1.

    o The selected operation set has an operation with column_address_count_enable set to On.

    o y1_address set to a value other than Hold in all Microprogram/Instruction/ AddressCommands wrappers (Hold is never used).

  • z_carry_out — specifies that there is a carry out from the Z address segment is required for this segment. This counter segment counts when instructed to count by the y1_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the Z address segment is generated.

  • x1_carry_out — specifies that there is a carry out from the X1 address segment is required for this segment. This counter segment counts when instructed to count by the y1_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the X1 address segment is generated. x1_carry_out can only be specified when the number of row address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero.

• x0_carry_out — specifies that there is a carry out from the X0 address segment is required for this segment. This counter segment counts when instructed to count by the y1_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the X0 address segment is generated. x0_carry_out can only be specified when the number of row address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero, and number_x_bits is greater than zero.

• y0_carry_out — specifies that there is a carry out from the Y0 address segment is required for this segment. This counter segment counts when instructed to count by the y1_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the Y0 address segment is generated. y0_carry_out can only be specified when the number of column address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero, and number_y0_bits is greater than zero.

The following is a sample linking of address segments for AddressRegisterA. In this example, assume that there are eight row (X) address bits, six column (Y) address bits, and 2 bank (Z) address bits specified in the memory templates. The resulting linked address counter is Y1<-Y0<-Z<-X1<-X0 where Y1 is the most significant address segment and X0 is the least significant address segment.

```
TestRegisterSetup {
  AddressGenerator {
    AddressRegisterA {
      number_x0_bits: 7;
      number_y0_bits: 4;
      z_carry_in: x1_carry_out;
      x1_carry_in: x0_carry_out;
      x0_carry_in: None;
      y1_carry_in: y0_carry_out;
      y0_carry_in: z_carry_out;
    } // end of AddressRegisterA wrapper
  } // end of AddressGenerator wrapper
} // end of TestRegisterSetup wrapper
```

• AddressRegisterA | AddressRegisterB/z_carry_in : <u>none</u> | x1_carry_out | x0_carry_out | y1_carry_out | y0_carry_out;

The optional z_carry_in property is one of several CarryIn properties used to configure the segments of the address counter. You have to specify z_carry_in when the number of bank address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero.

Valid values are as follows:

o none — specifies that there is no carry in required for this counter segment. This segment counts when instructed to count by the z_address property of the Microprogram/Instruction/AddressCommands wrapper.

o x1_carry_out — specifies that there is a carry out from the X1 address segment is required for this counter segment. This segment counts when instructed to count by the z_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the X1 address segment is generated. x1_carry_out

can only be specified when the number of row address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero.

- o x0_carry_out — specifies that there is a carry out from the X0 address segment is required for this counter segment. This segment counts when instructed to count by the z_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the X0 address segment is generated. x0_carry_out can only be specified when: The number of row address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero, and the number_x0_bits is greater than zero.

- o y1_carry_out — specifies that there is a carry out from the Y1 address segment is required for this counter segment. This segment counts when instructed to count by the z_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the Y1 address segment is generated. Y1CarryOut can only be specified when the number of column address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero.

- o y0_carry_out — specifies that there is a carry out from the Y0 address segment is required for this counter segment. This segment counts when instructed to count by the z_address property of the Microprogram/Instruction/AddressCommands wrapper, and a carry out from the Y0 address segment is generated. y0_carry_out may only be specified when wither the number of column address bits specified in any AddressCounter wrapper of the memory TCD is greater than zero, or the number_y0_bits is greater than zero.

The following is a sample linking of address segments for AddressRegisterA. In this example, assume that there are eight row (X) address bits, six column (Y) address bits, and 2 bank (Z) address bits specified in the memory templates. The resulting linked address counter is Y1<-Y0<-Z<-X1<-X0 where Y1 is the most significant address segment and X0 is the least significant address segment.

```
TestRegisterSetup {
  AddressGenerator {
    AddressRegisterA {
      number_x0_bits: 7;
      number_y0_bits: 4;
      z_carry_in: x1_carry_out;
      x1_carry_in: x0_carry_out;
      x0_carry_in: None;
      y1_carry_in: y0_carry_out;
      y0_carry_in: z_carry_out;
    } // end of AddressRegisterA wrapper
  } // end of AddressGenerator wrapper
} // end of TestRegisterSetup wrapper
```

DataGenerator

The DataGenerator located wrapper inside the TestRegisterSetup wrapper groups the properties of the data generator that require initialization prior to execution of the microprogram.

Syntax

```
MemoryOperationsSpecification {
  Algorithm(algorithm_name) {
    TestRegisterSetup {
      DataGenerator {
        load_write_data              : pattern;
        load_expect_data             : pattern;
        invert_data_with_row_bit     : none | int;
        invert_data_with_column_bit  : none | int;
      }
    }
  }
}
```

Description

The DataGenerator located wrapper inside the TestRegisterSetup wrapper groups the properties of the data generator that require initialization prior to execution of the microprogram. Using this wrapper you can define the patterns that are loaded into the WriteData register or the ExpectData register.

Parameters

- load_write_data : *pattern*;

  The optional load_write_data property enables you to specify a binary value to be loaded into the WriteData register. The value loaded into the WriteData register is the initial value of the write data prior to execution of the microprogram. By default, a binary pattern with each bit set to 0 is selected. Settings for *pattern* are:

binary	A specified binary bit pattern to be written. The default value is 0s — for example 0b00..00.
all_one	A bit pattern of all 1s is loaded for writing — for example 0b11..11.
all_zero	A bit pattern of all 0s is loaded for writing — for example 0b00..00.
alternating_zero	A bit pattern of 0b10..10 is loaded for writing.
alternating_one	A bit pattern of 0b01..01 is loaded for writing.

- load_expect_data : *pattern*;

  The optional load_expect_data property enables you to specify a binary value to be loaded into the ExpectData register. The value loaded into the ExpectData register is the initial

value of the expect data prior to execution of the microprogram.By default, a binary pattern with each bit set to 0 is selected. Settings for *pattern* are:

<u>binary</u>	A specified binary bit pattern to be compared. The default value is 0s — for example 0b00..00.
all_one	A bit pattern of all 1s is loaded for comparing — for example 0b11..11.
all_zero	A bit pattern of all 0s is loaded for comparing — for example 0b00..00.
alternating_zero	A bit pattern of 0b10..10 is loaded for comparing.
alternating_one	A bit pattern of 0b01..01 is loaded for comparing.

- invert_data_with_row_bit : <u>none</u> | *int*;

  The optional invert_data_with_row_bit property enables you to specify a row address bit that inverts the applied write and expect data registers. The applied write and expect data registers are inverted when the specified row address bit is a 1 and is not inverted when the row address bit is a 0.

  The value of none specifies that no row bit is selected to invert the write and expect data registers. This is the default.

  The value *int* is of the form r[<index>]. The specified r[<index>] must select a valid row address bit where index is in the range from zero to the number of row address bits minus one. The number of row address bits is specified in the AddressCounter wrapper of the memory TCD file.

  The following is a sample invert_data_with_row_bit value that selects row address bit 0 to invert the write and expect data registers.

```
TestRegisterSetup {
  DataGenerator {
    invert_data_with_row_bit: r[0];
    .
  } // end of DataGenerator wrapper
} // end of TestRegisterSetup wrapper
```

- invert_data_with_column_bit : <u>none</u> | *int*;

  The optional invert_data_with_column_bit property enables you to specify a column address bit that inverts the applied write and expect data registers. The applied write and expect data registers are inverted when the specified column address bit is a 1 and is not inverted when the column address bit is a 0.

  The value none specifies that no column bit is selected to invert the write and expect data registers. This is the default.

  The value *int* is of the form c[<index>]. The specified c[<index>] must select a valid column address bit where index is in the range from zero to the number of column address bits minus one. The number of column address bits is specified in the memory TCD file.

The following is a sample invert_data_with_column_bit value that selects column address bit 0 to invert the write and expect data registers.

```
TestRegisterSetup {
  DataGenerator {
    invert_data_with_column_bit: c[0];
      .
  } // end of DataGenerator wrapper
} // end of TestRegisterSetup wrapper
```

Related Topics

MemoryOperationsSpecification

Algorithm

MicroProgram

The mandatory MicroProgram wrapper enables you to describe the instructions that control the execution of the memory test algorithm.

Syntax

```
MemoryOperationsSpecification {
  Algorithm(algorithm_name) {
     Microprogram {
      Instruction(instruction_name) { // repeatable
        operation_select      : operation_name ; // default: NoOperation
        branch_to_instruction : instruction_name ;
        AdvancedOptions {
        }
        DataCommands {
        }
        AddressCommands {
        }
        CounterCommands {
        {
        NextConditions {
        }
      }
    }
  }
}
```

Description

The MicroProgram wrapper consists of a set of instructions specifying commands to the Pointer Control module, Address Generator module, Data Generator module, and Sequencer module. The MicroProgram wrapper must contain at least one Instruction wrapper.

Parameters

- Instruction(*instruction_name*)

  The Instruction is a mandatory repeatable wrapper containing properties that make up the operational fields of the instruction word. Each Instruction wrapper represents one instruction word. Its parameter *instruction_name* is a string uniquely identifying the Instruction.

- Instruction(*instruction_name*)/operation_select : *operation_name* ;

  The operation_select property enables you to identify an Operation from the selected operation set, which specifies the memory operation to be run.

  Valid values are as follows:

  - NoOperation — specifies that no operation has been selected. This is the default.

  - AutoRefresh — selects the AutoRefresh operation to be performed.

    o   *operation_name* — is a string that must match one of the Operation names in the selected OperationSet, which in turn is specified in the Core/Memory wrapper of the memory TCD file.

This example specifies that the Operation "Read" is selected.

```
MicroProgram {
   Instruction (Instruction_Two) {
      operation_select: read;
         .
   } //end of Instruction wrapper
} //end of MicroProgram wrapper
```

- Instruction(*instruction_name*)/branch_to_instruction : *instruction_name* ;

This property enables you to specify the instruction that the pointer control selects as the next instruction to run if the requested conditions are not all true (except for the Instruction/NextConditions/RepeatLoop condition—that is all Repeat wrappers have not been run). The Repeat wrapper next conditions are described within the RepeatLoop wrapper. When all requested conditions are true, the next sequential instruction is loaded to run.

The default is the Instruction wrapper name for which this property was to be specified within. *instruction_name* must identify an Instruction wrapper name for which the branch_to_instruction property is specified for a previously specified instruction. That pointer control only supports a branch to itself or a branch backwards.

Related Topics

MemoryOperationsSpecification

Algorithm

Instruction/AdvancedOptions

The AdvancedOptions wrapper enables you to account for specific memory features when describing the memory test algorithm.

Syntax

```
MemoryOperationsSpecification {
  Algorithm(algorithm_name) {
    MicroProgram {
      Instruction(instruction_name) {
        AdvancedOptions {
          disable_memories_without_group_writeenable  : on | off;
          disable_memories_without_outputenable       : on | off;
          disable_memories_without_readenable         : on | off;
          disable_memories_without_select             : on | off;
          disable_memories_without_writeenable        : on | off;
          inhibit_refresh                             : on | off;
        }
      }
    }
  }
}
```

Description

Using the properties of this wrapper, you can write a generic algorithm that can be applied to memories even though certain control signals are unavailable. This avoids the need to code multiple algorithms based on the memory type.

Parameters

- disable_memories_without_group_writeenable : on | off;

  The disable_memories_without_group_writeenable property specifies whether the memories without the group write enable control signal should be disabled for the specified instruction. This property enables memories with and without a group write enable control signal to be tested by the same controller.

  The value on disables memories without the group write enable control signal for the specified instruction.

  The default value off does not disable memories without the group write enable control signal for the specified instruction.

  A bit corresponding to the group write enable control signal is added to the instruction if one of the following is true:

    o Some, but not all, memories under test have the group write enable control signal AND the controller is soft programmable.

    o Zero (0) or more, but not all, memories under test have the group write enable control signal and at least one hard algorithm attempts to disable the group write

enable control signal. The memories under test can be either concurrent or sequential.

- disable_memories_without_outputenable : on | off;

  The disable_memories_without_outputenable property specifies whether the memories without the output enable control signal should be disabled for the specified instruction. This property enables memories with and without an output enable control signal to be tested by the same controller.

  The value on disables memories without the output enable control signal for the specified instruction.

  The default value off does not disable memories without the output enable control signal for the specified instruction.

  The memories under test can be either concurrent or sequential. A bit corresponding to the output enable control signal is added to the instruction if one of the following is true:

  - Some, but not all, memories under test have the output enable control signal and the controller is soft programmable.

  - Zero (0) or more, but not all, memories under test have the output enable control signal and at least one hard algorithm attempts to disable the output enable control signal.

- disable_memories_without_readenable : on | off;

  The disable_memories_without_readenable property specifies whether memories with one of the following characteristics should be disabled for the specified instruction:

  - No read enable control signal is present. The MemoryTemplate wrapper does not specify any port with function ReadEnable.

  - Data output is not preserved when the read enable control signal is deasserted. This behavior is indicated with Core/Memory/DataOutHoldWithInactiveReadEnable set to Off.

This property enables memories with and without a read enable control signal or memories with different Core/Memory/DataOutHoldWithInactiveReadEnable settings to be tested by the same controller.

The value on for the specified instruction, disables memories without the read enable control signal or that have Core/Memory/DataOutHoldWithInactiveReadEnable set to Off.

The default value off for the specified instruction, does not disable memories without the read enable control signal or that have Core/Memory/DataOutHoldWithInactiveReadEnable set to Off.

The memories under test can be either concurrent or sequential. A bit corresponding to the read enable control signal is added to the instruction if one of the following is true:

o   Some, but not all, memories under test have the read enable control signal or have Core/Memory/DataOutHoldWithInactiveReadEnable set to On and the controller is soft programmable.

o   Zero (0) or more, but not all, memories under test have the read enable control signal or have Core/Memory/DataOutHoldWithInactiveReadEnable set to On and at least one hard algorithm attempts to disable the read enable control signal.

> **Note**
> Core/Memory/DataOutHoldWithInactiveReadEnable is On by default.

- disable_memories_without_select : on | off;

  The disable_memories_without_select property specifies whether memories with one of the following characteristics should be disabled for the specified instruction:

  o   No select control signal is present. The Core/Memory/Port wrapper does not specify any port with function Select.

  o   Memory content is not preserved when the select control signal is deasserted. This behavior is indicated with Core/Memory/MemoryHoldWithInactiveSelect set to off.

  This property enables memories with and without a select control signal or memories with different Core/Memory/MemoryHoldWithInactiveSelect settings to be tested by the same controller.

  The value on for the specified instruction, disables memories without the select control signal or that have Core/Memory/MemoryHoldWithInactiveSelect set to Off.

  The default value off for the specified instruction, does not disable memories without the select control signal or that have Core/Memory/MemoryHoldWithInactiveSelect set to Off.

  The memories under test can be either concurrent or sequential. A bit corresponding to the select control signal is added to the instruction if one of the following is true:

  o   Some, but not all, memories under test have the select control signal or have Core/Memory/MemoryHoldWithInactiveSelect set to On and the controller is soft programmable.

  o   Zero (0) or more, but not all, memories under test have the select control signal or have Core/Memory/MemoryHoldWithInactiveSelect set to On and at least one hard algorithm attempts to disable the select control signal.

  > **Note**
  > Core/Memory/MemoryHoldWithInactiveSelect is set to On by default.

- disable_memories_without_writeenable : on | <u>off</u>;

  The disable_memories_without_writeenable property specifies whether the memories without the write enable control signal should be disabled for the specified instruction. This property enables memories with and without a write enable control signal to be tested by the same controller.

  A value of on disables memories without the write enable control signal for the specified instruction.

  The default value off does not disable memories without the write enable control signal for the specified instruction.

  The memories under test can be either concurrent or sequential. A bit corresponding to the write enable control signal is added to the instruction if one of the following is true:

  - Some, but not all, memories under test have the write enable control signal and the controller is soft programmable.

  - Zero (0) or more, but not all, memories under test have the write enable control signal and at least one hard algorithm attempts to disable the write enable control signal.

- inhibit_refresh : on | <u>off</u>;

  The InhibitRefresh property enables you to prevent the insertion of any AutoRefresh operations by the Refresh Control module.

  The value on inhibits AutoRefresh operations from being inserted when the Refresh Control module is enabled.

  The value off enables AutoRefresh operations to be inserted when the Refresh Control module is enabled. This is the default.

  The InhibitRefresh is meaningful only if:

  - The Core/Memory/MemoryType property in the memory library file has been specified as DRAM.

  - The Controller/DramOptions/run_time_refresh_interval property is not off in the MemoryBist wrapper of the PatternsSpecification.

Related Topics

MemoryOperationsSpecification

Algorithm

Core

Memory

Instruction/DataCommands

The DataCommands wrapper enables you to manipulate the data pattern written to the memory input and expected from the memory output.

Syntax

```
MemoryOperationsSpecification {
  Algorithm(algorithm_name) {
    MicroProgram {
      Instruction(instruction_name) {
        DataCommands {
          inhibit_data_compare : on | off;
          expect_data          : zero | one | data_reg |
                                 inverse_data_reg | data_reg_rotate |
                                 inverse_data_reg_rotate |
                                 data_reg_rotate_with_invert |
                                 inverse_data_reg_rotate_with_invert |
                                 data_reg_prdg | inverse_data_reg_prdg |
                                 reset_data_reg | set_data_reg |
          write_data           : zero | one | data_reg |
                                 inverse_data_reg |
                                 data_reg_rotate |
                                 inverse_data_reg_rotate |
                                 data_reg_rotate_with_invert |
                                 inverse_data_reg_rotate_with_invert |
                                 data_reg_prdg | inverse_data_reg_prdg |
                                 reset_data_reg | set_data_reg |
          address_a_equals_b   : off | invert_expect_data |
                                 invert_write_data |
                                 invert_write_data_and_expect_data;

        }
      }
    }
  }
}
```

Description

The data pattern can be selected based on the output of the expect and write data registers as well as rotating the register content. The compare event on the read data for the selected operation can also be suppressed.

Parameters

- inhibit_data_compare : on | off;

  The inhibit_data_compare property enables you to compare normally expected data and read data, or to disable any StrobeDataOut signal during running of the selected operation.

  For the value on, any StrobeDataOut signal is disabled and the expect data and read data are not compared.

  For the default value off, expected data and read data are compared normally.

This property is ignored for an instruction execution within an Instruction/NextConditions/ RepeatLoopA or RepeatLoopB or combination thereof. This property is overridden by one of or the combination thereof the RepeatLoopA/Repeat/inhibit_data_compare and RepeatLoopB/Repeat/inhibit_data_compare values specified.

This example specifies that any StrobeDataOut signal is disabled, and the expect data and read data are not compared.

```
MicroProgram {
  Instruction (Instruction_Two) {
    DataCommands {
      inhibit_data_compare: on;
      .
    } // end of DataCommands
  } //end of Instruction wrapper
} //end of MicroProgram wrapper
```

- expect_data : <u>zero</u> | one | data_reg | inverse_data_reg | data_reg_rotate | inverse_data_reg_rotate | data_reg_rotate_with_invert | inverse_data_reg_rotate_with_invert | data_reg_prdg | inverse_data_reg_prdg | reset_data_reg | set_data_reg ;

  The expect_data property specifies the expected data to be compared with data read from the memory.

  Valid values are as follows:

  - zero — selects expect data of all zeros. This is the default.

  - one — selects expect data of all ones.

  - data_reg — selects the expect data register loaded with a value specified by the DataGenerator/load_expect_data property.

  - inverse_data_reg — selects the inverted expect data register. The initial value of the expect data register is specified by the DataGenerator/load_expect_data property.

  - data_reg_rotate — selects the expect data register. The expect data register is rotated at the end of the executing operation. The initial value of the expect data register is specified by the DataGenerator/load_expect_data property.

  - inverse_data_reg_rotate — selects the inverted expect data register. The expect data register is rotated at the end of the executing operation. The initial value of the expect data register is specified by the DataGenerator/load_expect_data property.

  - data_reg_rotate_with_invert — selects the expect data register. The expect data register with an inverted feedback is rotated at the end of the executing operation. The initial value of the expect data register is specified by the DataGenerator/ load_expect_data property.

  - inverse_data_reg_rotate_with_invert — selects the inverted expect data register. The expect data register with an inverted feedback is rotated at the end of the executing

operation. The initial value of the expect data register is specified by the DataGenerator/load_expect_data property.

- o data_reg_prdg — applies the content of the expect data register to the memory outputs and updates at the end of the current operation to generate a new pseudo-random value for a subsequent operation.

- o inverse_data_reg_prdg — applies the content of the inverted expect data register to the memory outputs and updates at the end of the current operation to generate a new pseudo-random value for a subsequent operation.

- o reset_data_reg — selects expect data of constant zeros. The value is not modified by any inversion with address bits.

- o set_data_reg — selects expect data of constant ones. The value is not modified by any inversion with address bits.

These usage conditions apply:

- o For the following commands, the content of the expect data register is preserved at the end of the current operation:

zero	one
data_reg	inverse_data_reg
reset_data_reg	set_data_reg

- o The rotate command is performed on the last tick of the selected operation. For each execution, the rotate is one bit to the left or from the least significant bit, bit 0 of the expect data register, to the most significant bit of the expect data register. The most significant bit of the expect data register is rotated to the least significant bit, bit 0, of the expect data register.

- o The rotate_with_invert command specifies that the most significant bit of the expect data register is inverted when feed back to bit 0 of the expect data register.

- o The specified command, with the exception of reset_data_reg and set_data_reg, may be modified when an Instruction/NextConditions RepeatLoop is active. For a detailed description of the expect data command modifications, refer to the expect_data_sequence property.

- o The specified command, with the exception of reset_data_reg and set_data_reg, is modified by the DataGenerator/invert_data_with_row_bit and invert_data_with_column_bit properties.

- o The values applied by the reset_data_reg and set_data_reg commands are not modified by the DataGenerator/invert_data_with_row_bit and DataGenerator/invert_data_with_column_bit properties. These commands are useful when reading constant zeros or ones from all memory locations such as in a march-type algorithm.

In contrast, the zero and one commands are useful when reading alternating zeros or ones, based on address, such as in a checkerboard-type algorithm.

o data_reg_prdg and inverse_data_reg_prdg — To use the pseudo-random data pattern generation feature:

- The data registers must have at least 8 bits.

- The maximum length of the pseudo-random sequence is determined by the data register size as follows:

Data Register Size	Sequence Length	Polynomial
8-15	$2^8 - 1$	$x^8 + x^4 + x^3 + x^2 + 1$
16-31	$2^{16} - 1$	$x^{16} + x^{12} + x^3 + x + 1$
32-63	$2^{32} - 1$	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ (Standard CRC-32 polynomial)
64+	$2^{64} - 1$	$x^{64} + x^4 + x^3 + x + 1$ (Standard CRC-64 polynomial)

o The polynomials used to generate the pseudo-random sequence are fixed and implemented as a Linear Feedback Shift Register (LFSR) of type I.

o Both the write data and expect data register must be initialized with the same seed. This seed cannot be all 0s.

o The read address sequence must exactly match the write address sequence so that the comparators do not report mis-compares.

o A memory location must be read before writing a new pseudo-random value into that location so that the comparators do not report mis-compares.

This example specifies that expect data of all ones is selected.

```
MicroProgram {
  Instruction (Instruction_Two) {
    DataCommands {
      expect_data: one;
        .
    } // end of DataCommands wrapper
  } //end of Instruction wrapper
} //end of MicroProgram wrapper
```

- write_data : <u>zero</u> | one | data_reg | inverse_data_reg | data_reg_rotate | inverse_data_reg_rotate | data_reg_rotate_with_invert |

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

inverse_data_reg_rotate_with_invert | data_reg_prdg | inverse_data_reg_prdg | reset_data_reg | set_data_reg ;

The write_data property specifies the write data to be written to the memory.

Valid values are as follows:

- o   zero — selects write data of all zeros. This is the default

- o   one — selects write data of all ones.

- o   data_reg — selects the write data register loaded with a value specified by the DataGenerator/load_write_data property.

- o   inverse_data_reg — selects the inverted write data register. The initial value of the write data register is specified by the DataGenerator/load_write_data property.

- o   data_reg_rotate — selects the write data register. The write data register is rotated at the end of the executing operation. The initial value of the write data register is specified by the DataGenerator/load_write_data property.

- o   inverse_data_reg_rotate — selects the inverted write data register. The write data register is rotated at the end of the executing operation. The initial value of the write data register is specified by the DataGenerator/load_write_data property.

- o   data_reg_rotate_with_invert — selects the write data register. The write data register with an inverted feedback is rotated at the end of the executing operation. The initial value of the write data register is specified by the DataGenerator/load_write_data property.

- o   inverse_data_reg_rotate_with_invert — selects the inverted write data register. The write data register with an inverted feedback is rotated at the end of the executing operation. The initial value of the write data register is specified by the DataGenerator/load_write_data property.

- o   data_reg_prdg — applies the content of the write data register to the memory inputs and updates at the end of the current operation to generate a new pseudo-random value for a subsequent operation.

- o   inverse_data_reg_prdg — applies the content of the inverted write data register to the memory inputs and updates at the end of the current operation to generate a new pseudo-random value for a subsequent operation.

- o   reset_data_reg — selects write data of constant zeros. The value is not modified by any inversion with address bits.

- o   set_data_reg — selects write data of constant ones. The value is not modified by any inversion with address bits.

These usage conditions apply:

o   For the following commands, the content of the write data register is preserved at the end of the current operation:

zero one

data_reg inverse_data_reg

reset_data_reg set_data_reg

o   The rotate command is performed on the last tick of the selected operation. For each execution the rotate is one bit to the left or from the least significant bit, bit 0 of the write data register, to the most significant bit of the write data register. The most significant bit of the write data register is rotated to the least significant bit, bit 0, of the write data register.

o   The rotate_with_invert command specifies that the most significant bit of the write data register is inverted when feed back to bit 0 of the write data register.

o   The specified command, with the exception of reset_data_reg and set_data_reg, may be modified when a Instruction/NextConditions/Repeat Loop is active. For a detailed description of the write data command modifications, refer to the RepeatLoop/Repeat/write_data_sequence description.

o   The specified command, with the exception of reset_data_reg and set_data_reg, is modified by the DataGenerator/invert_data_with_row_bit and DataGenerator/invert_data_with_column_bit properties.

o   The values applied by the reset_data_reg and set_data_reg commands are not modified by the DataGenerator/invert_data_with_row_bit and DataGenerator/invert_data_with_column_bit properties. These commands are useful when writing constant zeros or ones to all memory locations such as in a march-type algorithm. In contrast, the zero and one commands are useful when writing alternating zeros or ones, based on address, such as in a checkerboard-type algorithm.

o   data_reg_prdg and inverse_data_reg_prdg — To use the pseudo-random data pattern generation feature:

•   The data registers must have at least 8 bits.

•   The maximum length of the pseudo-random sequence is determined by the data register size as follows::

Data Register Size	Sequence Length	Polynomial
8-15	$2^8 - 1$	$x^8 + x^4 + x^3 + x^2 + 1$

Data Register Size	Sequence Length	Polynomial
16-31	$2^{16} - 1$	$x^{16} + x^{12} + x^3 + x + 1$
32-63	$2^{32} - 1$	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ (Standard CRC-32 polynomial)
64+	$2^{64} - 1$	$x^{64} + x^4 + x^3 + x + 1$ (Standard CRC-64 polynomial)

- o The polynomials used to generate the pseudo-random sequence are fixed and implemented as a Linear Feedback Shift Register (LFSR) of type I.

- o Both the write data and expect data register must be initialized with the same seed. This seed cannot be all 0s.

- o The read address sequence must exactly match the write address sequence so that the comparators do not report mis-compares.

- o A memory location must be read before writing a new pseudo-random value into that location so that the comparators do not report mis-compares.

This example selects the write data register loaded with a value specified by the DataGenerator/load_write_data property.

```
MicroProgram {
  Instruction (Instruction_Two) {
    DataCommands {
      write_data: data_reg;
      .
    } // end of DataCommands wrapper
  } //end of Instruction wrapper
} //end of MicroProgram wrapper
```

- • address_a_equals_b : off | invert_expect_data | invert_write_data | invert_write_data_and_expect_data;

  The address_a_equals_b property enables you to invert the write data, expect data, or both write and expect data when AddressRegisterA equals AddressRegisterB. This property is useful for algorithms with a "home" and an "away" cell such as the standard LVGalPat Algorithm. This enables you to program a single instruction that can, for example, write a background of "zeros" but write "ones" to a single cell in the memory array.

  Valid values are as follows:

  - o off — disables the inversion of the write or expect data registers when AddressRegisterA equals AddressRegisterB. This is the default.

- o invert_write_data — enables the inversion of the applied write data registers when AddressRegisterA equals AddressRegisterB.

- o invert_expect_data — enables the inversion of the applied expect data registers when AddressRegisterA equals AddressRegisterB.

- o invert_write_data_and_expect_data — enables the inversion of the applied write and expect data registers when AddressRegisterA equals AddressRegisterB.

This example specifies that the inversion of the applied write data registers is enabled when AddressRegisterA equals AddressRegisterB.

```
MicroProgram {
  Instruction (Instruction_Two) {
    DataCommands {
      address_a_equals_b: invert_write_data;
      .
    } // end of DataCommands wrapper
  } //end of Instruction wrapper
} //end of MicroProgram wrapper
```

Related Topics

MemoryOperationsSpecification

Algorithm

Core

Memory

Instruction/AddressCommands

The AddressCommands wrapper specifies the value applied to the memory address input throughout the instruction, as well as how they are updated at the end of the instruction.

Syntax

```
MemoryOperationsSpecification {
  Algorithm(algorithm_name) {
    MicroProgram {
      Instruction(instruction_name) {
        AddressCommands {
          address_select               : select_a | select_b |
                                         a_xor_b | select_a_copy_to_b |
                                         select_b_copy_to_a |
                                         select_a_rotate_b |
                                         select_b_rotate_a |
                                         select_a_rotate_left_b |
                                         select_b_rotate_left_a |
                                         select_b_rotate_right_a |
          inhibit_last_address_count : on | off;
          x1_address                 : hold | increment | decrement |
                                       load_min | load_max;
          x0_address                 : hold | increment | decrement |
                                       load_min | load_max;
          y1_address                 : hold | increment | decrement |
                                       load_min | load_max;
          y0_address                 : hold | increment | decrement |
                                       load_min | load_max;
          z_address                  : hold | increment | decrement |
                                       load_min | load_max;
        }
      }
    }
  }
}
```

Description

The AddressCommands wrapper enables you to select the value applied to the memory address input throughout the instruction. You can also specify how the column, row and bank address segments are updated at the end of the instruction. Contents can be transferred between the two address registers and their content can also be modified by rotation.

Parameters

- address_select : select_a | select_b | a_xor_b | select_a_copy_to_b | select_b_copy_to_a |select_a_rotate_b | select_b_rotate_a | select_a_rotate_left_b | select_b_rotate_left_a | select_b_rotate_right_a ;

  The address_select property enables you to select address register A or address register B and to manipulate the address registers. The content of the selected address register is applied to the memory on the address bus. This property also defines which register the

address segment commands z_address, x0_address, x1_address, y0_address, and y1_address are to be performed on.

Valid values are as follows:

- o select_a — applies AddressRegisterA to the memory address bus, and the address segment commands are performed on AddressRegisterA. This is the default.

- o select_b — applies AddressRegisterB to the memory address bus, and the address segment commands are performed on AddressRegisterB.

- o a_xor_b — applies the result of the logical exclusive OR between AddressRegisterA and AddressRegisterB to the memory address bus, and the address segment commands are performed on AddressRegisterA.

- o select_a_copy_to_b — applies AddressRegisterA to the memory address bus, the address segment commands are performed on AddressRegisterA, and the result is copied to AddressRegisterB.

- o select_b_copy_to_a — applies AddressRegisterB to the memory address bus, the address segment commands are performed on AddressRegisterB, and the result is copied to AddressRegisterA.

- o select_a_rotate_b — applies AddressRegisterA to the memory address bus, the address segment commands are performed on AddressRegisterA, and the AddressRegisterB is rotated left.

- o select_b_rotate_a — applies AddressRegisterB to the memory address bus, the address segment commands are performed on AddressRegisterB, and the AddressRegisterA is rotated left.

- o select_a_rotate_left_b — applies AddressRegisterA to the memory address bus, the address segment commands are performed on AddressRegisterA, and AddressRegisterB is rotated towards the MSB. This command is equivalent to select_a_rotate_b.

- o select_b_rotate_left_a — applies AddressRegisterB to the memory address bus, the address segment commands are performed on AddressRegisterB, and AddressRegisterA is rotated towards the MSB. This command is equivalent to select_b_rotate_a.

- o select_b_rotate_right_a — applies AddressRegisterB to the memory address bus, the address segment commands are performed on AddressRegisterB, and AddressRegisterA is rotated towards the LSB.

These usage conditions apply:

- o The address register selected is valid for running the selected operation.

- o The copy command is performed on the last tick of the selected operation.

o The rotate command is performed on the last tick of the selected operation. The address register consists of BankAddress<-RowAddress<-ColumnAddress, where the column address is the least significant. For each execution, the rotate is one bit to the left (from bit 0 to the MSB) or to the right (from the MSB to bit 0) on the selected address register and involves all of the address register bits.

If some memories use a reduced number of address bits, redundant address rotations occur that may cause simulation mismatches, depending on the test algorithm implementation. To avoid these redundant address combinations, you should bind only homogeneous memories to a given controller when any of the rotate commands are used. If the controller is testing memories of different address sizes, you can create multiple versions of your test algorithm, customizing each version to compensate for the redundant address rotations.

This example specifies that AddressRegisterB is applied to the memory address bus, and the address segment commands are performed on AddressRegisterB.

```
MicroProgram {
  Instruction (Instruction_Two) {
    AddressCommands {
      address_select : select_b;
      .
    } // end of AddressCommands wrapper
  } //end of Instruction wrapper
} //end of MicroProgram wrapper
```

- inhibit_last_address_count : on | off;

The inhibit_last_address_count property enables you to prevent the selected AddressRegister from counting on the last execution of the selected instruction. Typically, this is used to prevent the address counter from wrapping on the last instruction execution from the maximum address to the minimum address or vice versa. Thus, on the next instruction a reverse address sequence is possible without requiring an additional instruction to change the address pointer.

The on value prevents the selected AddressRegister from counting on the last execution of the selected instruction when all requested conditions, specified in the Instruction/ NextConditions wrapper, are true and the next sequential instruction is loaded for execution.

For the value *off*, any address segment command set to Increment or Decrement is performed normally. This is the default.

These usage conditions apply:

o For an instruction that specifies a Instruction/NextConditions/RepeatLoopA or RepeatLoopB wrapper and the RepeatLoop is executing a Repeat, this property is overridden by the value specified for that Repeat in the inhibit_last_address_count property within the Repeat wrapper.

o This property is ignored if the following properties are defined as load_min or load_max: z_address, x0_address, x1_address, y0_address, and y1_address

This example specifies that the selected AddressRegister is prevented from counting on the last execution of the selected instruction when all requested NextConditions are true and the next sequential instruction is loaded for execution.

```
MicroProgram {
  Instruction (Instruction_Two) {
    AddressCommands {
      inhibit_last_address: on;
      .
    } // end of AddressCommands wrapper
  } //end of Instruction wrapper
} //end of MicroProgram wrapper
```

- x1_address : <u>hold</u> | increment | decrement | load_min | load_max;

  The x1_address property enables you to specify the address counting command for the X1 segment of the row address.

  Valid values are as follows:

  - hold — specifies that the X1 segment of the selected address register holds the current value. This is the default

  - increment — specifies an incrementing count direction from minimum to maximum for the X1 segment of the selected address register.

  - decrement — specifies a decrementing count direction from maximum to minimum for the X1 segment of the selected address register.

  - load_min — loads the X1 address segment with the value defined by the AddressGenerator/load_row_address_min property.

  - load_max — loads the X1 address segment with the value defined by the AddressGenerator/load_row_address_max property.

  The following usage conditions apply:

  - Specify x1_address when the number of row address bits specified in any Core/ Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

  - The specified command may be modified when an Instruction/NextConditions/ Repeat Loop is active. For a detailed description of the address segment command modifications, refer to address_sequence.

  - The x0_address property must have the same value as the x1_address property when it is defined as load_min or load_max.

    This example specifies an incrementing count direction from minimum to maximum for the X1 segment of the selected address register.

  This example specifies an incrementing count direction from minimum to maximum for the X1 segment of the selected address register.

```
MicroProgram {
  Instruction (Instruction_Two) {
    AddressCommands {
      x1_address : increment;
      .
    } // end of AddressCommands wrapper
  } //end of Instruction wrapper
} //end of MicroProgram wrapper
```

- x0_address : <u>hold</u> | increment | decrement | load_min | load_max;

  The x0_address property enables you to specify the address counting command for the X0 segment of the row address.

  Valid values are as follows:

  - hold — specifies that the X0 segment of the selected address register holds the current value. This is the default.

  - increment — specifies an incrementing count direction from minimum to maximum for the X0 segment of the selected address register.

  - decrement — specifies a decrementing count direction from maximum to minimum for the X0 segment of the selected address register.

  - load_min — loads the X0 address segment with the value defined by the AddressGenerator/load_row_address_min property.

  - load_max — loads the X0 address segment with the value defined by the AddressGenerator/load_row_address_max property.

  The following usage conditions apply:

  - Specify x0_address when:

    - The number of row address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

    - The value of the number_x0_bits property within the AddressRegisterA | AddressRegisterB wrapper is greater than zero for the selected address register.

  - The specified command may be modified when an Instruction/NextConditions/ Repeat Loop is active Repeat Loop is active. For a detailed description of the address segment command modifications, refer to address_sequence.

  - The x1_address property must have the same value as x0_address property when it is defined as load_min or load_max.

  This example specifies an incrementing count direction from minimum to maximum for the X0 segment of the selected address register.

```
MicroProgram {
  Instruction (Instruction_Two) {
    AddressCommands {
      x0_address: increment;
      .
    } // end of AddressCommands wrapper
  } //end of Instruction wrapper
} //end of MicroProgram wrapper
```

- y1_address : <u>hold</u> | increment | decrement | load_min | load_max;

  The y1_address property enables you to specify the address counting command for the Y1 segment of the column address.

  Valid values are as follows:

  o hold — specifies that the Y1 segment of the selected address register holds the current value. This is the default.

  o increment — specifies an incrementing count direction from minimum to maximum for the Y1 segment of the selected address register.

  o decrement — specifies a decrementing count direction from maximum to minimum for the Y1 segment of the selected address register.

  o load_min — loads the Y1 address segment with the value defined by the AddressGenerator/load_row_address_min property.

  o load_max — loads the Y1 address segment with the value defined by the AddressGenerator/load_row_address_max property.

  The following usage conditions apply:

  o Specify y1_address when the number of column address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

  o The specified command may be modified when an Instruction/NextConditions/Repeat Loop is active Repeat Loop is active. For a detailed description of the address segment command modifications, refer to address_sequence.

  o The y1_address property must have the same value as y0_address property when it is defined as load_min or load_max.

  This example specifies an incrementing count direction from minimum to maximum for the Y1 segment of the selected address register.

```
MicroProgram {
  Instruction (Instruction_Two) {
    AddressCommands {
      y1_address: increment;
      .
    } // end of AddressCommands wrapper
  } // end of Instruction wrapper
} // end of MicroProgram wrapper
```

- y0_address : <u>hold</u> | increment | decrement | load_min | load_max;

The y0_address property enables you to specify the address counting command for the Y0 segment of the column address.

Valid values are as follows:

- hold — specifies that the Y0 segment of the selected address register holds the current value.

- increment — specifies an incrementing count direction from minimum to maximum for the Y0 segment of the selected address register.

- decrement — specifies a decrementing count direction from maximum to minimum for the Y0 segment of the selected address register.

- load_min — loads the Y0 address segment with the value defined by the AddressGenerator/load_row_address_min property.

- load_max — loads the Y0 address segment with the value defined by the AddressGenerator/load_row_address_max property.

The following usage conditions apply:

- The Controller/AlgorithmResourceOptions/address_segment_x0_y0_allowed is on in the MemoryBist wrapper of the DftSpecification.

- Specify y0_address when:

  - The number of column address bits specified in any Core/Memory/ AddressCounter wrapper in the memory library file is greater than zero.

  - The value of the number_y0_bits property within the MemoryOperationsSpecification/Algorithm/TestRegisterSetup/ wrapper is greater than zero for the selected address register.

- The specified command may be modified when an Instruction/NextConditions/ Repeat Loop is active Repeat Loop is active. For a detailed description of the address segment command modifications, refer to address_sequence.

- The y1_address property must have the same value as y0_address property when it is defined as load_min or load_max.

This example specifies an incrementing count direction from minimum to maximum for the Y0 segment of the selected address register.

```
MicroProgram {
  Instruction (Instruction_Two) {
    AddressCommands {
      y0_address: increment;
      .
    } // end of AddressCommands wrapper
  } //end of Instruction wrapper
} //end of MicroProgram wrapper
```

- z_address : <u>hold</u> | increment | decrement | load_min | load_max;

The z_address property enables you to specify the address counting command for the Z segment of the bank address.

Valid values are as follows:

- hold — specifies that the Z segment of the selected address register holds the current value. This is the default.

- increment — specifies an incrementing count direction from minimum to maximum for the Z segment of the selected address register.

- decrement — specifies a decrementing count direction from maximum to minimum for the Z segment of the selected address register.

- load_min — loads the Z address segment with the value defined by the AddressGenerator/load_row_address_min property.

- load_max — loads the Z address segment with the value defined by the AddressGenerator/load_row_address_max property.

The following usage conditions apply:

- Specify z_address when the number of bank address bits specified in any Core/ Memory/AddressCounter wrapper in the memory library file is greater than zero.

- The specified command may be modified when an Instruction/NextConditions/ Repeat Loop is active Repeat Loop is active. For a detailed description of the address segment command modifications, refer to address_sequence.

This example specifies an incrementing count direction from minimum to maximum for the Z segment of the selected address register.

```
MicroProgram {
  Instruction (Instruction_Two) {
    AddressCommands {
      z_address: increment;
      .
    } // end of AddressCommands wrapper
  } //end of Instruction wrapper
} //end of MicroProgram wrapper
```

Related Topics

AddressGenerator

AddressCounter

Instruction/CounterCommands

The CounterCommands wrapper enables you to specify how the CounterA and the DelayCounter are updated at the end of the instruction.

Syntax

```
MemoryOperationsSpecification {
  Algorithm(algorithm_name) {
    MicroProgram {
      Instruction(instruction_name) {
        CounterCommands {
          counter_a      : hold | increment;
          delay_counter : hold | increment;
        }
      }
    }
  }
}
```

Description

The CounterA and DelayCounter can independently preserve the counter value or request the counter to increment by 1 at the end of the instruction.

Parameters

- counter_a : hold | increment;

  The counter_a property enables you to specify the command issued to the CounterA module.

  The value hold specifies that the counter is to hold. This is the default.

  The value increment specifies that the counter is enabled to count on the last tick of the executing operation.

  This example specifies that the counter is enabled to count on the last tick of the executing operation.

  ```
  MicroProgram {
    Instruction (Instruction_Two) {
      CounterCommands {
        counter_a: increment;
        .
      } // end of CounterCommands wrapper
    } //end of Instruction wrapper
  } //end of MicroProgram wrapper
  ```

- delay_counter : hold | increment;

  The delay_counter property specifies the command issued to the DelayCounter module.

  The value hold specifies that the counter is to hold. This is the default.

The value increment specifies that the counter is enabled to count on the last tick of the executing operation.

The delay_counter property can only be used when the Controller/DramOptions/run_time_refresh_interval property is off in the MemoryBist wrapper of the PatternsSpecification.

This example specifies that the counter is enabled to count on the last tick of the executing operation.

```
MicroProgram {
  Instruction (Instruction_Two) {
    CounterCommands {
      delay_counter: increment;
      .
    } // end of CounterCommands wrapper
  } //end of Instruction wrapper
} //end of MicroProgram wrapper
```

Instruction/NextConditions

The mandatory NextConditions wrapper contains properties that are used to sequence the microprogram instructions.

Syntax

```
MemoryOperationsSpecification {
  Algorithm(algorithm_name) {
    MicroProgram {
      Instruction(instruction_name) {
        NextConditions {
          x0_end_count              : on | off;
          x1_end_count              : on | off;
          y0_end_count              : on | off;
          y1_end_count              : on | off;
          z_end_count               : on | off;
          counter_a_end_count       : on | off;
          delay_counter_end_count : on | off;
          RepeatLoopA | RepeatLoopB {
            branch_to_instruction : instruction_name ;
            Repeat1 | Repeat2 | Repeat3 {
              enable                     : on | off;
              address_sequence           : no change | inverse;
              expect_data_sequence       : no change | inverse;
              write_data_sequence        : no change | inverse;
              inhibit_last_address_count : on | off;
              inhibit_data_compare       : on | off;
            }
          }
        }
      }
    }
  }
}
```

Description

The NextConditions wrapper properties sequence the microprogram instructions as follows:

- If all requested conditions are true the next sequential instruction word is loaded.

- If all requested conditions are true, except for the requested RepeatLoop condition, the next instruction word to be loaded is specified by the branch_to_instruction property in the RepeatLoop wrappers.

- If any of the requested conditions are not true the next instruction word to be loaded is specified by the branch_to_instruction property.

The NextConditions wrapper is mandatory within an Instruction wrapper and the wrapper can be empty if no properties need to change from default settings.

Parameters

- x0_end_count : on | off;

  The x0_end_count property enables you to specify whether or not the x0_end_count trigger is a required condition for advancing to the next instruction. The x0_end_count condition is generated if:

  - The Instruction/AddressCommands/x0_address property is set to increment, and the X0 segment has reached the maximum of the row address countrange specified in the AddressCounter/Function/CountRange property of the memory TCD file.

  - The Instruction/AddressCommands/x0_address property is set to decrement, and the X0 segment has reached the minimum of the row address countrange specified in the AddressCounter/Function/CountRange property of the memory TCD file.

  The value on specifies that the x0_end_count trigger is a condition required to generate a next condition.

  The value off specifies that the x0_end_count trigger is not condition required to generate a next condition. This is the default.

  These usage conditions apply:

  - The Controller/AlgorithmResourceOptions/address_segment_x0_y0_allowed is on in the MemoryBist wrapper of the DftSpecification.

  - Specify x0_end_count when:

    - The number of row address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

    - The number_x0_bits is greater than zero for the selected address register.

  - To use the x0_end_count condition the instruction must have the Instruction/AddressCommands/x0_address property specified either to increment or decrement.

- x1_end_count : on | off;

  The x1_end_count property enables you to specify whether or not the x1_end_count trigger is a required condition for advancing to the next instruction. The x1_end_count condition is generated if:

  - The Instruction/AddressCommands/x1_address property is set to increment, and the X1 segment has reached the maximum of the row address countrange specified in the AddressCounter/Function/CountRange property of the memory TCD file.

  - The Instruction/AddressCommands/x1_address property is set to decrement, and the X1 segment has reached the minimum of the row address countrange specified in the AddressCounter/Function/CountRange property of the memory TCD file.

  The value on specifies that the x1_end_count trigger is a condition required to generate a next condition.

The value off specifies that the x1_end_count trigger is not a condition required to generate a next condition. This is the default.

These usage conditions apply:

- o Specify x1_end_count when the number of row address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

- o To use the x1_end_count condition the instruction must have the Instruction/AddressCommands/x1_address property specified either to increment or decrement.

- y0_end_count : on | off;

The y0_end_count property enables you to specify whether or not the y0_end_count trigger is a required condition for advancing to the next instruction. The y0_end_count condition is generated if:

- o The Instruction/AddressCommands/y0_address property is set to increment, and the Y0 segment has reached the maximum of the column address countrange specified in the AddressCounter/Function/CountRange property of the memory TCD file.

- o The Instruction/AddressCommands/y0_address property is set to decrement, and the Y0 segment has reached the minimum of the column address countrange specified in the AddressCounter/Function/CountRange property of the memory TCD file.

The value on specifies that the y0_end_count trigger is a condition required to generate a next condition.

The value off specifies that the y0_end_count trigger is not a condition required to generate a next condition. This is the default.

These usage conditions apply:

- o The Controller/AlgorithmResourceOptions/address_segment_x0_y0_allowed is on in the MemoryBist wrapper of the DftSpecification.

- o Specify y0_end_count when:

  - The number of column address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

  - The number_y0_bits is greater than zero for the selected address register.

- o To use the y0_end_count condition, the instruction must have the Instruction/AddressCommands/y0_address property specified either to increment or decrement.

- y1_end_count : on | off;

  The y1_end_count property enables you to specify whether or not the y1_end_count trigger is a required condition for advancing to the next instruction. The Y1_EndCount condition is generated if:

  - The Instruction/AddressCommands/y1_address property is set to increment, and the Y1 segment has reached the maximum of the column address countrange specified in the AddressCounter/Function/CountRange property of the memory TCD file.

  - The Instruction/AddressCommands/y1_address property is set to decrement, and the Y1 segment has reached the minimum of the column address countrange specified in the AddressCounter/Function/CountRange property of the memory TCD file.

  The value on specifies that the y1_end_count trigger is a condition required to generate a next condition.

  The value off specifies that the y1_end_count trigger is not a condition required to generate a next condition. This is the default.

  These usage conditions apply:

  - Specify y1_end_count when the number of column address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

  - To use the y1_end_count condition the instruction must have the Instruction/AddressCommands/y1_address property specified either to increment or decrement.

- z_end_count : on | off;

  The z_end_count property enables you to specify whether or not the z_end_count trigger is a required condition for advancing to the next instruction. The Z_EndCount condition is generated if:

  - The Instruction/AddressCommands/z_address property is set to increment, and the Z address has reached the maximum of the bank address countrange specified in the AddressCounter/Function/CountRange property of the memory TCD file.

  - The Instruction/AddressCommands/z_address property is set to decrement, and the Z address has reached the minimum of the bank address countrange specified in the AddressCounter/Function/CountRange property of the memory TCD file.

  The value on specifies that the z_end_count trigger is a condition required to generate a next condition.

  The value off specifies that the z_end_count trigger is not a condition required to generate a next condition. This is the default.

  These usage conditions apply:

  - Specify z_end_count when the number of bank address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD file is greater than zero.

- o   To use the z_end_count condition the instruction must have the Instruction/
  AddressCommands/z_address property specified either to increment or decrement.

- counter_a_end_count : on | off;

  The counter_a_end_count property enables you to specify whether or not the
  counter_a_end_count trigger is a required condition for advancing to the next instruction.
  The counter_a_end_count condition is generated when the CounterA is incremented, and
  the counter has reached the value specified by the Algorithm/TestRegisterSetup/
  load_counter_a_end_count property.

  The on value specifies that the counter_a_end_count trigger is a condition required to
  generate a next condition.

  The value *off* specifies that the counter_a_end_count trigger is not a condition required to
  generate a next condition. This is the default.

  To use the counter_a_end_count condition the instruction must have the Instruction/
  CounterCommands/counter_a property specified to increment.

- delay_counter_end_count : on | off;

  The delay_counter_end_count property enables you to specify whether or not the
  delay_counter_end_count trigger is a required condition for advancing to the next
  instruction. The delay_counter_end_count condition is generated when the DelayCounter is
  incremented and the counter has reached the value specified by the Algorithm/
  TestRegisterSetup/load_delay_counter property.

  The on value specifies that the delay_counter_end_count trigger is a condition required to
  generate a next condition.

  The value off specifies that the delay_counter_end_count trigger is not a condition required
  to generate a next condition. This is the default.

  To use the delay_counter_end_count condition the instruction must have the Instruction/
  CounterCommands/delay_counter property specified to increment.

- RepeatLoopA | RepeatLoopB wrapper

  The RepeatLoop wrapper groups the properties of the Repeat Loop module used to repeat
  execution of one or several sequential instructions.

  A group of sequential instructions for RepeatLoopA is defined from the instruction
  specified by RepeatLoopA/branch_to_instruction to the instruction specifying
  RepeatLoopA in the NextConditions wrapper.

  This group of sequential instructions is re-run for each Repeat sub-wrapper in RepeatLoopA
  with modified address and data commands as specified within the Repeat sub-wrapper.
  RepeatLoop(B) is similar.

  When the loop of instructions are repeated for each Repeat sub-wrapper the RepeatLoop
  condition becomes true. One or both of the RepeatLoop sub-wrappers may be specified in
  one instruction.

The RepeatLoopA and RepeatLoopB loops may also be nested. When loops are nested the inner loop re-runs each Repeat sub-wrapper containing properties that modify the address and data commands. However, each of these Repeat wrappers is also influenced by the outer loop Repeat wrapper's properties that modify the address and data commands.

These usage conditions apply:

- o   Each of the RepeatLoopA and RepeatLoopB wrappers may be used only once in the MicroProgram wrapper.

- o   If both RepeatLoopA and RepeatLoopB are specified in the NextConditions wrapper of different instructions, the sequential set of instructions for RepeatLoopA must be completely nested in the sequential set of instructions for RepeatLoopB. Similarly, the sequential set of instructions for RepeatLoopB must be completely nested in the sequential set of instructions for RepeatLoopA.

- o   If both RepeatLoopA and RepeatLoopB are specified in the same NextConditions sub-wrapper, the sequential set of instructions for RepeatLoopA must be completely nested in the sequential set of instructions for RepeatLoopB.

- RepeatLoopA | RepeatLoopB / branch_to_instruction : *instruction_name* ;

  The branch_to_instruction property enables you to specify the instruction that the pointer control selects as the next instruction to be run if all requested conditions are true (except for the RepeatLoop condition—that is all Repeat wrappers have not been run). The next conditions are described in the NextConditions wrapper.

  This property is mandatory inside any RepeatLoopA | RepeatLoopA wrapper specified.

  The value *instruction_name* is a valid Instruction wrapper name. The default value is the Instruction wrapper name for which this property was specified within.

  This property is used as follows:

  - o   The *instruction_name* must identify an Instruction wrapper name.

  - o   *instruction_name* specified can be the Instruction wrapper name for which the branch_to_instruction property is specified of a previously specified instruction. That pointer control only supports a branch to itself or a branch backwards.

  - o   If both RepeatLoopA and RepeatLoopB wrappers are specified in the same RepeatLoop wrapper the following must be true:

    - RepeatLoopB/branch_to_instruction must specify the same instruction or an instruction sequentially specified before the RepeatLoopA/ branch_to_instruction. That is RepeatLoopA must be nested with RepeatLoopB.

  - o   If both RepeatLoopA and RepeatLoopB wrappers are used in the MicroProgram wrapper but NOT in the same instruction one of the following must be true:

    - RepeatLoopA consisting of the sequential group of instructions from the specified RepeatLoopA/branch_to_instruction property to the instruction specifying RepeatLoopA must be nested within Repeat LoopB consisting of the

sequential group of instructions from the specified RepeatLoopB/ branch_to_instruction to the instruction specifying RepeatLoopB. RepeatLoopA must be nested within RepeatLoopB.

- Similarly to the above, RepeatLoopB must be nested within RepeatLoopA.

- Repeat LoopA consisting of the sequential group of instructions from the specified RepeatLoopA/branch_to_instruction to the instruction specifying RepeatLoopA does not use any of the same instructions as RepeatLoopB consisting of the sequential group of instructions from the specified RepeatLoopB/branch_to_instruction to the instruction specifying RepeatLoopB. That is RepeatLoopA and RepeatLoopB do not share any of the same instructions and thus are not nested.

- RepeatLoopA | RepeatLoopB / Repeat1 | Repeat2 | Repeat3 wrapper

The Repeat wrappers groups the properties that enable you to modify the address sequencing, write data sequencing, and expect data sequencing for the repeated instruction. These instruction modifications apply to the sequential group of instructions from the instruction specified by the RepeatLoopA/branch_to_instruction, and RepeatLoopB/ branch_to_instruction property, respectively, to the instruction specifying RepeatLoop.

A minimum of one Repeat wrapper must be specified per RepeatLoop wrapper. The Repeat wrapper can be specified up to a maximum of three times per the RepeatLoop wrapper.

- enable : on | off;

The enable property activates the repetition of the group of instructions in the RepeatLoop and applies the settings within the associated Repeat1, Repeat2 or Repeat3 wrapper.

- address_sequence : no_change | inverse;

The address_sequence property enables you to specify that the address segment commands run by instructions during the Repeat either be performed as specified by the instruction, or the command is modified to a complimentary command.

Valid values are as follows:

  o no_change — specifies that the address sequencing is to remain as programmed for execution of the Repeat wrapper. This is the default.

  o inverse — specifies that the address sequencing is to be modified to the complementary command from the specified command in each instruction within the RepeatLoop wrapper.

These usage conditions apply:

  o If RepeatLoopA and RepeatLoopB are not nested the resulting no_change and inverse modifications for the address_sequence property are shown in Table 7-6.

Table B-3. Un-Nested address_sequence Property Modification

x1_address x0_address y1_address y0_address z_address	RepeatLoop(A | B)/ Repeat/ address_sequence	Modified x1_address Modified x0_address Modified y1_address Modified y0_address Modified z_address
hold	no_change	hold
hold	inverse	hold
increment	no_change	increment
increment	inverse	decrement
decrement	no_change	decrement
decrement	inverse	increment
load_min	no_change	load_min
load_min	inverse	load_min
load_max	no_change	load_max
load_max	inverse	load_max

o If RepeatLoopA and RepeatLoopB are nested the resulting no_change and inverse modifications for the address_sequence property are shown in Table 7-7.

Table B-4. Nested address_sequence Property Modification

x1_address x0_address y1_address y0_address z_address	RepeatLoop(A)/ Repeat/ address_sequence "-" indicates the Repeat is not active	RepeatLoop(B)/ Repeat/ address_sequence "-" indicates the Repeat is not active	Modified x1_address Modified x0_address Modified y1_address Modified y0_address Modified z_address
hold	no_change	-	hold
hold	inverse	-	hold
hold	no_change	no_change	hold
hold	no_change	inverse	hold
hold	inverse	no_change	hold
hold	inverse	inverse	hold
hold	-	no_change	hold
hold	-	inverse	hold
increment	no_change	-	increment
increment	inverse	-	decrement

Table B-4. Nested address_sequence Property Modification (cont.)

x1_address x0_address y1_address y0_address z_address	RepeatLoop(A)/ Repeat/ address_sequence "-" indicates the Repeat is not active	RepeatLoop(B)/ Repeat/ address_sequence "-" indicates the Repeat is not active	Modified x1_address Modified x0_address Modified y1_address Modified y0_address Modified z_address
increment	no_change	no_change	increment
increment	no_change	inverse	decrement
increment	inverse	no_change	decrement
increment	inverse	inverse	increment
increment	-	no_change	increment
increment	-	inverse	decrement
decrement	no_change	-	decrement
decrement	inverse	-	increment
decrement	no_change	no_change	decrement
decrement	no_change	inverse	increment
decrement	inverse	no_change	increment
decrement	inverse	inverse	decrement
decrement	-	no_change	decrement
decrement	-	inverse	increment
load_min	no_change	-	load_min
load_min	inverse	-	load_min
load_min	no_change	no_change	load_min
load_min	no_change	inverse	load_min
load_min	inverse	no_change	load_min
load_min	inverse	inverse	load_min
load_min	-	no_change	load_min
load_min	-	inverse	load_min
load_max	no_change	-	load_max
load_max	inverse	-	load_max
load_max	no_change	no_change	load_max
load_max	no_change	inverse	load_max
load_max	inverse	no_change	load_max

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Table B-4. Nested address_sequence Property Modification (cont.)

x1_address x0_address y1_address y0_address z_address	RepeatLoop(A)/ Repeat/ address_sequence "-" indicates the Repeat is not active	RepeatLoop(B)/ Repeat/ address_sequence "-" indicates the Repeat is not active	Modified x1_address Modified x0_address Modified y1_address Modified y0_address Modified z_address
load_max	inverse	inverse	load_max
load_max	-	no_change	load_max
load_max	-	inverse	load_max

- expect_data_sequence : <u>no_change</u> | inverse;

  The expect_data_sequence property enables you to specify that the expect data commands run by instructions during the Repeat either be performed as specified by the instruction, or the command is modified to a complimentary command.

  Valid values are as follows:

  - no_change — specifies that the expect data sequencing is to remain as programmed for execution of the Repeat. This is the default.

  - inverse — specifies that the expect data sequencing is to be modified to the complementary command from the specified command in each instruction within the RepeatLoop.

  These usage conditions apply:

  - If RepeatLoopA and RepeatLoopB are not nested the resulting no_change and inverse modifications for the expect data command are shown in Table 7-8.

Table B-5. Un-Nested expect_data_sequence Property Modification

expect_data	RepeatLoop(A | B)/ Repeat/ expect_data_sequence	Modified expect_data
zero	no_change	zero
zero	inverse	one
one	no_change	one
one	inverse	zero
data_reg	no_change	data_reg
data_reg	inverse	inverse_data_reg
inverse_data_reg	no_change	inverse_data_reg
inverse_data_reg	inverse	data_reg
data_reg_rotate	no_change	data_reg_rotate

Table B-5. Un-Nested expect_data_sequence Property Modification (cont.)

expect_data	RepeatLoop(A | B)/ Repeat/ expect_data_sequence	Modified expect_data
data_reg_rotate	inverse	inverse_data_reg_rotate
inverse_data_reg_rotate	no_change	inverse_data_reg_rotate
inverse_data_reg_rotate	inverse	data_reg_rotate
data_reg_rotate_with_invert	no_change	data_reg_rotate_with_invert
data_reg_rotate_with_invert	inverse	inverse_data_reg_rotate_with_invert
inverse_data_reg_rotate_with_invert	no_change	inverse_data_reg_rotate_with_invert
inverse_data_reg_rotate_with_invert	inverse	data_reg_rotate_with_invert

o If RepeatLoopA and RepeatLoopB are nested the resulting no_change and inverse modifications for the expect data command are shown in Table 7-9.

Table B-6. Nested expect_data_sequence Property Modification

expect_data	RepeatLoopA/ Repeat/ expect_data_ sequence	RepeatLoopB/ Repeat/ expect_data_ sequence	Modified expect_data
zero	no_change	-	zero
zero	inverse	-	one
zero	no_change	no_change	zero
zero	no_change	inverse	one
zero	inverse	no_change	one
zero	inverse	inverse	zero
zero	-	no_change	zero
zero	-	inverse	one
one	no_change	-	one
one	inverse	-	zero
one	no_change	no_change	one
one	no_change	inverse	zero

Table B-6. Nested expect_data_sequence Property Modification (cont.)

expect_data	RepeatLoopA/ Repeat/ expect_data_ sequence	RepeatLoopB/ Repeat/ expect_data_ sequence	Modified expect_data
one	inverse	no_change	zero
one	inverse	inverse	one
one	-	no_change	one
one	-	inverse	zero
data_reg	no_change	-	data_reg
data_reg	inverse	-	inverse_data_reg
data_reg	no_change	no_change	data_reg
data_reg	no_change	inverse	inverse_data_reg
data_reg	inverse	no_change	inverse_data_reg
data_reg	inverse	inverse	data_reg
data_reg	-	no_change	data_reg
data_reg	-	inverse	inverse_data_reg
inverse_data_reg	no_change	-	inverse_data_reg
inverse_data_reg	inverse	-	data_reg
inverse_data_reg	no_change	no_change	inverse_data_reg
inverse_data_reg	no_change	inverse	data_reg
inverse_data_reg	inverse	no_change	data_reg
inverse_data_reg	inverse	inverse	inverse_data_reg
inverse_data_reg	-	no_change	inverse_data_reg
inverse_data_reg	-	inverse	data_reg
data_reg_rotate	no_change	-	data_reg_rotate
data_reg_rotate	inverse	-	inverse_data_reg_rotate
data_reg_rotate	no_change	no_change	data_reg_rotate
data_reg_rotate	no_change	inverse	inverse_data_reg_rotate
data_reg_rotate	inverse	no_change	inverse_data_reg_rotate
data_reg_rotate	inverse	inverse	data_reg_rotate

Table B-6. Nested expect_data_sequence Property Modification (cont.)

expect_data	RepeatLoopA/ Repeat/ expect_data_ sequence	RepeatLoopB/ Repeat/ expect_data_ sequence	Modified expect_data
data_reg_rotate	-	no_change	data_reg_rotate
data_reg_rotate	-	inverse	inverse_data_reg_rotate
inverse_data_reg_rotate	no_change	-	inverse_data_reg_rotate
inverse_data_reg_rotate	inverse	-	data_reg_rotate
inverse_data_reg_rotate	no_change	no_change	inverse_data_reg_rotate
inverse_data_reg_rotate	no_change	inverse	data_reg_rotate
inverse_data_reg_rotate	inverse	no_change	data_reg_rotate
inverse_data_reg_rotate	inverse	inverse	inverse_data_reg_rotate
inverse_data_reg_rotate	-	no_change	inverse_data_reg_rotate
inverse_data_reg_rotate	-	inverse	data_reg_rotate
inverse_data_reg_rotate_with_invert	no_change	-	data_reg_rotate_with_invert
data_reg_rotate_with_invert	inverse	-	inverse_data_reg_rotate_with_invert
data_reg_rotate_with_invert	no_change	no_change	data_reg_rotate_with_invert
data_reg_rotate_with_invert	no_change	inverse	inverse_data_reg_rotate_with_invert
data_reg_rotate_with_invert	inverse	no_change	inverse_data_reg_rotate_with_invert
data_reg_rotate_with_invert	inverse	inverse	data_reg_rotate_with_invert
data_reg_rotate_with_invert	-	no_change	data_reg_rotate_with_invert
data_reg_rotate_with_invert	-	inverse	inverse_data_reg_rotate_with_invert
inverse_data_reg_rotate_with_invert	no_change	-	inverse_data_reg_rotate_with_invert
inverse_data_reg_rotate_with_invert	inverse	-	data_reg_rotate_with_invert

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Table B-6. Nested expect_data_sequence Property Modification (cont.)

expect_data	RepeatLoopA/ Repeat/ expect_data_ sequence	RepeatLoopB/ Repeat/ expect_data_ sequence	Modified expect_data
inverse_data_reg_rotate_with_invert	no_change	no_change	inverse_data_reg_rotate_ with_invert
inverse_data_reg_rotate_with_invert	no_change	inverse	data_reg_rotate_with_inv ert
inverse_data_reg_rotate_with_invert	inverse	no_change	data_reg_rotate_with_inv ert
inverse_data_reg_rotate_with_invert	inverse	inverse	inverse_data_reg_rotate_ with_invert
inverse_data_reg_rotate_with_invert	-	no_change	inverse_data_reg_rotate_ with_invert
inverse_data_reg_rotate_with_invert	-	inverse	data_reg_rotate_with_inv ert

- write_data_sequence : <u>no_change</u> | inverse;

  The write_data_sequence property enables you to specify that the commands run by instructions during Repeat either be performed as specified by the instruction, or the command is modified to a complimentary command.

  Valid values are as follows:

  o no_change — specifies that the Write data sequencing is to remain as programmed for execution of the Repeat. This is the default.

  o inverse — specifies that the Write data sequencing is to be modified to the complementary command from the specified command in each instruction within RepeatLoop.

  These usage conditions apply:

  o If RepeatLoopA and RepeatLoopB are not nested the resulting no_change and inverse modifications for the write_data_sequence property are shown in Table B-7.

  ____ **Note** ____
  The other write_data *data_reg* commands (data_reg_rotate, inverse_data_reg_rotate, data_reg_rotate_with_invert, inverse_data_reg_rotate_with_invert, data_reg_prpg, and inverse_data_reg_prpg) follow the same behavior as data_reg and inverse_data_reg in Table B-7.

Table B-7. Un-Nested write_data_sequence Property Modification

write_data	RepeatLoop(A | B)/ Repeat/ write_data_sequence	Modified write_data
zero	no_change	zero
zero	inverse	one
one	no_change	one
one	inverse	zero
date_reg	no_change	date_reg
date_reg	inverse	inverse_date_reg
inverse_date_reg	no_change	inverse_date_reg
inverse_date_reg	inverse	date_reg
date_reg_rotate	no_change	date_reg_rotate
date_reg_rotate	inverse	inverse_date_reg_rotate
inverse_date_reg_rotate	no_change	inverse_date_reg_rotate
inverse_date_reg_rotate	inverse	date_reg_rotate
date_reg_rotate_with_invert	no_change	date_reg_rotate_with_invert
date_reg_rotate_with_invert	inverse	inverse_date_reg_rotate_with_invert
inverse_date_reg_rotate_with_ invert	no_change	inverse_date_reg_rotate_with_invert
inverse_date_reg_rotate_with_ invert	inverse	date_reg_rotate_with_invert

- If RepeatLoopA and RepeatLoopB are nested the resulting no_change and inverse modifications for the write_data_sequence property are shown in Table 7-11.

Table B-8. Nested write_data_sequence Property Modification

write_data	RepeatLoopA/ Repeat/ write_data_ sequence	RepeatLoopB/ Repeat/ write_data_ sequence	Modified write_data
zero	no_change	-	zero
zero	inverse	-	one
zero	no_change	no_change	zero
zero	no_change	inverse	one

Table B-8. Nested write_data_sequence Property Modification (cont.)

write_data	RepeatLoopA/ Repeat/ write_data_ sequence	RepeatLoopB/ Repeat/ write_data_ sequence	Modified write_data
zero	inverse	no_change	one
zero	inverse	inverse	zero
zero	-	no_change	zero
zero	-	inverse	one
one	no_change	-	one
one	inverse	-	zero
one	no_change	no_change	one
one	no_change	inverse	zero
one	inverse	no_change	zero
one	inverse	inverse	one
one	-	no_change	one
one	-	inverse	zero
date_reg	no_change	-	date_reg
date_reg	inverse	-	inverse_date_reg
date_reg	no_change	no_change	date_reg
date_reg	no_change	inverse	inverse_date_reg
date_reg	inverse	no_change	inverse_date_reg
date_reg	inverse	inverse	date_reg
date_reg	-	no_change	date_reg
date_reg	-	inverse	inverse_date_reg
inverse_date_reg	no_change	-	inverse_date_reg
inverse_date_reg	inverse	-	date_reg
inverse_date_reg	no_change	no_change	inverse_date_reg
inverse_date_reg	no_change	inverse	date_reg
inverse_date_reg	inverse	no_change	date_reg
inverse_date_reg	inverse	inverse	inverse_date_reg
inverse_date_reg	-	no_change	inverse_date_reg
inverse_date_reg	-	inverse	date_reg

Table B-8. Nested write_data_sequence Property Modification (cont.)

write_data	RepeatLoopA/ Repeat/ write_data_ sequence	RepeatLoopB/ Repeat/ write_data_ sequence	Modified write_data
date_reg_rotate	no_change	-	date_reg_rotate
date_reg_rotate	inverse	-	inverse_date_reg_rotate
date_reg_rotate	no_change	no_change	date_reg_rotate
date_reg_rotate	no_change	inverse	inverse_date_reg_rotate
date_reg_rotate	inverse	no_change	inverse_date_reg_rotate
date_reg_rotate	inverse	inverse	date_reg_rotate
date_reg_rotate	-	no_change	date_reg_rotate
date_reg_rotate	-	inverse	inverse_date_reg_rotate
inverse_date_reg_rotate	no_change	-	inverse_date_reg_rotate
inverse_date_reg_rotate	inverse	-	date_reg_rotate
inverse_date_reg_rotate	no_change	no_change	inverse_date_reg_rotate
inverse_date_reg_rotate	no_change	inverse	date_reg_rotate
inverse_date_reg_rotate	inverse	no_change	date_reg_rotate
inverse_date_reg_rotate	inverse	inverse	inverse_date_reg_rotate
inverse_date_reg_rotate	-	no_change	inverse_date_reg_rotate
inverse_date_reg_rotate	-	inverse	date_reg_rotate
date_reg_rotate_with_invert	no_change	-	date_reg_rotate _with_invert
date_reg_rotate_with_invert	inverse	-	inverse_date_reg_rotate_wi th_invert
date_reg_rotate_with_invert	no_change	no_change	date_reg_rotate_with_inver t
date_reg_rotate_with_invert	no_change	inverse	inverse_date_reg_rotate _with_invert
date_reg_rotate_with_invert	inverse	no_change	inverse_date_reg_rotate _with_invert
date_reg_rotate_with_invert	inverse	inverse	date_reg_rotate_with_inver t

Table B-8. Nested write_data_sequence Property Modification (cont.)

write_data	RepeatLoopA/ Repeat/ write_data_ sequence	RepeatLoopB/ Repeat/ write_data_ sequence	Modified write_data
date_reg_rotate_with_invert	-	no_change	date_reg_rotate _with_invert
date_reg_rotate_with_invert	-	inverse	inverse_date_reg_rotate_with_invert
inverse_date_reg_rotate_with _invert	no_change	-	inverse_date_reg_rotate_with_invert
inverse_date_reg_rotate_with _invert	inverse	-	date_reg_rotate_with_invert
inverse_date_reg_rotate_with _invert	no_change	no_change	inverse_date_reg_rotate _with_invert
inverse_date_reg_rotate_with _invert	no_change	inverse	date_reg_rotate _with_invert
inverse_date_reg_rotate_with _invert	inverse	no_change	date_reg_rotate_with_invert
inverse_date_reg_rotate_with _invert	inverse	inverse	inverse_date_reg_rotate_with_invert
inverse_date_reg_rotate_with _invert	-	no_change	inverse_date_reg_rotate_with_invert
inverse_date_reg_rotate_with _invert	-	inverse	date_reg_rotate _with_invert

- inhibit_last_address_count : on | off;

  The inhibit_last_address_count property enables you to prevent the selected AddressRegister from counting on the last execution of the selected instruction. Typically, this is used to prevent the address counter from wrapping on the last instruction execution from the maximum address to the minimum address or vice versa. Thus, on the next instruction a reverse address sequence is possible without requiring an additional instruction to change the address pointer.

  For the value off, any address segment command set to increment or decrement is performed normally. This is the default.

  The value on prevents the selected AddressRegister from counting on the last execution of the selected instruction when all requested NextConditions are true and the next sequential instruction is loaded for execution.

This property overrides the Instruction/AddressCommands/inhibit_last_address_count property when RepeatLoop is executing this Repeat wrapper.

This property is ignored when the following properties are defined as load_min or load_max: x_address, x1_address, y0_address, y1_address, z_address of the wrapper Instruction/AddressCommands.

- inhibit_data_compare : on | off;

The inhibit_data_compare property enables you to compare normally expected data and read data, or to disable any StrobeDataOut signal during running of the selected operation.

The value on specifies any StrobeDataOut signal is disabled and the expect data and read data are not compared.

The default value of off specifies that expected data and read data are compared normally.

OperationSet

The OperationSet wrapper specifies the name of the operation set that the memory BIST controller uses to generate waveforms that drive the memory.

Syntax

```
MemoryOperationsSpecification {
   OperationSet(operation_set_name) {
      SignalPipelineStages {
      }
      Operation (operation_name) {
      }
   }
}
```

Description

This section discusses the complete syntax for the OperationSet wrapper used in Tessent MemoryBIST. The OperationSet wrapper specifies the set of memory access operations used by the controller to apply the test algorithm.

The values for the library operation sets (Sync, SyncWR, SyncWRvcd, Async, AsyncWR and ROM) are reserved and cannot be used as custom operation set names.

> **Tip**
> Siemens EDA recommends using the library operation sets as templates to create your own operation sets. The library operation sets are available in the *<install_dir>/lib/technology/ icbist/lvision/example* folder of the tool tree. The library operation sets are generic and not optimized for one type of memory. Custom operation sets might be required to accommodate specific timing requirements or modes of operation.

Parameters

- *operation_set_name*

  The *operation_set_name* property is a unique identifier that specifies a built-in operation set or a user-defined access type. Valid values for *operation_set_name* are as follows:

  o Async— is a reserved string that specifies the library asynchronous operation set. This property applies to a memory for which the contents of a new location within its array are put at the outputs after an address change (a clock is not used in the read operation).

  o AsyncWR — is a reserved string that specifies a library asynchronous operation set for a multi-port memory.

  o ROM — is a reserved string that specifies the library operation set for ROMs.

- o Sync — is a reserved string that specifies the library synchronous operation set. This property applies to a memory for which the contents of a new location within its array are displayed at the outputs after an address change and a clock.

- o SyncWR — is a reserved string that specifies a library synchronous operation set for a multi-port memory.

- o SyncWRvcd — is a reserved string that specifies a library synchronous operation set for a multi-port memory. This property applies to a memory tested by the SMarchCHKBvcd algorithm.

- o *operationSetName* — is a user-defined identifier that matches the name of an operation set defined by the OperationSet wrapper. If you specify a user-defined name for the OperationSet property, Tessent MemoryBIST searches the specified memory library files for an OperationSet wrapper with the same identifier.

SignalPipelineStages

The optional SignalPipelineStages wrapper enables you to declare the number of pipelining stages required for a BIST function in a particular OperationSet.

Syntax

```
MemoryOperationsSpecification {
  OperationSet(operation_set_name) {
    SignalPipelineStages {
      select          : int ; // default: 0
      output_enable   : int ; // default: 0
      read_enable     : int ; // default: 0
      strobe_data_out : int ; // default: 0
      write_enable    : int ; // default: 0
      UserSignals {
        user<0...23>  : int ; // default: 0
      }
      DramSignals {
        activate      : int ; // default: 0
        precharge     : int ; // default: 0
        cas           : int ; // default: 0
        ras           : int ; // default: 0
        refresh       : int ; // default: 0
      }
    }
  }
}
```

Description

The waveform for the BIST function in a particular OperationSet is defined in the OperationSet/Operation wrapper. This generated waveform for the named BIST function is then pipelined by the number of stages specified by int as declared by the subsequent properties under the SignalPipelineStages wrapper.

The specified BIST function must be a Function assigned to a Core/Memory/Port in the memory TCD file. If the Function property of one of the memory ports is defined as validData in the memory TCD file, the number of pipeline stages for strobe_data_out must be 0.

Parameters

- select : *int* ; // default: 0

  Delays the select signal waveform, as specified by the select property in the Operation/Cycle wrapper, by *int* clock cycles. The default is 0.

- output_enable : *int* ; // default: 0

  Delays the output enable signal waveform, as specified by the output_enable property in the Operation/Cycle wrapper, by *int* clock cycles. The default is 0.

- read_enable : *int* ; // default: 0

  Delays the read enable signal waveform, as specified by the read_enable property in the Operation/Cycle wrapper, by *int* clock cycles. The default is 0.

- strobe_data_out : *int* ; // default: 0

  Delays the compare enable signal, as specified by the strobe_data_out property in the Operation/Cycle wrapper, by *int* clock cycles. The default is 0.

  > **_____ Note _____**
  >
  > If you specify a strobe_data_out delay greater than "0", in most cases you also need to specify a similar delay for the output_enable property.

- write_enable : *int* ; // default: 0

  Delays the write enable signal waveform, as specified by the write_enable property in the Operation/Cycle wrapper, by *int* clock cycles. The default is 0.

- UserSignals/user0 - user23: *int* ; // default: 0

  The user<n> (n=0,...,23) property delays the user<n> signal waveform, as specified by the user<n> property in the Operation/Cycle/UserSignals wrapper, by *int* clock cycles. The default is 0.

- DramSignals/activate : *int* ; // default: 0

  Delays the activate signal waveform, as specified by the activate property in the Operation/Cycle/DramSignals wrapper, by *int* clock cycles. The default is 0.

- DramSignals/precharge : *int* ; // default: 0

  Delays the precharge signal waveform, as specified by the precharge property in the Operation/Cycle/DramSignals wrapper, by *int* clock cycles. The default is 0.

- DramSignals/cas : *int* ; // default: 0

  Delays the CAS signal waveform, as specified by the cas property in the Operation/Cycle/DramSignals wrapper, by *int* clock cycles. The default is 0.

- DramSignals/ras : *int* ; // default: 0

  Delays the RAS signal waveform, as specified by the ras property in the Operation/Cycle/DramSignals wrapper, by *int* clock cycles. The default is 0.

- DramSignals/refresh : *int* ; // default: 0

  Delays the refresh signal waveform, as specified by the refresh property in the Operation/Cycle/DramSignals wrapper, by *int* clock cycles. The default is 0.

Operation

The Operation wrapper enables you to define memory operations based on a sequence of control, data, and address signal events.

Syntax

```
MemoryOperationsSpecification {
  OperationSet(operation_set_name) {
    Operation(operation_name) {
      multiplexed_address_padding : binary ; // default: 0
      AddressOverrides {
      }
      Cycle {
      }
    }
  }
}
```

Description

The Operation wrapper enables you to define memory operations based on a sequence of control, data, and address signal events. All event sequences are synchronous with respect to cycles of an implied BIST clock.

Parameters

- multiplexed_address_padding : *binary*;

  The multiplexed_address_padding property is used when the row address and column address widths are different, and they are multiplexed onto the same address port.

  The *binary* value specifies the padding bit values. By default, Tessent MemoryBIST pads the bits with all zeros.

  These usage conditions apply:

  - This property is only valid for a multiplexed-address bus.

  - *binary* must be exactly as wide as the difference between the RowAddress and ColumnAddress bus widths.

  - When the number of RowAddress bits is greater than the number of ColumnAddress bits, *binary* applies to the bits padded to the ColumnAddress. The address padding bits are always assigned to the most significant bits of the ColumnAddress bus.

  - When the number of ColumnAddress bits is greater than the number of RowAddress bits, *binary* applies to the bits padded to the RowAddress. The address padding bits are always assigned to the most significant bits of the RowAddress bus.

  - The value defined in row_address, column_address, bank_address overrides the value for the multiplexed_address_padding property.

The following illustrates how you can use multiplexed_address_padding when the RowAddress is 2-bits wider than the ColumnAddress. The binary 2'b11 is assigned to the most significant bits of the ColumnAddress bus.

```
Operation (AutoRefresh){
  multiplexed_address_padding: 2'b11;
  .
} //end of Operation wrapper
```

AddressOverrides

The AddressOverrides wrapper enables you to define properties for forcing a particular value on the row address, column address or bank address.

Syntax

```
MemoryOperationsSpecification {
  OperationSet(operation_set_name) {
    Operation(operation_name) {
      AddressOverrides {
        row_address              : binary ; // default: x
        column_address           : binary ; // default: x
        bank_address             : binary ; // default: x
      }
      Cycle {
      }
    }
  }
}
```

Description

All forced event sequences on row, column or bank address are synchronous with respect to cycles of an implied BIST clock.

Parameters

- row_address : *binary*;

  The row_address property is optional. This property enables you to identify the row address bits to be forced and specify the value that these bits are forced to.

  Valid values are as follows:

  - *binary* specifies the forced values that are to be applied to the address-bus.

  - You can specify a 1, 0 or an x for the *binary* value. Any unspecified MSB's default to x.

  This property can only be specified if the number of *RowAddress* bits in the LogicalAddress wrapper (Core/Memory/AddressCounter) of the memory TCD file is greater than zero.

  The specified range of *RowAddress* bits must be within the range of 0 to the number of *RowAddress* bits minus one, which is specified in the LogicalAddressMap wrapper of the memory TCD file.

The following example specifies that the *RowAddress* bits 11, 10, and 8 are forced to the values 1'b1, 1'b0, and 1'b1 respectively:

```
Operation (AutoRefresh){
  AddressOverrides {
    row_address : 12'b10x1xxxxxxxx;
  } //end of AddressOverrides wrapper
} //end of Operation wrapper
```

If the address-bus in this example is wider than 12 bits, all MSB's are forced to 'x'.

- column_address : *binary*;

  The column_address property is optional. This property enables you to identify the column address bits to be forced and specify the value that these bits are forced.

  Valid values are as follows:

  - *binary* specifies the forced values that are to be applied to the address-bus.

  - You can specify a 1, 0 or an x for the *binary* value. Any unspecified MSB's default to x.

  This property can only be specified if the number of *ColumnAddress* bits in the LogicalAddress wrapper (Core/Memory/AddressCounter) of the memory TCD file is greater than zero.

  The specified range of *ColumnAddress* bits must be within the range of 0 to the number of *ColumnAddress* bits minus one, which is specified in the LogicalAddressMap wrapper of the memory TCD file.

- bank_address : *binary*;

  The bank_address property is optional. This property enables you to identify the bank address bits to be forced and specify the value that these bits are forced.

  Valid values are as follows:

  - *binary* specifies the forced values that are to be applied to the address-bus.

  - You can specify a 1, 0 or an x for the *binary* value. Any unspecified MSB's default to x.

  This property can only be specified if the number of *BankAddress* bits in the LogicalAddress wrapper (Core/Memory/AddressCounter) of the memory TCD file is greater than zero.

  The specified range of *BankAddress* bits must be within the range of 0 to the number of *BankAddress* bits minus one, which is specified in the LogicalAddressMap wrapper of the memory TCD file.

Cycle

The Cycle wrapper corresponds to a clock cycle and enables you to identify the state of the signals during one memory BIST controller clock cycle.

Syntax

```
MemoryOperationsSpecification {
  OperationSet(operation_set_name) {
    Operation(operation_name) {
      AddressOverrides {
      }
      Cycle {
        select                : on | off | auto;
        read_enable           : on | off | auto;
        output_enable         : on | off | auto;
        strobe_data_out       : on | off;
        write_enable          : on | off | auto;
        even_group_write_enable : on | off | auto;
        odd_group_write_enable  : on | off | auto;
        bist_data_enable      : on | off | auto;
        AdvancedSignals {
        }
        UserSignals {
        }
        DramSignals {
        }
        ConcurrentPortSignals {
        }
      }
    }
  }
}
```

Description

The Cycle wrapper corresponds to a clock cycle and enables you to identify the state of the signals during one memory BIST controller clock cycle.

Parameters

- select: on | off | auto;

  The select property controls the memory port that uses the Function property (Core/ Memory/Port) specified to Select.

  Valid values are as follows:

  - on — activates the Select signal.

  - off — deactivates the Select signal.

  - The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

To specify the activation value use the Polarity property in the appropriate Core/Memory/ Port wrapper of the memory TCD file. This property affects your memory only if a Port wrapper in the memory TCD file specifies the Function property to select.

- read_enable: on | off | <u>auto</u>;

The read_enable property controls the memory port that uses the Function property (Core/ Memory/Port) specified to ReadEnable.

Valid values are as follows:

  - o   on — activates the ReadEnable signal.

  - o   off — deactivates the ReadEnable signal.

  - o   The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

To specify the activation value use the Polarity property in the appropriate Core/Memory/ Port wrapper of the memory TCD file. This property affects your memory only if a Port wrapper in the memory TCD file specifies the Function property to readenable.

- output_enable: on | off | <u>auto</u>;

The output_enable property enables you to enable output drivers of the embedded memory.

Valid values are as follows:

  - o   on — activates the output enable signal.

  - o   off — deactivates the output enable signal.

  - o   The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

Use the Polarity property in the appropriate Core/Memory/Port wrapper of the memory TCD file to specify the activation value. This property affects your memory only if a Port wrapper in the memory TCD file specifies the Function property to outputenable.

> **Note**
> If you do not specify output_enable in any operation and your design has an output enable signal, Tessent MemoryBIST creates an error indicating that the signal is not used.

- strobe_data_out: on | <u>off</u>;

The strobe_data_out property samples the data from the memory and compares the value with the expected data defined in the test algorithm.

The default value is off. strobe_data_out is active only for the clock cycle in which the property is specified.

These usage conditions apply:

- o Memory output data must be valid at the end of the clock cycle in which strobe_data_out occurs.

- o All OperationSet wrappers require at least one operation that contains a strobe_data_out.

- o A strobe_data_out property can be specified on any or all Cycles of any Operation as timing permits.

- o For memories where the BIST function *validData* is specified, the strobe_data_out property must be specified on the earliest cycle of the operation when data is available.

- o Pipelining of strobe_data_out may be specified using the PipelineDepth property in the Core/Memory wrapper or the OperationSet/SignalPipelineStages wrapper in the memory TCD file.

The following example specifies that the data is sampled from the memory for comparison with expected data pattern:

```
Operation (Read) {
  Cycle {
    read_enable: on;
  }
  Cycle {
    read_enable: off;
    strobe_data_out: on;
  }
}
```

- • write_enable: on | off | auto;

  The write_enable property enables you to control the memory WriteEnable signal.

  Valid values are as follows:

  - o on — activates the WriteEnable signal.

  - o off — deactivates the WriteEnable signal.

  - o The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

  To specify the activation value use the Polarity property in the appropriate Core/Memory/Port wrapper of the memory TCD file. This property affects your memory only if a Port wrapper in the memory TCD file specifies the Function property to WriteEnable.

  Refer to the Note in the odd_group_write_enable property description for further information.

- even_group_write_enable: on | off | <u>auto</u>;

  The even_group_write_enable property enables you to control the memory GroupWriteEnable signals that are assigned into the even set. This property also modifies the expect data register output of the current instruction. The even bits of the expect data register output are inverted in the cycle where even_group_write_enable is off and odd_group_write_enable is on.

  Valid values are as follows:

  - on — activates the even GroupWriteEnable signal.

  - off — deactivates the even GroupWriteEnable signal.

  - The default value of auto resolves to on for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

  To specify the activation value use the Polarity property in the appropriate Core/Memory/Port wrapper of the memory TCD file. This property affects your memory only if a Port wrapper in the memory TCD file specifies the Function property to GroupWriteEnable.

  Refer to the Note in the odd_group_write_enable property description for further information.

- odd_group_write_enable: on | off | <u>auto</u>;

  The odd_group_write_enable property enables you to control the memory GroupWriteEnable signals that are assigned into the odd set. This property also modifies the expect data register output of the current instruction. The odd bits of the expect data register output are inverted in the cycle where odd_group_write_enable is off and even_group_write_enable is on.

  Valid values are as follows:

  - on — activates the odd GroupWriteEnable signal.

  - off — deactivates the odd GroupWriteEnable signal.

  - The default value of auto resolves to on for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

  To specify the activation value use the Polarity property in the appropriate Core/Memory/Port wrapper of the memory TCD file. This property affects your memory only if a Port wrapper in the memory TCD specifies the Function property to GroupWriteEnable.

_____ **Note** _____
In the case where a memory includes port(s) with Function property of
GroupWriteEnable and does not include port(s) with Function WriteEnable, the
memory inputs controlled by the even_group_write_enable and
odd_group_write_enable signals are additionally gated with the write_enable signal by
adding an AND gate in the associated MBIST interface. If a custom operation set is
used, then write_enable must be set to On whenever odd_group_write_enable or
even_group_write_enable is set to On.
_____

- bist_data_enable: on | off | <u>auto</u>;

    o The default value of auto resolves to off for the first cycle and preserves the setting
      of the previous cycle for subsequent cycles.

Cycle/AdvancedSignals

The AdvancedSignals wrapper enables you to define specific addressing and data inversion behavior of an algorithm from the operation set.

Syntax

```
MemoryOperationsSpecification {
  OperationSet(operation_set_name) {
    Operation(operation_name) {
      Cycle {
        AdvancedSignals {
          column_address_count_enable : on | off | auto;
          row_address_count_enable    : on | off | auto;
          switch_address_register      : on | off;
          address_override_enable      : on | off | auto;
          invert_expect_data           : on | off;
          invert_write_data            : on | off;
        }
      }
    }
  }
}
```

Description

The AdvancedSignals wrapper enables you to define specific addressing and data inversion behavior of an algorithm from the operation set. All event sequences are synchronous with respect to cycles of an implied BIST clock.

Parameters

- column_address_count_enable: on | off | <u>auto</u>;

  The column_address_count_enable property enables you to enable counting of the column address within an operation. Typically, this property is used to perform back-to-back column address sequence.

  This property is limited to enabling counting of the Y0 or Y1 address segments of the selected address register. Any carryout from the Y address segment is ignored except on the last cycle of the operation. The column address count direction is determined by the y0_address and y1_address properties of the Instruction/AddressCommands wrapper.

  Valid values are as follows:

  - o on — enables counting of the Y Column Address segment.

  - o off — disables counting of the Y Column Address segment.

  The default value of auto resolves to off for the first cycle of each operation. During an operation, a signal transition remains in effect until the next transition occurs on that signal. For example, when a Cycle wrapper specifies column_address_count_enable: on, the column_address_count_enable signal transition remains active until another Cycle wrapper specifies column_address_count_enable: off.

These usage conditions apply:

- o This property is used only for programmable memory BIST controllers.

- o Address counting must be enabled in the Algorithm wrapper by specifying one of the following commands in order for column_address_count_enable to have an effect:

```
y0_address: increment;
y0_address: decrement;
y1_address: increment;
y1_address: decrement;
```

- o The settings shown in Table B-9 are required when using column_address_count_enable to increment/decrement the column address by 1 on every clock cycle and benefit from maximum circuit performance.

Table B-9. Required Settings for Increment/Decrement by 1

DftSpecification	max_y0_segment_bits: 1; //default
	
	max_x0_segment_bits: 1; //default
		Enables the implementation of the carry-look-ahead (CLA) circuit.

Algorithm	AddressRegisterA {
	
	  x1_carry_in: y1_carry_out;
	
	  y1_carry_in: none;
	
	}
		This is only an example. The address register could include bank address bits as well. The essential setting is highlighted in red.

Operation	`Operation (Write2CellsFastY) {`
	`  Cycle {`
	`    select: on;`
	`    write_enable: on;`
	`    read_enable: off;`
	`    AdvancedSignals {`
	`      column_address_count_enable: on;`
	`    }`
	`  }`
	`  Cycle {`
	`  }`
	`}`
		This is only an example. The essential characteristics are that the operation must be two cycles long and column_address_count_enable remains on in both cycles of the operation.

Algorithms and operation sets using these settings can be applied to any memory, so it is not necessary to run algorithms using freeze_step. The tool automatically selects the best implementation without changing the functionality, based on the number of column address bits as follows:

Number of Column Address Bits	Automatic Modification
0	The row address is incremented instead as if row_address_count_enable: on had been specified in the operation.
1	No change required to the algorithm or operation.

Number of Column Address Bits	Automatic Modification
2 or more	The algorithm and operation are automatically modified to segment the address register (number_y0_bits: 1 and number_x0_bits: 1).

_____**Note**_____

Algorithms explicitly specifying address register segmentation (that is number_y0_bits:1 and number_x0_bits: 1) can only be used for memories with 2 or more column address bits. It is therefore preferable to use the settings of the table to make the algorithm more general.

_____**Note**_____

Operations that are two cycles long but explicitly set column_address_count_enable: off in the second cycle are automatically modified to column_address_count_enable: on. This is necessary for the correct operation of the CLA circuit.

_____**Note**_____

The address at the start of the operation and after execution of the operation must be Even if the Instruction/AddressCommand wrapper address property is set to increment, and Odd if the address property is set to decrement, to obtain a linear address sequence.

o   The settings shown in Table B-10 are required when using column_address_count_enable to increment/decrement the column address by a value greater than one on every clock cycle.

Table B-10. Required Settings for Increment/Decrement by > 1

DftSpecification	max_y0_segment_bits: auto;	Turns off the implementation of the carry-look-ahead (CLA) circuit.

Algorithm	`AddressRegisterA {`
	`  x1_carry_in: y1_carry_out;`
	`  y0_carry_in: x1_carry_out;`
	`  `**`y1_carry_in: none;`**
	`}`
		This setting is only an example. The essential setting is highlighted in red.

Table B-10. Required Settings for Increment/Decrement by > 1 (cont.)

| Operation | ```
Operation (Write2CellsFastY) {
  Cycle {
    select: on;
    write_enable: on;
    read_enable: off;
    AdvancedSignals {
      column_address_count_enable: on;
    }
  }
  Cycle {
  }
}
``` | This is a typical example. The operation length is arbitrary and column_address_count_enable can be present in any cycle. |
|---|---|---|

These settings enable full flexibility of the address sequence but the resulting circuit runs at significantly lower speed than for the default (that is max_y0_segment_bits: 1 and max_x0_segment_bits: 1). The circuit is also larger because of the possible address segment arrangements.

Finally, algorithms using these settings are not compatible with memories with less than two column address bits and typically require the use of freeze_step: *int*. This increases test time because of more frequent controller setup operations.

- row_address_count_enable: on | off | <u>auto</u>;

The row_address_count_enable property enables you to enable counting of the row address within an operation. Typically, this property is used to perform back-to-back row address sequence.

This property is limited to enabling counting of the X0 or X1 address segments of the selected address register. Any carryout from the X address segment is ignored except on the last cycle of the operation. The row address count direction is determined by the x0_address and x1_address properties of the Instruction/AddressCommands wrapper.

Valid values are as follows:

- o on — enables counting of the X Row Address segment.

- o off — disables counting of the X Row Address segment.

The default value of auto resolves to off for the first cycle of each operation. During an operation, a signal transition remains in effect until the next transition occurs on that signal. For example, when a Cycle wrapper specifies row_address_count_enable: on, the row_address_count_enable signal transition remains active until another Cycle wrapper specifies row_address_count_enable: off.

These usage conditions apply:

- o This property is used only for programmable memory BIST controllers.

o Address counting must be enabled in the Algorithm wrapper by specifying one of the following commands in order for row_address_count_enable to have an effect:

```
x0_address: increment;
x0_address: decrement;
x1_address: increment;
x1_address: decrement;
```

o The settings shown in Table B-11 are required when using row_address_count_enable to increment/decrement the row address by 1 on every clock cycle and benefit from maximum circuit performance.

Table B-11. Required Settings for Increment/Decrement by 1

DftSpecification	max_x0_segment_bits: 1; //default	Enables the implementation of the carry-look-ahead (CLA) circuit.

Algorithm	`AddressRegisterA {`
	`  y1_carry_in: x1_carry_out;`
	`  `**`x1_carry_in: none;`**
	`}`
		This is only an example. The address register could include bank address bits as well. The essential setting is highlighted in red.

Operation	`Operation (Write2CellsFastY) {`
	`  Cycle {`
	`    select: on;`
	`    write_enable: on;`
	`    read_enable: off;`
	`    AdvancedSignals {`
	`      `**`row_address_count_enable: on;`**
	`    }`
	`  }`
	`  Cycle {`
	`  }`
	`}`
		This is only an example. The essential characteristics are that the operation must be two cycles long and row_address_count_enable remains on in both cycles of the operation.

___ **Note** ___

Algorithms explicitly specifying address register segmentation (that is number_x0_bits:1) can also be used. Memories always have at least four rows, so address segmentation is possible.

___ **Note** ___

Operations that are two cycles long but explicitly set row_address_count_enable: off in the second cycle are automatically modified to row_address_count_enable: on. This is necessary for the correct operation of the CLA circuit.

___ **Note** ___

The address at the start of the operation and after execution of the operation must be Even if the Instruction/AddressCommand wrapper address property is set to increment, and Odd if the address property is set to decrement, to obtain a linear address sequence.

o The settings shown in Table B-12 are required when using row_address_count_enable to increment/decrement the row address by a value greater than one on every clock cycle.

Table B-12. Required Settings for Increment/Decrement by > 1

| DftSpecification | ```
max_y0_segment_bits: auto;
``` | Disables the implementation of the carry-look-ahead (CLA) circuit. |
|---|---|---|
| Algorithm | ```
AddressRegisterA {
  y1_carry_in: x1_carry_out;
  x0_carry_in: y1_carry_out;
  x1_carry_in: none;
}
``` | This setting is only an example. The essential setting is highlighted in red. |
| Operation | ```
Operation (Write2CellsFastX) {
  Cycle {
    select: on;
    write_enable: on;
    read_enable: off;
    AdvancedSignals {
      row_address_count_enable: on;
    }
  }
  Cycle {
  }
}
``` | This is a typical example. The operation length is arbitrary and row_address_count_enable can be present in any cycle. |

These settings enable full flexibility of the address sequence but the resulting circuit runs at significantly lower speed than for the default (max_x0_segment_bits: 1). The circuit is also larger because of the possible address segment arrangements.

- switch_address_register: on | off;

The switch_address_register property is used to switch between the two address registers that the algorithm uses — Home and Away.

For the value on, a signal is generated by the signal generator to force the memory address to select the other Address Register that is not being used by the current instruction. For example, if the current instruction uses AddressRegisterA, the memory address selects AddressRegisterB when this new property is set to on. Similarly — if B is used rather than A.

If the value is off, the tool does not switch between the two address registers. This is the default.

These usage conditions apply:

o You need to have specific operation name that use this property.

o The switch_address_register property works for operations with two Cycles.

o The switch_address_register property can only be forced to on in the second Cycle of the Operation.

- o   If the operation requires three of more Cycles, the property should be set to off in the third Cycle — you can only set it to on in the second Cycle wrapper.

- o   The instruction before the current instruction must have expect_data property defined in the Instruction/DataCommands wrapper.

- o   The assumption is made that the Away cell and the Home cell hold different values.

- address_override_enable: on | off | <u>auto</u>;

  - o   The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

- invert_expect_data: on | <u>off</u>;

  The invert_expect_data property enables you to invert the data pattern, specified by the expect_data property (Instruction/DataCommands) of an algorithm, so that the data pattern can be modified on every clock cycle within an operation.

  Valid values are as follows:

  - o   on — inverts the expect data.

  - o   off — does not invert the expect data. This is the default.

  Data pattern changes within an operation are limited to the pattern specified by the expect_data of the corresponding algorithm instruction and its inverse.

- invert_write_data: on | <u>off</u>;

  The invert_write_data property enables you to invert the data pattern specified by the write_data property (Instruction/DataCommands) of an algorithm, so that the data pattern can be modified on every clock cycle within an operation.

  Valid values are as follows:

  - o   on — inverts the write data.

  - o   off — does not invert the write data. This is the default.

  Data pattern changes within an operation are limited to the pattern specified by the write_data of the corresponding algorithm instruction and its inverse.

Cycle/UserSignals

The UserSignals wrapper enables you to control the memory port that uses the Function property specified to user<n>.

Syntax

```
MemoryOperationsSpecification {
  OperationSet(operation_set_name) {
    Operation(operation_name) {
      Cycle {
        UserSignals {
          user<0...23> : on | off | auto;
        }
      }
    }
  }
}
```

Description

The UserSignals wrapper enables you to control the memory port that uses the Function property specified to user<n>. All event sequences are synchronous with respect to cycles of an implied BIST clock. The default value of auto resolves to off for the first cycle of an operation. During subsequent cycles of an operation, a signal transition remains in effect until the next transition on that signal is specified.

Parameters

- user0 - user23: on | off | auto;

  The user<n> (n=0,...,23) property controls the memory port that uses the Function property specified to User<n>.

  Valid values are as follows:

  - on — activates the user<n> signal.

  - off — deactivates the user<n> signal.

  - The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

  To specify the activation value use the Polarity property in the appropriate Core/Memory/ Port wrapper of the memory TCD file.

  For the first Cycle wrapper of each operation the default value user<n>: off is applied. During an operation, a signal transition remains in effect until the next transition on that signal. For example, once a Cycle wrapper specifies user<n>: on, the user<n> signal remains active until another Cycle wrapper specifies the user<n> property to off.

  This property affects your memory only if a Port wrapper for that memory specifies the Function property to user<n>.

Cycle/DramSignals

The DramSignal wrapper enables you to control memory ports that specify various Function properties.

Syntax

```
MemoryOperationsSpecification {
  OperationSet(operation_set_name) {
    Operation(operation_name) {
      Cycle {
        DramSignals {
          activate        : on | off | auto;
          precharge       : on | off | auto;
          cas             : on | off | auto;
          ras             : on | off | auto;
          refresh         : on | off | auto;
          address_select  : column | row | auto;
        }
      }
    }
  }
}
```

Description

The DramSignal wrapper enables you to control memory ports that specify various Function properties. All event sequences are synchronous with respect to cycles of an implied BIST clock.

Parameters

- activate: on | off | auto;

  The activate property controls the memory port that uses the Function property specified to activate.

  Valid values are as follows:

  - on — enables the Activate signal.

  - off — disables the Activate signal.

  - The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

  To specify the activation value use the Polarity property in the appropriate Core/Memory/ Port wrapper of the memory TCD file. This property affects your memory only if a Port wrapper for that memory specifies Function Activate.

- precharge: on | off | auto;

  The precharge property controls the memory port that uses the Function property specified to precharge.

Valid values are as follows:

- o   on — activates the Precharge signal.

- o   off — deactivates the Precharge signal.

- o   The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

To specify the activation value use the Polarity property in the appropriate Core/Memory/Port wrapper of the memory TCD file. This property affects your memory only if a Port wrapper in the memory TCD file specifies the Function property to Precharge.

- cas: on | off | auto;

  The cas property controls the memory port that uses the Function property specified to cas.

  Valid values are as follows:

  - o   on — activates the CAS signal.

  - o   off — deactivates the CAS signal.

  - o   The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

  To specify the activation value use the Polarity property in the appropriate Core/Memory/Port wrapper of the memory TCD file. This property affects your memory only if a Port wrapper in the memory TCD file specifies Function CAS.

- ras: on | off | auto;

  The ras property controls the memory port that uses the Function property specified to ras.

  Vlid values are as follows:

  - o   on — activates the RAS signal.

  - o   off — deactivates the RAS signal.

  - o   The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

  To specify the activation value use the Polarity property in the appropriate Core/Memory/Port wrapper of the memory TCD file. This property affects your memory only if a Port wrapper in the memory TCD file specifies the Function property to RAS.

- refresh: on | off | auto;

  The refresh property controls the memory port that uses the Function property specified to refresh.

  Valid values are as follows:

  - o   on — activates the Refresh signal.

  - o   off — deactivates the Refresh signal.

    o   The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

To specify the activation value use the Polarity property in the appropriate Core/Memory/ Port wrapper of the memory TCD file. This property affects your memory only if a Port wrapper in the memory TCD file specifies the Function property to Refresh.

- address_select: column | row | <u>auto</u>;

The address_select property controls the row or column values driven onto a multiplexed address port.

Valid values are as follows:

    o   column — drives the column address values to the address port when multiplexed addressing is used. For non-multiplexed addresses, this Cycle assignment is ignored.

    o   row — drives the row segment values to the address port when multiplexed addressing is used. For non-multiplexed addresses, this Cycle assignment is ignored.

    o   The default value of auto resolves to row for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

        Example:

        For the first Cycle wrapper of each operation the default is address_select : row. During an operation, a signal transition remains in effect until the next transition occurs on that signal. For example, when a Cycle wrapper specifies address_select: column, the Address signal transition remains active until another Cycle wrapper specifies address_select: row.

This property is ignored when the LogicalAddressMap (Core/Memory/AddressCounter) does not specify RowAddress bits, or ColumnAddress bits, and the RowAddress must be multiplexed with the ColumnAddress.

Cycle/ConcurrentPortSignals

The ConcurrentPortSignals wrapper enables you to activate and deactivate concurrent memory read and write properties to help detect faults specific to multi-port memories.

Syntax

```
MemoryOperationsSpecification {
  OperationSet(operation_set_name) {
    Operation(operation_name) {
      Cycle {
        ConcurrentPortSignals {
          read_enable          : on | off | auto;
          write_enable         : on | off | auto;
          read_column_address  : on | off | auto;
          read_row_address     : on | off | auto;
          write_row_address    : on | off | auto;
          write_column_address : on | off | auto;
          write_data_polarity  : no_change | inverse | auto;
          even_group_write_enable : on | off | auto;
          odd_group_write_enable  : on | off | auto;
        }
      }
    }
  }
}
```

Description

All event sequences specified in the ConcurrentPortSignals wrapper are synchronous with respect to cycles of an implied BIST clock.

Concurrent read and write operations provide more flexibility than ShadowRead and ShadowWrite operations. Concurrent operations enable modification of both the row and column address from the operation set, and are therefore preferred when creating custom operation sets for a programmable controller. ShadowRead only enables modification to the row address and ShadowWrite is only used by library algorithms and is not controllable from the operation set.

Parameters

- read_enable: on | off | auto;

  The read_enable property enables you to activate/deactivate the concurrent read.

  The concurrent read enable has the following limitations:

  o Concurrent read is not supported for ROM or 1RW memories.

  o The read_enable property cannot be turned on or off per algorithm instruction. If an algorithm uses operations with and without concurrent read, two versions of the operation must be defined in the operation set.

Valid values are as follows:

- o on — activates the concurrent read.

- o off — deactivates the concurrent read.

- o The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

Use the read_enable property with the read_column_address and read_row_address properties. You can specify the concurrent read properties — read_column_address, read_row_address, and read_enable — in one or more Cycle/ConcurrentPortSignals wrappers of an operation, thereby enabling concurrent read during any read or write cycle.

In general, the result of the concurrent read operation is not compared to an expected value. This is the case when using library operation sets. However, custom operation sets can perform this compare for one of the logical ports being used to perform concurrent read operations. The logical port being compared is the one selected for the current test port. A compare is performed when all of the following conditions are present:

- o No read operation is required by the algorithm on the logical port with read capability, which is part of the current test port. The port is said to be inactive.

- o Concurrent read is enabled by adding a Cycle wrapper in a custom operation set that contains one of the following combinations of properties:

```
Cycle {
  select: on;
  read_enable: off;
...
  ConcurrentPortSignals {
    read_enable: on;
    ...
  }
}

Cycle {
  select: off;
  read_enable: on;
...
  ConcurrentPortSignals {
    read_enable: on;
    ...
  }
}

Cycle {
  select: off;
  read_enable: off;
...
  ConcurrentPortSignals {
    read_enable: on;
    ...
  }
}
```

This causes the memory output to be updated with the value read at the reference address determined by the algorithm if both ConcurrentReadColumnAddress and ConcurrentReadRowAddress are set to off, or to an address located in an adjacent column if ConcurrentReadColumnAddress is set to on, or to an address located in an adjacent row if ConcurrentReadRowAddress is set to on. This is true as long as the current test port is composed of two logical ports. If the current test port only involves a logical port of type RW, no concurrent read operation is performed because the logical port is always under control of the algorithm.

o A strobe is present in the same cycle (asynchronous read port) or the next cycle (synchronous read port) wrapper. A strobe is normally not used for the Cycle wrapper combinations listed above and is only useful for specialized diagnostic tests. The expected value is determined by the value of the expect_data property specified in the algorithm.

- write_enable: on | off | auto;

  The write_enable property is used to detect faults specific to multi-port memories.

  This property enables you to perform a write operation from inactive ports on a memory cell located in a column or row that is adjacent to the memory cell at the reference address indicated by the algorithm. The write_data_polarity property can be used to invert the data value written from the inactive port.

  The concurrent write operation has the following limitations:

  o Concurrent write is restricted to 2RW, 1R1W and nR2W memories.

  o Concurrent write cannot be turned on or off per algorithm instruction. If an algorithm uses operations with and without concurrent write, two versions of the operation must be defined in the operation set.

  Valid values are as follows:

  o on — Activates the write access signals for the inactive write ports in the specified cycle as shown in the table below:

Table B-13. write _enable Write Access Operation When Explicitly Specified

write_column_address	write_row_address	write_enable	Write Access for Inactive Port
n/a	n/a	Specified to Off	Inactive
Off	Off	Specified to On	To reference cell
Off	On	Specified to On	To adjacent row
On	Off	Specified to On	To adjacent column
On	On	Specified to On	To diagonal address

  o off — Deactivates the concurrent write operation.

o auto — The default value depends on the write_row_address and write_column_address settings to maintain compatibility with existing operation sets that do not specify write_enable. Default settings and write access behavior are shown in the table below:

Table B-14. write_enable Write Access Operation Default Settings

write_column_address	write_row_address	write_enable	Write Access for Inactive Port
Off	Off	Defaults to Off	Inactive
Off	On	Defaults to On	To adjacent row
On	Off	Defaults to On	To adjacent column
On	On	Defaults to On	To diagonal address

Use the write_enable property with the write_column_address and write_row_address properties. You can specify the concurrent write properties — write_column_address, write_row_address and write_enable — in one or more Cycle/ConcurrentPortSignals wrappers of an operation, thereby enabling concurrent write during any read or write cycle.

_____ **Caution** _____
Performing a concurrent write simultaneously with a read or write access from the inactive port at the reference address may corrupt the memory data.
_____

- read_column_address: on | off | <u>auto</u>;

The read_column_address property enables you to select the column address of the concurrent read operation.

The concurrent read column address has the following limitations:

o The inactive read address is limited to the adjacent column or row.

o read_column_address is not supported for ROM or 1RW memories.

o The read_column_address property cannot be turned on or off per algorithm instruction. If an algorithm uses operations with and without concurrent read, two versions of the operation must be defined in the operation set.

Valid values are as follows:

o on — activates the concurrent read column address. Concurrent read inverts the least significant column address bit for memories with column address bits or the least significant row address bit for memories without column address bits on the inactive read port.

o off — deactivates the concurrent read column address. The address of the inactive port is controlled by the normal test port logic.

o The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

These usage conditions apply:

o read_column_address takes effect only if Cycle/ConcurrentPortSignals/read_enable is set to on.

o read_column_address is ignored if Core/Memory/ConcurrentRead property is set to off.

o read_column_address has the same effect as read_row_address for memories without column address bits. That is, the concurrent read row address is activated.

o You can specify the concurrent read properties — read_column_address, read_row_address, and read_enable — in one or more Cycle wrappers of an operation, thereby enabling concurrent read during any read or write cycle.

- read_row_address: on | off | auto;

The read_row_address property enables you to select the row address of the concurrent read operation.

The concurrent read row address has the following limitations:

o The inactive read address is limited to the adjacent row.

o read_row_address is not supported for ROM or 1RW memories.

o The read_row_address property cannot be turned on or off per algorithm instruction. If an algorithm uses operations with and without concurrent read, two versions of the operation must be defined in the operation set.

Valid values are as follows:

o on — activates the concurrent read row address. Concurrent read inverts the least significant row address bit on the inactive read port.

o off — deactivates the concurrent read row address. The address of the inactive port is controlled by the normal test port logic.

o The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

These usage conditions apply:

o read_row_address takes effect only if read_enable is set to on.

o read_row_address is ignored if the ConcurrentRead property of Core/Memory is set to off.

o read_row_address is ignored if the memory does not have a row address.

- o You can specify the concurrent read properties — read_column_address, read_row_address, and read_enable — in one or more Cycle wrappers of an operation, thereby enabling concurrent read during any read or write cycle.

- write_row_address: on | off | auto;

  The write_row_address property enables you to select the concurrent write row address and to activate the concurrent operation.

  The concurrent write row address has the following limitations:

  - o The inactive write address is limited to the adjacent row.

  - o The concurrent write is restricted to 2RW, 1R1W and nR2W memories. Limiting the memory types prevent simultaneous write contention.

  - o The concurrent write cannot be turned On/Off per algorithm instruction. If an algorithm uses operations with and without concurrent write, two versions of the operation must be defined in the operation set.

  Valid values are as follows:

  - o on — activates the concurrent write row address. Concurrent write inverts the least significant row address bit on the inactive write port.

  - o off — deactivates the concurrent write row address. The address of the inactive port is controlled by the normal test port logic.

  - o The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

  The concurrent write properties — write_row_address, write_column_address, and write_data_priority — can be specified in one or more Cycle wrappers of an operation and allow concurrent write during any read or write cycle.

- write_column_address: on | off | auto;

  The write_column_address property enables you to select the concurrent write column address and to activate the concurrent operation.

  The concurrent write column address has the following limitations:

  - o The inactive write address is limited to the adjacent column.

  - o The concurrent write is restricted to 2RW, 1R1W, and nR2W memories. Limiting the memory types prevent simultaneous write contention.

  - o The concurrent write cannot be turned On/Off per algorithm instruction. If an algorithm uses operations with and without concurrent write, two versions of the operation must be defined in the operation set.

  Valid values are as follows:

  - o on — activates the concurrent write column address. Concurrent write inverts the least significant column address bit on the inactive write port.

o off — deactivates the concurrent write column address. The address of the inactive port is controlled by the normal test port logic.

o The default value of auto resolves to off for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

The concurrent write properties — write_column_address, write_row_address, and write_data_priority — can be specified in one or more Cycle wrappers of an operation and enable concurrent write during any read or write cycle.

- write_data_polarity: no_change | inverse | <u>auto</u>;

The write_data_polarity property enables you to select the data pattern of the concurrent write operation.

Valid values are as follows:

o no_change — applies the current active port data to the inactive write port during the concurrent write operation.

o inverse — applies the complement of the current active port data to the inactive write port during the concurrent write operation.

o The default value of auto resolves to no_change for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

These usage conditions apply:

o The concurrent write properties — write_data_polarity, write_column_address, and write_row_address — can be specified in one or more Cycle wrappers of an operation and enable concurrent write during any read or write cycle.

o This property is only valid if either write_column_address or write_row_address is set to on.

The following example applies the complement of the active port data pattern to the inactive write port during the concurrent write operation at the adjacent column.

```
Operation (AutoRefresh){
  Cycle {
    ConcurrentPortSignals {
      write_column_address: on;
      write_data_polarity: inverse;
      .
    } // end of ConcurrentPortSignals wrapper
  } //end of Cycle wrapper
} //end of Operation wrapper
```

- even_group_write_enable: on | off | <u>auto</u>;

The even_group_write_enable property enables you to control the memory GroupWriteEnable signals that are assigned into the even set, during the concurrent write operation. This feature may be useful when creating custom algorithms and operation sets for memories with write-only ports.

Valid values are as follows:

- o on — activates the even GroupWriteEnable signal on the inactive memory write ports during concurrent write operations.

- o off — deactivates the even GroupWriteEnable signal on the inactive memory write ports during concurrent write operations.

- o The default value of auto resolves to on for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

These usage conditions apply:

- o This property affects your memory only if a Port wrapper in the memory TCD for that memory specifies the Function property to GroupWriteEnable.

- o This property affects your memory only if it has the ConcurrentWrite property set to on in the memory TCD.

- o This property is only effective when the ConcurrentPortSignals/write_enable property is activated in the same Cycle wrapper of the OperationSet.

- odd_group_write_enable: on | off | <u>auto</u>;

  The odd_group_write_enable property enables you to control the memory GroupWriteEnable signals that are assigned into the odd set, during the concurrent write operation. This feature may be useful when creating custom algorithms and operation sets for memories with write-only ports.

  Valid values are as follows:

  - o on — activates the odd GroupWriteEnable signal on the inactive memory write ports during concurrent write operations.

  - o off — deactivates the odd GroupWriteEnable signal on the inactive memory write ports during concurrent write operations.

  - o The default value of auto resolves to on for the first cycle and preserves the setting of the previous cycle for subsequent cycles.

  These usage conditions apply:

  - o This property affects your memory only if a Port wrapper in the memory TCD for that memory specifies the Function property to GroupWriteEnable.

  - o This property affects your memory only if it has the ConcurrentWrite property set to on in the memory TCD.

  - o This property is only effective when the ConcurrentPortSignals/write_enable property is activated in the same Cycle wrapper of the OperationSet.

Appendix C
MemoryBIST Algorithms

Siemens EDA provides a library of test patterns or algorithms for testing your memories. These algorithms represent some of the tests that you can perform on your memories using Tessent Shell's MemoryBIST programmable controllers. You can use the detailed algorithm examples provided in this appendix as a basis for creating your own custom algorithms.

This appendix provides the following information about each algorithm:

- A brief description of the algorithm

- Algorithm length

- The controller address and data setup for the algorithm

- The algorithm sequence of instructions

- An example of the algorithm programming syntax

- Identifies the detected faults

- Identifies the availability of the algorithm

Any abbreviations used in this appendix are described in the "Notation Describing MemoryBIST Algorithms" section.

This appendix covers the following topics:

Notation Describing MemoryBIST Algorithms

This chapter uses the notation described below for the MemoryBIST algorithms:

- **R0** — Read current location and compare most significant output bit to 0.

- **R1** — Read current location and compare most significant output bit to 1.

- **Rc** — Read current location and compress contents into a MISR.

- **W0** — Write to current location, applying 0 to the least significant input bit.

- **W1** — Write to current location, applying 1 to the least significant input bit.

- **M** — Multi-cycle delay (always equal to 4).

- **A** — Number of address locations.

- **R** — Number of row address locations.

- **C** — Number of column address locations.

- **BRC** — Begin row cycle.

- **ERC** — End row cycle.

- **%operationType** — Number of clock cycles specified for an operation.

- **MemRst** — Test phase run when the following property is on:

  PatternsSpecification/Patterns/TestStep/MemoryBist/AdvancedOptions/memory_reset

- **Wait** — Optional pause for parallel retention test.

- **PRT** — Test phase run for parallel retention test.

MemoryBIST Algorithm Detected Faults

The Tessent Shell MemoryBIST algorithms can detect a variety of fault types. The comparisons provided in the table can be used to choose the algorithms that best suit your needs.

Table C-1. MemoryBIST Algorithm Fault Detection

Fault Type	SMarch	SMarchCHKB	SMarchCHKBci	SMarchCHKBcil	SMarchCHKBvcd		LVMarchX	LVMarchY	LVMarchCMinus	LVMarchLA	LVRowBar	LVColumnBar	LVGalPat	LVGalColumn	LVGalRow	LVCheckerboard1X1	LVCheckerboard4X4	LVWalkingPat	LVBitSurroundDisturb
Stuck-At Faults	F 5-9	F 11-14	F 11-14	F 11-14	F 11-14		F	F	F	F	F	F	F	F	F	F	F	F	F
Stuck-Open Faults	F 5-9	F 11-14	F 11-14	F 11-14	F 11-14		F	F	F	F	F	F	F	F	F	F	F	F	F
Address Decoder Faults	F 5-9	F 11-14	F 11-14	F 11-14	F 11-14		F	F	F	F			F	F	F			F	F
Transition Faults	F 5-9	F 11-14	F 11-14	F 11-14	F 11-14		F	F	F	F			F	F	F			F	F
Inversion Coupling Faults	F 5-9	F 11-14	F 11-14	F 11-14	F 11-14		F	F	F	F			F	F	F			F	F
Idempotent Coupling Faults	F 5-9	F 11-14	F 11-14	F 11-14	F 11-14				F	F			F	F	F			F	F
Dynamic Coupling Faults	F 5-9	F 11-14	F 11-14	F 11-14	F 11-14				F	F			F	F	F			F	
Data Retention Faults	F 5,6	F 6,7	F 6,7	F 6,7	F 6,7														
Write Recovery Faults	F 5-9	F 11-14	F 11-14	F 11-14	F 11-14		F		F										F
Destructive Read Faults	F 7,8	F 13,14	F 13,14	F 13,14	F 6,7								F	F	F				F
Read Disturb Faults	F 7,8	F 13,14	F 13,14	F 13,14	F 6,7		F	F	F	F			F	F	F			F	
Write Disturb Faults	F 5-9	F 11-14	F 11-14	F 11-14	F 11-14					F									

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Table C-1. MemoryBIST Algorithm Fault Detection (cont.)

Fault Type	SMarch	SMarchCHKB	SMarchCHKBci	SMarchCHKBcil	SMarchCHKBvcd	LVMarchX	LVMarchY	LVMarchCMinus	LVMarchLA	LVRowBar	LVColumnBar	LVGalPat	LVGalColumn	LVGalRow	LVCheckerboard1X1	LVCheckerboard4X4	LVWalkingPat	LVBitSurroundDisturb
Single Port Bitline Coupling Faults	G[1]	G[1]	F 5.5,6.5	F 5.5,6.5	F 5.5,6.5 6,7													
Access Transistor Current Leakage Faults				F 15,18	F 15,18							F	F	F			F	F
Data Path Shorts					F 2													
Bit/Group/Global Write Enable Faults					F 3													
Read Enable Faults					F 3.6													
Memory Select Faults					F 3.7													
Multiport Synchronous Bitline Coupling Faults	G[2]	G[2]	G[3]	G[3]	F 5.1,6.2			G	G									G
Multi-Port Interference Faults	G[2]	G[2]	G[4]	G[4]	F 5.5,6.5	G	G	G	G	G	G	G	G	G	G	G	G	G

Table Legend:

```
F              — Full coverage
G              — Good coverage
no entry       — Low/No coverage
numeric entry  — The algorithm phase(s) where the fault is detected
```

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Notes:

1. Approximately 50% coverage

2. Shadow read only

3. Shadow write cannot be applied to a memory that has rows numbering as a power of two unless it has an input port with Function type ShadowAddressEnable. The coverage of bitline coupling faults might be slightly reduced in this case.

4. Column shadow write cannot be applied to memory with write-only ports unless it is capable of performing a row cycle when the write enable input is set to its inactive value. This requires a test mode of operation of the memory. The coverage of multi-port interference faults might be slightly reduced if such a test mode is not available.

MemoryBIST Algorithm Test Times

The algorithm complexity and a test time calculation are provided in the table for each Tessent MemoryBIST algorithm.

Table C-2. MemoryBist Algorithm Test Times

Algorithm	Algorithm Complexity	Test Time @ 200MHz (ms)	Number of Instructions
SMarch	22N	0.11	14
SMarchCHKB	26N	0.13	19
SMarchCHKBci	36N	0.18	27
SMarchCHKBcil	44N	0.23	49
SMarchCHKBvcd	68N	0.35	90
ReadOnly	4N	0.02	3
LVMarchX	8N	0.04	3
LVMarchY	12N	0.06	4
LVMarchCMinus	12N	0.06	3
LVMarchLA	28N	0.14	5
LVRowBar	8N	0.04	5
LVColumnBar	8N	0.04	5
LVGalPat	$(8*(Nx*Ny)^2+12Nx*Ny)*Nz$	42.0	6
LVGalColumn	$(8Ny*(Nx)^2+12Nx*Ny)*Nz$	10.55	7
LVGalRow	$(8Nx*(Ny)^2+12Nx*Ny)*Nz$	0.23	7

Table C-2. MemoryBist Algorithm Test Times (cont.)

Algorithm	Algorithm Complexity	Test Time @ 200MHz (ms)	Number of Instructions
LVCheckerboard1X1	8N	0.04	3
LVCheckerboard4X4	8N	0.04	3
LVWalkingPat	$(4*(Nx*Ny)^2+16Nx*Ny)*Nz$	21.05	5
LVBitSurroundDisturb	140N	0.72	8

Where:

- **N** — Equivalent to Nx*Ny*Nz

- **Nx** — Number of rows

- **Ny** — Number of columns

- **Nz** — Number of banks

The test time results are for a memory with 1k words, organized as 256 rows of four columns, with a 200 MHz tester clock. The number of bits per word is not important for this calculation.

Test times for memories of different word sizes and organization can be calculated using the equation given in the "Algorithm Complexity" table row. The resulting value is then multiplied by the tester clock period to obtain the total test time for that memory and algorithm.

The algorithm complexity of all LV* algorithms can be reduced by using custom algorithms and operation sets that remove redundant cycles. In some cases, it is necessary to slightly modify the algorithm itself to make use of complex operations. The test time reduction ranges that are possible varies between 10% to 50%. Refer to "Optimizing Custom Algorithms and Operation Sets" for further information.

During Parallel Static Retention Testing (PSRT), the algorithms listed in Table C-3 have shorter test durations because of the reduced number of algorithm test phases that are performed during PSRT sub-phases. The test times noted in the table do not include the pauses between PSRT sub-phases. Refer to the respective algorithm test description table for information on which algorithm phases are run during each PSRT sub-phase.

Table C-3. MemoryBist Algorithm PSRT Test Times

Algorithm	Algorithm Complexity	Test Time @ 200MHz (ms)	Number of Instructions
SMarch	10N	0.05	6
SMarchCHKB	6N	0.03	9
SMarchCHKBci	6N	0.03	9
SMarchCHKBcil	6N	0.03	9

Table C-3. MemoryBist Algorithm PSRT Test Times (cont.)

Algorithm	Algorithm Complexity	Test Time @ 200MHz (ms)	Number of Instructions
SMarchCHKBvcd	8N	0.04	34

Available Library Algorithms

The following describes each algorithm in the Siemens EDA algorithm library.

SMarch Algorithm

The memory BIST controller performs all operations using fast row accesses. In a fast row count sequence, the column address remains constant until the memory BIST controller accesses all rows.

Table C-4 describes the SMarch algorithm per test port.

Table C-4. Description of SMarch Test Algorithm per Test Port

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
1*+	0		Idle	NoOperation	Performs multi-cycle initialization.

Table C-4. Description of SMarch Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
2*+	1,2	1	(RxW1) (R1W1)	ReadModifyWrite ReadModifyWrite	Scans 1s through first word.
3*+	3,4	1	(R1W0) (R0Rx)	ReadModifyWrite ReadRead	Scans 0s through first word.
4*+	5	2 to W Fast row	(W0Rx)	Write	Writes 0s in all other words.
5**	6,7	1 to W Fast row	(R0W1) (R1W1)	ReadModifyWrite ReadModifyWrite	Reads 0s and replaces with 1s.
6***	8,9	1 to w Fast row	(R1W0) (R0W0)	ReadModifyWrite ReadModifyWrite	Reads 1s and replaces with 0s.
7	10,11	w to 1 Fast row	(R0W1)(R1Rx)	ReadModifyWrite ReadRead	Reads 0s and replaces with 1s, reverse address sequence. Does back-to-back reads before changing address.
8	10,11	W to 1 Fast row	(R1W0) (R0Rx)	ReadModifyWrite ReadRead	Reads 1s and replace with 0s. Does back-to-back reads before changing address.
9	12,13	1 to W Fast row	(R0W0) (R0Rx)	ReadModifyWrite ReadRead	Only the first read operation is significant.

Where:

x+	Algorithm phases performed during MemRst
x*	Algorithm phases performed during the PSRT start_to_pause sub-phase
x**	Algorithm phases performed during the PSRT pause_to_pause sub-phase
x***	Algorithm phases performed during the PSRT pause_to_end sub-phase

Detected Faults

The SMarch algorithm detects the failure modes indicated in Table C-1.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Test Time

The time required for the memory BIST controller to test your design using the SMarch algorithm is outlined in Table C-2.

Specification

To test SRAMs using the SMarch algorithm, specify Algorithm SMarch in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

ReadOnly Algorithm

The ReadOnly algorithm is the default test algorithm that the memory BIST controller uses to test ROMs. The ReadOnly algorithm is a simple two-pass algorithm that reads and compresses the ROM contents by traversing the address space in both ascending and descending order.

For ROMs with multiple read ports, memory BIST provides separate MISRs for each port and repeats the test for each port while performing shadow reads on the inactive ports. Shadow reads perform normal read operations to strategic addresses but do not compress the results.

The memory BIST controller performs all operations using fast column accesses. In a fast column count sequence, the row address remains constant until the memory BIST controller accesses all columns.

Table C-5 describes the ReadOnly algorithm per test port.

Table C-5. Description of ReadOnly Test Algorithm per Test Port

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
1	0		Idle	None	Performs multi-cycle initialization.
2	1	1 to W Fast column	(Rc)	Read	Reads and compresses the ROM contents.
3	1	W to 1 Fast column	(Rc)	Read	Reads and compresses the ROM contents in reverse address sequence.
4	2	N/A	(MISR Compare)	CompareMISR	Compares the GO bit based on the final signature.

Detected Faults

The ReadOnly algorithm detects the following failure modes:

- Stuck to opposite value cell faults. A stuck to opposite value fault is a single memory cell stuck at a logic 1 when the expected value is a logic 0 (or stuck at a logic 0 when the expected value is a logic 1).

- Address decoder faults. These faults result in any of the following: any given address does not access any cells, any given address simultaneously accesses multiple cells, and multiple addresses access a single cell.

Test Time

The time required for the memory BIST controller to test your design using the ReadOnly algorithm is outlined in Table C-2.

Specification

To test ROMs using the ReadOnly algorithm, specify Algorithm as ReadOnly in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

SMarchCHKB Algorithm

When you specify Algorithm equal to SMarchCHKB, the memory BIST controller performs all operations using either fast column or fast row accesses. In a fast row count sequence, the column address remains constant until the memory BIST controller accesses all of the rows. In a fast column count sequence, the row address remains constant until the memory BIST controller accesses all of the columns.

In the algorithm description, Phases 2 and 3 ensure that the memory BIST controller can access the memory by writing and reading to the first address location. Phases 5, 6, and 7 use normal cycles with a checkerboard pattern. The 0s and 1s in the other phases refer to solid zero and solid one patterns.

Table C-6 describes the SMarchCHKB algorithm per test port.

Table C-6. Description of SMarchCHKB Test Algorithm per Test Port

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
1*+	0		Idle	NoOperation	Performs multi-cycle initialization.

Table C-6. Description of SMarchCHKB Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
2*+	1,2	1	(RxW1) (R1W1)	ReadModifyWrite ReadModifyWrite	Scans 1s through first word.
3*+	3,4	1	(R1W0) (R0Rx)	ReadModifyWrite ReadRead	Scans 0s through first word.
4*	5		Idle	NoOperation	Performs multi-cycle initialization.
5*	6	1 to W Fast column	(RxW0)	ReadModifyWrite	Writes checkerboard background data.
		-	Optional Wait	None	Pauses the testbench to perform the static retention test. (There is no hardware in the controller that performs the static retention test.)
6**	7	1 to W Fast column	(R0W1)	ReadModifyWrite	Reads checkerboard background, and replaces it with inverse checkerboard data.
		-	Optional Wait	None	Pauses the testbench to perform the static retention test. (There is no hardware in the controller that performs the static retention test.)
7***	8	1 to W Fast column	(R1W0)	ReadModifyWrite	Reads inverse checkerboard data. Memory contents are now a don't care.
8	9	-	Idle	NoOperation	Performs multi-cycle initialization.
9	10,11	1	(RxW0) (R0Rx)	ReadModifyWrite ReadRead	Scans 0s through first word.
10+	12	2 to W Fast row	(W0Rx)	Write	Writes 0s in all other words.

Table C-6. Description of SMarchCHKB Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
11	13,14	1 to W Fast row	(R0W1) (R1W1)	ReadModifyWrite ReadModifyWrite	Reads 0s and replaces with 1s.
12	13,14	1 to W Fast row	(R1W0) (R0W0)	ReadModifyWrite ReadModifyWrite	Reads 1s and replaces with 0s.
13	15,16	W to 1 Fast row	(R0W1) (R1Rx)	ReadModifyWrite ReadRead	Reads 0s and replaces with 1s. Does back-to-back reads before changing address.
14	15,16	W to 1 Fast row	(R1W0) (R0Rx)	ReadModifyWrite ReadRead	Reads 1s and replaces with 0s. Does back-to-back reads before changing address.
15	17,18	1 to W Fast row	(R0W0) (R0Rx)	ReadModifyWrite ReadRead	Only the first read operation is significant.

Where:

x^+	Algorithm phases performed during MemRst
x*	Algorithm phases performed during the PSRT start_to_pause sub-phase
x**	Algorithm phases performed during the PSRT pause_to_pause sub-phase
x***	Algorithm phases performed during the PSRT pause_to_end sub-phase

Detected Faults

The SMarchCHKB algorithm detects the failure modes indicated in Table C-1.

Test Time

The time required for the memory BIST controller to test your design using the SMarchCHKB algorithm is outlined in Table C-2.

Specification

To test SRAMs using the SMarchCHKB algorithm, specify Algorithm SMarchCHKB in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

SMarchCHKBci Algorithm

The SMarchCHKBci algorithm is the default test algorithm that the memory BIST controller uses to test RAMs. This algorithm is similar to the SMarchCHKB test algorithm, and accommodates synchronous and asynchronous SRAMs with single or multiple ReadWrite ports. The algorithm is capable of detecting signal coupling between bitlines of adjacent columns and port interference faults that are caused by high resistance ground connections to one of the N-channel source terminals.

A few steps are added to the current algorithm to make sure that the appropriate data combinations are applied to every cell. The entire algorithm is shown in Table C-7.

Table C-7. Description of SMarchCHKBci Test Algorithm per Test Port

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
1*+	0		Idle	NoOperation	
2*+	1,2	1	(RxW1) (R1W1)	ReadModifyWrite ReadModifyWrite	Scans 1s through first word.
3*+	3,4	1	(R1W0) (R0Rx)	ReadModifyWrite ReadRead	Scans 0s through first word. A back-to-back read is performed before changing address.
4*	5		Idle	NoOperation	Performs Multi-cycle Initialization
5*	6	1 to W Fast column	(RxW0)	ReadModifyWrite	Writes checkerboard background data.
5.5	7	1 to W Fast column	(R0W0)	ReadModifyWrite	Reads the checkerboard background while *ShadowWrite* is on. Writes checkerboard data.
		-	Optional Wait	None	Pauses the test to perform the static retention test.

Table C-7. Description of SMarchCHKBci Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
6**	8	1 to W Fast column	(R0W1)	ReadModifyWrite	Reads checkerboard background. Replaces it with Inverse Checkerboard.
6.5	9	1 to W Fast column	(R1W1)	ReadModifyWrite	Reads Inverse Checkerboard Pattern while *ShadowWrite* is on, Write Inverse Checkerboard Pattern.
		-	Optional Wait	None	Pauses the test to perform the static retention test.
7***	10	1 to W Fast column	(R1W0)	ReadModifyWrite	Reads Inverse Checkerboard data, replaces it with checkerboard data.
8	11	-	Idle	NoOperation	Performs Multi-cycle Initialization.
9	12,13	1	(RxW0) (R0Rx)	ReadModifyWrite ReadRead	Scans 0s through the first word. A back-to-back read is performed before changing address.
10+	14	2 to W Fast row	(W0Rx)	Write	Writes 0s in all words.
11	15,16	1 to W Fast row	(R0W1) (R1W1)	ReadModifyWrite ReadModifyWrite	Reads all 0s replaces them with 1s.
12	15,16	1 to W Fast row	(R1W0) (R0W0)	ReadModifyWrite ReadModifyWrite	Reads 1s, and replaces them with 0s.
13	17,18	w to1 Fast row	(R0W1) (R1Rx)	ReadModifyWrite ReadRead	Reads 0s, and replaces them with 1s. A back-to-back read is performed before changing address.
14	17,18	w to 1 Fast row	(R1W0) (R0Rx)	ReadModifyWrite ReadRead	Reads 1s, and replaces them with 0s. A back-to-back read is performed before changing address.

Table C-7. Description of SMarchCHKBci Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
15	19	w to 1	(R0W0)	ReadModifyWrite	Reads 0 while *ShadowWrite* is on. Writes 0.
16	20,21	1 Fast row	(RxW1) (R1Rx)	ReadModifyWrite ReadRead	Writes 1 through the first word. A back-to-back read is performed before changing address.
17	22	2 to W Fast row	(W1Rx)	Write	Writes 1 to all locations.
18	23	1 to W Fast row	(R1W1)	ReadModifyWrite	Reads 1 while *ShadowWrite* is on and writes it back.
19	24,25	1	(R1Wmemcontents) (WmemcontentsRx)	ReadModifyWrite Write	Writes the memory content into the first word (default 0), and writes it back.
20	26	2 to W Fast row	(Wmemcontents Rmemcontents)	Write	Writes memory contents to the rest of the memory (default 0).

Where:

x^+	Algorithm phases performed during MemRst
x*	Algorithm phases performed during the PSRT start_to_pause sub-phase
x**	Algorithm phases performed during the PSRT pause_to_pause sub-phase
x***	Algorithm phases performed during the PSRT pause_to_end sub-phase

Detected Faults

The SMarchCHKBci algorithm detects the failure modes indicated in Table C-1.

Test Time

The time required for the memory BIST controller to test your design using the SMarchCHKBci algorithm is outlined in Table C-2.

Specification

To test SRAMs using the SMarchCHKBci algorithm, specify Algorithm SMarchCHKBci in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

SMarchCHKBcil Algorithm

This algorithm is similar to the SMarchCHKBci test algorithm and accommodates synchronous and asynchronous memories with single or multiple ReadWrite ports. The algorithm is capable of detecting bitline and wordline current leakage defects in single and multi-port memories.

A few steps are added to the current algorithm to make sure that the appropriate data combinations are applied to every cell. Table C-8 shows the entire algorithm.

Table C-8. Description of SMarchCHKBcil Test Algorithm per Test Port

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
1*+	0		Idle	NoOperation	
2*+	1,2	1	(RxW1) (R1W1)	ReadModifyWrite ReadModifyWrite	Scans 1s through first word.
3*+	3,4	1	(R1W0) (R0Rx)	ReadModifyWrite ReadRead	Scans 0s through first word. A back-to-back read is performed before changing address.
4*	5		Idle	NoOperation	Performs Multi-cycle Initialization
5*	6	1 to W Fast column	(RxW0)	ReadModifyWrite	Writes checkerboard background data.
5.5	7	1 to W Fast column	(R0W0)	ReadModifyWrite	Reads the checkerboard background while *ShadowWrite* is on. Writes checkerboard data.
		-	Optional Wait	None	Pauses the test to perform the static retention test.

Table C-8. Description of SMarchCHKBcil Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
6**	8	1 to W Fast column	(R0W1)	ReadModifyWrite	Reads checkerboard background. Replaces it with Inverse Checkerboard.
6.5	9	1 to W Fast column	(R1W1)	ReadModifyWrite	Reads Inverse Checkerboard Pattern while *ShadowWrite* is on. Writes Inverse Checkerboard Pattern.
		-	Optional Wait	None	Pauses the test to perform the static retention test.
7***	10	1 to W Fast column	(R1W0)	ReadModifyWrite	Reads Inverse Checkerboard data, replaces it with checkerboard data.
8	11	-	Idle	NoOperation	Performs Multi-cycle Initialization.
9	12,13	1	(RxW0) (R0Rx)	ReadModifyWrite ReadRead	Scans 0s through the first word. A back-to-back read is performed before changing address.
10+	14	2 to w Fast row	(W0Rx)	Write	Writes 0s in all words.
11	15,16	1 to w Fast row	(R0W1) (R1W1)	ReadModifyWrite ReadModifyWrite	Reads all 0s, replaces them with 1s.
12	15,16	1 to w Fast row	(R1W0) (R0W0)	ReadModifyWrite ReadModifyWrite	Reads 1s, and replaces them with 0s.
13	17,18	w to1 Fast row	(R0W1) (R1Rx)	ReadModifyWrite ReadRead	Reads 0s, and replaces them with 1s. A back-to-back read is performed before changing address.
14	17,18	w to 1 Fast row	(R1W0) (R0Rx)	ReadModifyWrite ReadRead	Reads 1s, and replaces them with 0s. A back-to-back read is performed before changing address.

Table C-8. Description of SMarchCHKBcil Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
15	19-30	1 to w Fast row	(R0W1) (R0Rx) (R1W0)	ReadModifyWrite ReadRead ReadModifyWrite	This phase is composed of three operations. For all operations, the column address is constant. The first set of operations is a Read 0. followed by a Write 1. The second set of operations is to perform two back-to-back Read 0 from a different row (on the same column) while *ShadowWrite* is on. However, the result of the second read operation is not compared. The third set of operations is to Read a 1 from the current row and then restore its value to 0.
16	31,32	1	(RxW1) (R1Rx)	ReadModifyWrite ReadRead	Writes 1 through the first word. A back-to-back read is performed before changing address.
17	33	2 to w	(W1Rx)	Write	Writes 1 to all locations.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Table C-8. Description of SMarchCHKBcil Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
18	34-45	1 to w fast row	(R1W0) (R1Rx) (R0W1)	ReadModifyWrite ReadRead ReadModifyWrite	This phase is composed of three operations. For all operations, the column address is constant. The first set of operations is a Read 1 followed by a Write 0. The second set of operations is to perform two back-to-back Read 1 from a different row while *ShadowWrite* is on. However, the result of the second read operation is not compared. The third set of operations is to Read a 0 from the current row and then restore its value to 1.
19	46,47	1	(RxWmemcontents) (WmemcontentsRx)	ReadModifyWrite Write	Writes the memory content into the first word (default 0), and writes it back.
20	48	2 to w	Wmemcontents Rmemcontents	Write	Writes memory contents to the rest of the memory (default 0).

Where:

x^+	Algorithm phases performed during MemRst
x*	Algorithm phases performed during the PSRT start_to_pause sub-phase
x**	Algorithm phases performed during the PSRT pause_to_pause sub-phase
x***	Algorithm phases performed during the PSRT pause_to_end sub-phase

Detected Faults

The SMarchCHKBcil algorithm detects the failure modes indicated in Table C-1.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Test Time

The time required for the memory BIST controller to test your design using the SMarchCHKBcil algorithm is outlined in Table C-2.

Specification

To test SRAMs using the SMarchCHKBcil algorithm, specify Algorithm SMarchCHKBcil in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

SMarchCHKBvcd Algorithm

This algorithm is an enhanced SMarchCHKBcil test algorithm to detect data path shorts as well as enable the detection of voltage drop on cells in multi-port memories.

A few steps are added to the SMarchCHKBcil algorithm to make sure that the appropriate data combinations are applied to every cell. However, this algorithm only can be used with full parallel data access (serial interfacing is not supported).

Usage

The implementation is enhanced to remove operations that require reading the memory and writing the memory in the next cycle. These operations have caused timing closure issues. The algorithm makes the assumption that data inputs are physically laid out so that even-index inputs are interleaved with odd-index inputs when testing for shorts between bits of the internal memory data bus. The algorithm makes the same assumption for group write enable inputs.

Each phase of the algorithm is described for the case of a single bank. If more than one bank exists, the phase repeats for each bank. The address counter in a phase determines whether the banks are addressed in ascending or descending order.

The algorithm requires a special operation set. Additional operations perform the specialized test sequences. Table C-9 lists the operations that the algorithm requires. The library operation set SyncWRvcd is compatible with this algorithm. If you create a custom operation set, you must define all necessary operations.

Table C-9 shows the entire SMarchCHKBvcd algorithm. The codes in the Operations Used and Description columns map to Table C-10, which gives the corresponding operation names that are defined in the SyncWRvcd operation set.

Table C-9. Description of SMarchCHKBvcd Test Algorithm per Test Port

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
1*+	0		Idle	OP0	
2*	1-3	1 to C, Row 1 1 to C, Row 1	(W0R0Rx) (W1R1Rx) Idle	OP5 OP5 OP0	This phase is used to detect data path shorts. The same pattern is written to all words with no data mapping applied. Step 1 — (OP5) Write checkerboard to the data path and read it back for all words in the first row. Step 2 — (OP5) Write inverse checkerboard to the data path and read it back for all words in the first row.

Table C-9. Description of SMarchCHKBvcd Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
3*	4-12	1 to C, Row 1 1 to C, Row 1 1 to C, Row 1 1 to C, Row 1	(W0R0Rx)(W1R1Rx) (W0R0Rx)(W1R1Rx) (W0R0Rx)(W1R1 Rx) (W1R1Rx)(W0R1Rx) Idle	OP5,OP6 OP5,OP7 OP5,OP5 OP5,OP8 OP0	This phase is used to detect bit/group write enable faults. Repeat all groups of operations for all words in the first row. Step 1 — (OP5) Write and read 0s for the word. Step 2 * (OP6) Write 1s while Even Group Write Enables are on followed by reading data pattern 0101 ... 0101. Step 3 — (OP5) Write and read 0s for the word. Step 4 * (OP7) Write 1s while Odd Group Write Enables are on followed by reading data pattern 1010 ... 1010. Step 5 — (OP5) Write and read 0s for the word. Step 6 — (OP5) Write 1s while all Group Write Enables are on followed by reading 1s. Step 7 — (OP5) Write and read 1s for the word. Step 8 * (OP8) Attempt to write 0s while all Group Write Enables are off followed by reading 1s.

Note: Steps marked with * indicate that memories without the feature under test are disabled to avoid corruption of their content.

Table C-9. Description of SMarchCHKBvcd Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
3.6*	13-19	1 to C, Row 1	(W1Rx)	OP1	This phase is used to detect Read Enable stuck-active faults.
		1 to C, Row 2	(W0Rx)	OP1	
		1 to C, Row 1	(R1Rx)	OP2	Step 1 — (OP1) Write 1s to all words in the first row.
		1 to C, Row 2	(R1Rx)	OP9	
		1 to C, Row 2	(R0Rx)	OP2	Step 2 — (OP1) Write 0s to all words in the second row.
		1 to C, Row 1	(R0Rx)	OP9	
			Idle	OP0	Step 3 — (OP2) Read 1s from all words in the first row.
					Step 4 — (OP9) Attempt to read 0s from all words in the second row while Read Enable is off.
					Step 5 — (OP2) Read 0s from all words in the second row.
					Step 6 — (OP9) Attempt to read 1s from all words in the first row while Read Enable is off.

> **Note:** During Steps 4 and 6, memories without a ReadEnable input are disabled to avoid invalid failures. DataOutHoldWithInactiveReadEnable must be set to Off for memories with a ReadEnable input and whose outputs do not hold their values when the ReadEnable is inactive.

Table C-9. Description of SMarchCHKBvcd Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
3.7*	20-26	1 to C, Row 1 1 to C, Row 2 1 to C, Row 1 1 to C, Row 1 1 to C, Row 2 1 to C, Row 2	(R1Rx) (R0Rx) (R0W0) (R1Rx) (R1W1) (R0Rx) Idle	OP2 OP2 OP4 OP2 OP4 OP2 OP0	This phase is used to detect Chip Select stuck-active faults. Step 1 — (OP2) Read 1s from all words in the first row. Step 2 — (OP2) Read 0s from all words in the second row. Step 3 — (OP4) Attempt to read 1s and write 0s for all words in the first row while Chip Select is off. Step 4 — (OP2) Read 1s from all words in the first row. Step 5 — (OP4) Attempt to read 0s and write 1s for all words in the second row while Chip Select is off. Step 6 — (OP2) Read 0s from all words in the second row.
	Note: During Steps 3 and 5, memories without a Select input are disabled to avoid invalid failures. Prior to v2021.4, MemoryHoldWithInactiveSelect must be set to Off if a subset of the memory logical ports have a Select input.				
4*	27		Idle	OP0	
5*	28-29	1 to W Fast column	(RxW0) Idle	OP3 OP0	Write checkerboard background data.

Table C-9. Description of SMarchCHKBvcd Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
5.1	30-31	1 to W Fast row	(R0W1) (W0Rx)	OP10 OP12	This phase is only useful for multi-port memories. Step 1 — (OP10) Read checkerboard and write inverse checkerboard data. Step 2 — (OP12) Write checkerboard data restoring the original background and reading it back. During the Read portion of the operation, a write_column_address operation is performed on the adjacent column of inactive write port. The operation is not performed for memories without column address bits. During the Write portion of the operation, a read_column_address operation is performed on the adjacent column of inactive read port for memories with column address bits. The operation is performed on the adjacent row for memories without column address bits.
5.2	32	1 to W Fast row	(R0W1)	OP3	Read checkerboard background and replace it with inverse checkerboard data.

Table C-9. Description of SMarchCHKBvcd Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
5.3	33-34	1 to W Fast row	(W0Rx) (R0W0)	OP1 OP3	Step 1 — (OP1) Write checkerboard data and read it back. Step 2 — (OP3) Read checkerboard data and write it back before changing the address. The back-to-back Write operations occur at the end of the last Write operation on the current cell and the Write operation on the cell in the next row address.
5.5	35-36	1 to W Fast column	(R0W0) Idle	OP11 OP0	Read checkerboard background while ConcurrentWrite is on, and ConcurrentRead is off (Read is performed at the address specified by the test algorithm from all inactive R/W ports). Write checkerboard with ConcurrentRead on and ConcurrentWrite off.
			Optional Wait	None	Pause the test to perform the static retention test.
6**	37-38	1 to W Fast column	(R0Rx) Idle	OP2 OP0	Read checkerboard background data followed by a read operation whose data out is not compared.
6.1**	39-40	1 to W Fast column	(R0W1) Idle	OP3 OP0	Read checkerboard background while ConcurrentRead is off. Replace it with inverse checkerboard data while ConcurrentRead is on. ConcurrentWrite is off in both cases.

Table C-9. Description of SMarchCHKBvcd Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
6.2	41-42	1 to W Fast row	(R1W0) (W1Rx)	OP10 OP12	This phase is only useful for multi-port memories. Step 1 — (OP10) Read inverse checkerboard and write checkerboard data. Step 2 — (OP12) Write inverse checkerboard data restoring the original background and reading it back. During the Read portion of the operation, a write_column_address operation is performed on the adjacent column of inactive write port. The operation is not performed for memories without column address bits. During the Write portion of the operation, a read_column_address operation is performed on the adjacent column of inactive read port for memories with column address bits. The operation is performed on the adjacent row for memories without column address bits.
6.3	43	1 to W Fast row	(R1W0)	OP3	Read inverse checkerboard background and replace it with checkerboard data.

Table C-9. Description of SMarchCHKBvcd Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
6.4	44-45	1 to W Fast row	(W1Rx) (R1W1)	OP1 OP3	Step 1 — (OP1) Write inverse checkerboard data and read it back. Step 2 — (OP3) Read inverse checkerboard data and write it back before changing the address. The back-to-back Write operations occur at the end of the last Write operation on the current cell and the Write operation on the cell in the next row address.
6.5	46-47	1 to W Fast column	(R1W1) Idle	OP11 OP0	Read inverse checkerboard background while ConcurrentWrite is on, and ConcurrentRead is off (Read is performed at the address specified by the test algorithm from all inactive R/W ports). Write inverse checkerboard with ConcurrentRead on and ConcurrentWrite off.
			Optional Wait	None	Pause the test to perform the static retention test.
7***	48-49	1 to W Fast column	(R1Rx) Idle	OP2 OP0	Read inverse checkerboard background data followed by a read operation whose data out is not compared.
7.1	50-51	1 to W Fast column	(R1W0) Idle	OP3 OP0	Read inverse checkerboard data while ConcurrentRead is off. Replace it with checkerboard data while ConcurrentRead is on.
8	52		Idle	OP0	Perform multi-cycle initialization.

Table C-9. Description of SMarchCHKBvcd Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
10^+	53	1 to W Fast row	(W0Rx)	OP1	Write 0s to all words.
11	54-55	1 to W Fast row	(R0W1) (R1W1)	OP3 OP3	Step 1 — (OP3) Read 0 followed by write 1. Step 2 — (OP3) Read 1 and write it back.
12	56-57	1 to W Fast row	(R1W0) (R0W0)	OP3 OP3	Step 1 — (OP3) Read 1 followed by write 0. Step 2 — (OP3) Read 0 and write it back.
13	58-59	W to 1 Fast row	(R0W1) (R1W1)	OP3 OP3	Step 1 — (OP3) Read 0 followed by write 1. Step 2 — (OP3) Read 1 and write it back before decrementing the address.
14	60-61	W to 1 Fast row	(R1W0) (R0W0)	OP3 OP3	Step 1 — (OP3) Read 1 followed by write 0. Step 2 — (OP3) Read 0 and write it back before decrementing the address.
15	62-74	1 to W Fast row	(R0W1) (R0W0) (R1W0) Idle	OP3 OP3 OP3 OP0	Step 1 — (OP3) Read 0 followed by write 1. Step 2 — (OP3) Read 0 from adjacent row on the same column and write it back. Step 3 — (OP3) Read 1 from the current row and restore its value to 0.
17	75	1 to W Fast row	(W1Rx)	OP1	Write 1s to all words.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Table C-9. Description of SMarchCHKBvcd Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
18	76-88	1 to W Fast row	(R1W0) (R1W1) (R0W1) Idle	OP3 OP3 OP3 OP0	Step 1 — (OP3) Read 1 followed by write 0. Step 2 — (OP3) Read 1 from adjacent row on the same column and write it back. Step 3 — (OP3) Read 0 from the current row and restore its value to 1.
20	89	1 to W Fast row	(WmemcontentsRx)	OP1	Write the memory content to all words (default 0).

Where:

x^+	Algorithm phases performed during MemRst
x*	Algorithm phases performed during the PSRT start_to_pause sub-phase
x**	Algorithm phases performed during the PSRT pause_to_pause sub-phase
x***	Algorithm phases performed during the PSRT pause_to_end sub-phase

Table C-10 shows the mapping of operation names to the codes used in the Operation Used and Description columns of Table C-9.

Table C-10. Mapping of Operation Code to Operation Name

Code	Operation Name
OP0	NoOperation
OP1	Write
OP2	Read
OP3	ReadModifyWrite
OP4	ReadModifyWrite_WithSelectOff
OP5	WriteReadCompare
OP6	WriteReadCompare_EvenGWE_On
OP7	WriteReadCompare_OddGWE_On
OP8	WriteReadCompare_AllGWE_Off
OP9	ReadWithReadEnableOff

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Table C-10. Mapping of Operation Code to Operation Name (cont.)

Code	Operation Name
OP10	ReadModifyWrite_Column_ConcurrentWriteRead
OP11	ReadModifyWrite_Row_ConcurrentWriteRead
OP12	WriteRead_Column_ConcurrentReadWrite

Detected Faults

The SMarchCHKBvcd algorithm detects the failure modes indicated in Table C-1.

Test Time

The time required for the memory BIST controller to test your design using the SMarchCHKBvcd algorithm is outlined in Table C-2.

Specification

To test SRAMs using the SMarchCHKBvcd algorithm, specify Algorithm SMarchCHKBvcd in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

Usage Conditions

The OperationSet property of the memory TCD file must be the Siemens EDA library SyncWRvcd.

The SMarchCHKBvcd algorithm performs specialized tests on the chip select and read enable ports, if they are present. To use this algorithm, the memory data output value must be preserved when the chip select or read enable port is deasserted. If the memory data output value is not preserved in these cases, in the Memory wrapper of the memory TCD file, set the MemoryHoldWithInactiveSelect or DataOutHoldWithInactiveReadEnable properties to off, as appropriate for your memory.

LVMarchX Algorithm

The LVMarchX algorithm is a test algorithm that is available for loading into the memory controller to perform a March X algorithm. The March X algorithm is performed as follows:

1. Write background of D-data incrementing from address minimum to address maximum.

2. Read D-data and write $\overline{\text{D}}$-data incrementing from address minimum to address maximum.

3. Read $\overline{\text{D}}$-data and write D-data decrementing from address maximum to address minimum.

4. Read D-data decrementing from address maximum to address minimum.

Test Time

The time required for the memory BIST controller to test your design using the LVMarchX algorithm is outlined in Table C-2.

TestRegisterSetup

The test register setup describes the controller test register values to be initialized for the address and data prior to execution of the algorithm.

AddressRegisterA

The AddressRegisterA segments are configured to sequence the address counting with fast rows as follows:

$$Z \Leftarrow Y1 \Leftarrow X1$$

where:

- The row address segment X1 counts when instructed.

- The column address segment Y1 counts when instructed and a carry out from the X1 address segment is generated. An x1_carry_out is generated when:

  o The X1 address segment is incrementing and has reached the maximum OR.

  o The X1 address segment is decrementing and has reached the minimum.

- The bank address segment Z counts when instructed and a carry out from the Y1 address segment is generated. An y1_carry_out is generated when:

  o The Y1 address segment is incrementing and has reached the maximum AND a x1_carry_out is generated OR.

  o The Y1 address segment is decrementing and has reached the minimum AND a x1_carry_out is generated.

DataRegister

The logical data pattern D is loaded into both the write and expect data registers and is a word of all zeroes.

Algorithm Sequence

Table C-11 describes the LVMarchX algorithm sequence.

Table C-11. Description of LVMarchX Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast row	WD	Write	Write background of D-data.
1	1	-	-	min to max, fast row	RDW$\overline{\text{D}}$	ReadModifyWrite	Read D-data and Write data $\overline{\text{D}}$-data. After all addresses have been accessed branch to instruction 1 and repeat one time as follows with RepeatLoopA: • Repeat #1 - repeat instructions with inverted the address sequencing, write data sequencing, and expect data sequencing
2	1	Repeat #1	-	max to min, fast row	R$\overline{\text{D}}$WD	ReadModifyWrite	Read $\overline{\text{D}}$-data and Write D-data.
3	2	-	-	max to min, fast row	RD	Read	Read D-data.

Example Algorithm Wrapper

Figure C-1 illustrates the Algorithm wrapper for the example memory. Two equivalent algorithms (TessentMarchXFastX and TessentMarchXFastY), that have been optimized to eliminate redundant operations, are also available in the *lib/technology/memory_bist* directory of the tool tree. These algorithms run slightly faster, but diagnosis needs to be performed in two steps as explained in the "Diagnosis Considerations" section.

Figure C-1. LVMarchX Example Algorithm Wrapper

```
MemoryOperationsSpecification {
  Algorithm (LVMarchX){
    TestRegisterSetup {
      operation_set_select : Sync;
      AddressGenerator {
        AddressRegisterA {
          z_carry_in : y1_carry_out;
          y1_carry_in : x1_carry_out;
          x1_carry_in : none;
        }
      }
      DataGenerator {
        load_write_data  : all_zero;
        load_expect_data : all_zero;
      }
    }
    MicroProgram {
      Instruction (M0_W0) {
        operation_select : Write;
        AddressCommands {
          address_select : select_a;
          z_address : increment;
          x1_address : increment;
          y1_address : increment;
        }
        DataCommands {
          write_data : data_reg;
        }
        NextConditions {
          z_end_count : on;
          x1_end_count : on;
          y1_end_count : on;
        }
      }
      Instruction (M1_R0_W1) {
        operation_select : ReadModifyWrite;
        AddressCommands {
          address_select : select_a;
          z_address  : increment;
          x1_address : increment;
          y1_address : increment;
          inhibit_last_address_count : on;
        }
        DataCommands {
          expect_data : data_reg;
          write_data : inverse_data_reg;
        }
        NextConditions {
          z_end_count : on;
          x1_end_count : on;
          y1_end_count : on;
          RepeatLoopA {
            branch_to_instruction : M1_R0_W1;
            Repeat1 {
              enable : on;
```

```
                    write_data_sequence : inverse;
                    expect_data_sequence : inverse;
                    address_sequence : inverse;
                }
            }
        }
    }
    Instruction (M3_R0) {
        operation_select : Read;
        AddressCommands {
            address_select : select_a;
            z_address : decrement;
            x1_address : decrement;
            y1_address : decrement;
            inhibit_last_address_count : on;
        }
        DataCommands {
            expect_data : data_reg;
        }
        NextConditions {
            z_end_count : on;
            x1_end_count : on;
            y1_end_count : on;
        }
    }
}
```

Fault Coverage

The faults detected by the LVMarchX algorithm are identified in Table C-1.

Specification

To test SRAMs using the LVMarchX algorithm, specify Algorithm LVMarchX in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

Usage Conditions

The following usage conditions apply to the LVMarchX algorithm:

- Operations named Write, ReadModifyWrite, and Read must exist or be mapped to another operation in OperationSet in the memory TCD file.

- For soft programmable controller usage, the DftSpecification/MemoryBist/Controller/ AlgorithmResourceOptions/soft_instruction_count property must be greater than or equal to three.

LVMarchY Algorithm

The LVMarchY algorithm is a test algorithm that is available for loading into the memory controller to perform a March Y algorithm. The March Y algorithm is performed as follows:

1. Write background of D-data incrementing from address minimum to address maximum.

2. Read D-data, write $\overline{D}$-data, and read $\overline{D}$-data incrementing from address minimum to address maximum.

3. Read $\overline{D}$-data, write D-data, and read D-data decrementing from address maximum to address minimum.

4. Read D-data decrementing from address maximum to address minimum.

Test Time

The time required for the memory BIST controller to test your design using the LVMarchY algorithm is outlined in Table C-2.

TestRegisterSetup

The test register setup describes the controller test register values to be initialized for the address and data prior to execution of the algorithm.

AddressRegisterA

The AddressRegisterA segments are configured to sequence the address counting with fast rows as follows:

$$Z \Leftarrow Y1 \Leftarrow X1$$

where:

- The row address segment X1 counts when instructed.

- The column address segment Y1 counts when instructed and a carry out from the X1 address segment is generated. An x1_carry_out is generated when:

  o The X1 address segment is incrementing and has reached the maximum OR

  o The X1 address segment is decrementing and has reached the minimum

- The bank address segment Z counts when instructed and a carry out from the Y1 address segment is generated. An y1_carry_out is generated when:

  o The Y1 address segment is incrementing and has reached the maximum AND a x1_carry_out is generated OR.

  o The Y1 address segment is decrementing and has reached the minimum AND a x1_carry_out is generated.

DataRegister

The logical data pattern D is loaded into both the write and expect data registers and is a word of all zeroes.

Algorithm Sequence

Table C-12 describes the LVMarchY algorithm sequence.

Table C-12. Description of LVMarchY Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast row	WD	Write	Write background of D-data.
1	1	-	-	-	RDW$\overline{\text{D}}$	ReadModifyWrite	Read D-data and Write $\overline{\text{D}}$-data.
	2	-	-	min to max, fast row	R$\overline{\text{D}}$	Read	Read $\overline{\text{D}}$-data. Branch back to Instruction 1 until all addresses are accessed. After all addresses have been accessed branch to instruction 1 and repeat one time as follows with RepeatLoopA: • Repeat #1 - repeat instructions with inverted the address sequencing, write data sequencing, and expect data sequencing
2	1	Repeat #1	-	-	R$\overline{\text{D}}$WD	ReadModifyWrite	Read $\overline{\text{D}}$-data and Write D-data.
	2	Repeat #1	-	max to min, fast row	RD	Read	Read D-data. Branch back to Instruction 1 until all addresses are accessed.
3	3	-	-	max to min, fast row	RD	Read	Read D-data.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Example Algorithm Wrapper

Figure C-2 illustrates the Algorithm wrapper for the example memory. Two equivalent algorithms (TessentMarchYFastX and TessentMarchYFastY), that have been optimized to eliminate redundant operations, are also available in the *lib/technology/memory_bist* directory of the tool tree. These algorithms run slightly faster, but diagnosis needs to be performed in two steps as explained in the "Diagnosis Considerations" section.

Figure C-2. LVMarchY Example Algorithm Wrapper

```
MemoryOperationsSpecification {
  Algorithm (LVMarchY) {
   TestRegisterSetup {
      operation_set_select : Sync;
      AddressGenerator {
         AddressRegisterA {
            z_carry_in : y1_carry_out;
            y1_carry_in : x1_carry_out;
            x1_carry_in : none;
         }
      }
      DataGenerator {
         load_write_data  : all_zero;
         load_expect_data : all_zero;
      }
   }
  MicroProgram {
      Instruction (M0_W0) {
         operation_select : Write;
         AddressCommands {
            address_select : select_a;
            z_address : increment;
            x1_address : increment;
            y1_address : increment;
         }
         DataCommands {
            write_data : data_reg;
         }
         NextConditions {
            z_end_count : on;
            x1_end_count : on;
            y1_end_count : on;
         }
      }
      Instruction (M1_R0_W1) {
         operation_select : ReadModifyWrite;
         AddressCommands {
            address_select : select_a;
         }
         DataCommands {
            expect_data : data_reg;
            write_data : inverse_data_reg;
         }
         NextConditions {
         }
      }
      Instruction (M1_R1) {
         operation_select : Read;
         AddressCommands {
            address_select : select_a;
            z_address : increment;
            x1_address : increment;
            y1_address : increment;
            inhibit_last_address_count : on;
         }
```

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

```
                    DataCommands {
                        expect_data : inverse_data_reg;
                    }
                    branch_to_instruction : M1_R0_W1;
                    NextConditions {
                        z_end_count : on;
                        x1_end_count: on;
                        y1_end_count : on;
                        RepeatLoopA {
                            branch_to_instruction : M1_R0_W1;
                            Repeat1 {
                                enable : on;
                                write_data_sequence : inverse;
                                expect_data_sequence : inverse;
                                address_sequence : inverse;
                            }
                        }
                    }
                }
                Instruction (M3_R0) {
                    operation_select : Read;
                    AddressCommands {
                        address_select : select_a;
                        z_address : decrement;
                        x1_address : decrement;
                        y1_address : decrement;
                        inhibit_last_address_count : on;
                    }
                    DataCommands {
                        expect_data : data_reg;
                    }
                    NextConditions {
                        z_end_count : on;
                        x1_end_count : on;
                        y1_end_count : on;
                    }
                }
            }
        }
    }
```

Fault Coverage

The faults detected by the LVMarchY algorithm are identified in Table C-1.

Specification

To test SRAMs using the LVMarchY algorithm, specify Algorithm LVMarchY in the memory
TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in
the DefaultsSpecification.

Usage Conditions

The following usage conditions apply to the LVMarchY algorithm:

- Operations named Write, ReadModifyWrite, and Read must exist or be mapped to another operation in OperationSet specified in the memory TCD file.

- For soft programmable controller usage, the DftSpecification/MemoryBist/Controller/ AlgorithmResourceOptions/soft_instruction_count property must be greater than or equal to four.

LVMarchCMinus Algorithm

The LVMarchCMinus algorithm is a test algorithm that is available for loading into the memory controller to perform a March C- algorithm. The March C- algorithm is performed as follows:

1. Write background of D-data incrementing from address minimum to address maximum.

2. Read D-data and write $\overline{D}$-data incrementing from address minimum to address maximum.

3. Read $\overline{D}$-data and write D-data incrementing from address minimum to address maximum.

4. Read D-data and write $\overline{D}$-data decrementing from address maximum to address minimum.

5. Read $\overline{D}$-data and write D-data decrementing from address maximum to address minimum.

6. Read D-data decrementing from address maximum to address minimum.

Test Time

The time required for the memory BIST controller to test your design using the LVMarchCMinus algorithm is outlined in Table C-2.

TestRegisterSetup

The test register setup describes the controller test register values to be initialized for the address and data prior to execution of the algorithm.

AddressRegisterA

The AddressRegisterA segments are configured to sequence the address counting with fast rows as follows:

$$Z \Leftarrow Y1 \Leftarrow X1$$

where:

- The row address segment X1 counts when instructed.

- The column address segment Y1 counts when instructed and a carry out from the X1 address segment is generated. An x1_carry_out is generated when:

  o The X1 address segment is incrementing and has reached the maximum OR.

  o The X1 address segment is decrementing and has reached the minimum.

- The bank address segment Z counts when instructed and a carry out from the Y1 address segment is generated. An y1_carry_out is generated when:

  o The Y1 address segment is incrementing and has reached the maximum AND a x1_carry_out is generated OR.

  o The Y1 address segment is decrementing and has reached the minimum AND a x1_carry_out is generated.

DataRegister

The logical data pattern D is loaded into both the write and expect data registers and is a word of all zeroes.

Algorithm Sequence

Table C-13 describes the LVMarchCMinus algorithm sequence.

Table C-13. Description of LVMarchCMinus Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast row	WD	Write	Write background of D-data.

Table C-13. Description of LVMarchCMinus Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
1	1	-	-	min to max, fast row	RDW$\overline{\text{D}}$	ReadModifyWrite	Read D-data and Write $\overline{\text{D}}$-data. After all addresses have been accessed branch to instruction 1 and repeat three times as follows with RepeatLoopA: • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing • Repeat #2 - repeat instructions with inverted the address sequencing • Repeat #3 - repeat instructions with inverted the address sequencing, write data sequencing, and expect data sequencing
2	1	Repeat #1	-	min to max, fast row	R$\overline{\text{D}}$WD	ReadModifyWrite	Read $\overline{\text{D}}$-data and Write D-data.
3	1	Repeat #2	-	max to min, fast row	RDW$\overline{\text{D}}$	ReadModifyWrite	Read D-data and Write $\overline{\text{D}}$-data.
4	1	Repeat #3	-	max to min, fast row	R$\overline{\text{D}}$WD	ReadModifyWrite	Read $\overline{\text{D}}$-data and Write D-data.
5	2	-	-	max to min, fast row	RD	Read	Read D-data.

Example Algorithm Wrapper

Figure C-3 illustrates the Algorithm wrapper for the example memory. Two equivalent algorithms (TessentMarchCMinusFastX and TessentMarchCMinusFastY), that have been optimized to eliminate redundant operations, are also available in the *lib/technology/ memory_bist* directory of the tool tree. These algorithms run slightly faster, but diagnosis needs to be performed in two steps as explained in the "Diagnosis Considerations" section.

Figure C-3. LVMarchCMinus Example Algorithm Wrapper

```
MemoryOperationsSpecification {
  Algorithm (LVMarchCMinus){
   TestRegisterSetup {
       operation_set_select : Sync;
       AddressGenerator {
          AddressRegisterA {
             z_carry_in : y1_carry_out;
             y1_carry_in : x1_carry_out;
             x1_carry_in : none;
          }
       }
       DataGenerator {
          load_write_data  : all_zero;
          load_expect_data : all_zero;
       }
   }
   MicroProgram {
       Instruction (M0_W0) {
          operation_select : Write;
          AddressCommands {
             address_select : select_a;
             z_address : increment;
             x1_address : increment;
             y1_address : increment;
          }
          DataCommands {
             write_data : data_reg;
          }
          NextConditions {
             z_end_count : on;
             x1_end_count : on;
             y1_end_count : on;
          }
       }
       Instruction (M1_R0_W1) {
          operation_select : Readmodifywrite;
          AddressCommands {
             address_select : select_a;
             z_address : increment;
             x1_address : increment;
             y1_address : increment;
          }
          DataCommands {
             expect_data : data_reg;
             write_data : inverse_data_reg;
          }
          NextConditions {
             z_end_count : on;
             x1_end_count : on;
             y1_end_count : on;
             RepeatLoopA {
                branch_to_instruction : M1_R0_W1;
                Repeat1 {
                   enable : on;
                   write_data_sequence : inverse;
```

```
                            expect_data_sequence : inverse;
                            inhibit_last_address_count : on;
                        }
                        Repeat2 {
                            enable : on;
                            address_sequence : inverse;
                        }
                        Repeat3 {
                            enable : on;
                            address_sequence : inverse;
                            expect_data_sequence : inverse;
                            write_data_sequence : inverse;
                        }
                    }
                }
            }
            Instruction (M5_R0) {
                operation_select : Read;
                AddressCommands {
                    address_select : select_a;
                    z_address : decrement;
                    x1_address : decrement;
                    y1_address : decrement;
                    inhibit_last_address_count : on;
                }
                DataCommands {
                    expect_data : data_reg;
                }
                NextConditions {
                    z_end_count : on;
                    x1_end_count : on;
                    y1_end_count : on;
                }
            }
        }
    }
}
```

Fault Coverage

The faults detected by the LVMarchCMinus algorithm are identified in Table C-1.

Specification

To test SRAMs using the LVMarchCMinus algorithm, specify Algorithm LVMarchCMinus in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

Usage Conditions

The following usage conditions apply to the LVMarchCMinus algorithm:

- Operations named Write, ReadModifyWrite, and Read must exist or be mapped to another operation in OperationSet of the memory TCD file.

- For soft programmable controller usage, the DftSpecification/MemoryBist/Controller/ AlgorithmResourceOptions/soft_instruction_count property must be greater than or equal to three.

LVMarchLA Algorithm

The LVMarchLA algorithm is a test algorithm that is available for loading into the memory controller to perform a March LA algorithm. The March LA algorithm is performed as follows:

1. Write background of D-data incrementing from address minimum to address maximum.

2. Read D-data, write $\overline{D}$-data, write D-data, write $\overline{D}$-data, and read $\overline{D}$-data incrementing from address minimum to address maximum.

3. Read $\overline{D}$-data, write D-data, write $\overline{D}$-data, write D-data, and read D-data incrementing from address minimum to address maximum.

4. Read D-data, write $\overline{D}$-data, write D-data, write $\overline{D}$-data, and read $\overline{D}$-data decrementing from address maximum to address minimum.

5. Read $\overline{D}$-data, write D-data, write $\overline{D}$-data, write D-data, and read D-data decrementing from address maximum to address minimum.

6. Read D-data decrementing from address maximum to address minimum.

Test Time

The time required for the memory BIST controller to test your design using the LVMarchLA algorithm is outlined in Table C-2.

TestRegisterSetup

The test register setup describes the controller test register values to be initialized for the address and data prior to execution of the algorithm.

AddressRegisterA

The AddressRegisterA segments are configured to sequence the address counting with fast rows as follows:

$$Z \Leftarrow Y1 \Leftarrow X1$$

where:

- The row address segment X1 counts when instructed.

- The column address segment Y1 counts when instructed and a carry out from the X1 address segment is generated. An x1_carry_out is generated when:

  o The X1 address segment is incrementing and has reached the maximum OR.

  o The X1 address segment is decrementing and has reached the minimum.

- The bank address segment Z counts when instructed and a carry out from the Y1 address segment is generated. An y1_carry_out is generated when:

  o The Y1 address segment is incrementing and has reached the maximum AND a x1_carry_out is generated OR.

  o The Y1 address segment is decrementing and has reached the minimum AND a x1_carry_out is generated.

DataRegister

The logical data pattern D is loaded into both the write and expect data registers and is a word of all zeroes.

Algorithm Sequence

Table C-14 describes the LVMarchLA algorithm sequence.

Table C-14. Description of LVMarchLA Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast row	WD	Write	Write background of D-data.

Table C-14. Description of LVMarchLA Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
1	1	-	-	-	$R D W \overline{D}$	ReadModifyWrite	Read D-data and Write $\overline{D}$-data.
	2	-	-		$W \overline{D}$	Write	Write $\overline{D}$-data.
	3	-	-	min to max, fast row	$W \overline{D} R \overline{D}$	WriteRead	Write $\overline{D}$-data and Read $\overline{D}$-data. Branch back to Instruction 1 until all addresses are accessed. After all addresses have been accessed branch to instruction 1 and repeat three times as follows with RepeatLoopA: • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing • Repeat #2 - repeat instructions with inverted the address sequencing • Repeat #3 - repeat instructions with inverted the address sequencing, write data sequencing, and expect data sequencing
2	1	Repeat #1	-	-	$R \overline{D} W D$	ReadModifyWrite	Read $\overline{D}$-data and Write D-data.
	2	Repeat #1	-		$W D$	Write	Write D-data.
	3	Repeat #1	-	min to max, fast row	$W D R D$	WriteRead	Write D-data and Read data D-data. Branch back to Instruction 1 until all addresses are accessed.

Table C-14. Description of LVMarchLA Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
3	1	Repeat #2	-	-	RDW$\overline{D}$	ReadModifyWrite	Read D-data and Write $\overline{D}$-data.
	2	Repeat #2	-		WD	Write	Write D-data.
	3	Repeat #2	-	max to min, fast row	W$\overline{D}$R$\overline{D}$	WriteRead	Write $\overline{D}$-data and Read data $\overline{D}$-data. Branch back to Instruction 1 until all addresses are accessed.
4	1	Repeat #3	-	-	R$\overline{D}$WD	ReadModifyWrite	Read $\overline{D}$-data and Write data D-data.
	2	Repeat #3	-	-	WD	Write	Write D-data.
	3	Repeat #3	-	max to min, fast row	WDRD	WriteRead	Write D-data and Read data D-data. Branch back to Instruction 1 until all addresses are accessed.
5	4	-	-	max to min, fast row	RD	Read	Read D-data.

Example Algorithm Wrapper

Figure C-4 illustrates the Algorithm wrapper for the example memory. Two equivalent algorithms (TessentMarchLAFastX and TessentMarchLAFastY), that have been optimized to eliminate redundant operations, are also available in the *lib/technology/memory_bist* directory of the tool tree. These algorithms run slightly faster, but diagnosis needs to be performed in two steps as explained in the "Diagnosis Considerations" section.

Figure C-4. LVMarchLA Example Algorithm Wrapper

```
MemoryOperationsSpecification {
  Algorithm (LVMarchLA){
   TestRegisterSetup {
      operation_set_select : Sync;
      AddressGenerator {
         AddressRegisterA {
            z_carry_in : y1_carry_out;
            y1_carry_in : x1_carry_out;
            x1_carry_in : none;
         }
      }
      DataGenerator {
         load_write_data  : all_zero;
         load_expect_data : all_zero;
      }
   }
  MicroProgram {
      Instruction (M0_W0) {
         operation_select : Write;
         AddressCommands {
            address_select : select_a;
            z_address : increment;
            x1_address : increment;
            y1_address : increment;
         }
         DataCommands {
            write_data : data_reg;
         }
         NextConditions {
            z_end_count : on;
            x1_end_count : on;
            y1_end_count : on;
         }
      }
      Instruction (M1_R0_W1) {
         operation_select : ReadModifyWrite;
         AddressCommands {
            address_select : select_a;
         }
         DataCommands {
            expect_data : data_reg;
            write_data : inverse_data_reg;
         }
         NextConditions {
         }
      }
      Instruction (M1_W0) {
         operation_select : Write;
         AddressCommands {
            address_select : select_a;
         }
         DataCommands {
            write_data : data_reg;
         }
         NextConditions {
```

```
            }
        }
        Instruction (M1_W1_R1) {
            operation_select : Write_read_operation;
            AddressCommands {
                address_select : select_a;
                z_address : increment;
                x1_address : increment;
                y1_address : increment;
            }
            DataCommands {
                write_data : inverse_data_reg;
                expect_data : inverse_data_reg;
            }
            branch_to_instruction : M1_R0_W1;
            NextConditions {
                z_end_count : on;
                x1_end_count : on;
                y1_end_count : on;
                RepeatLoopA {
                    branch_to_instruction : M1_R0_W1;
                    Repeat1 {
                        enable : on;
                        write_data_sequence : inverse;
                        expect_data_sequence : inverse;
                        inhibit_last_address_count : on;
                    }
                    Repeat2 {
                        enable : on;
                        address_sequence : inverse;
                    }
                    Repeat3 {
                        enable : on;
                        address_sequence : inverse;
                        write_data_sequence : inverse;
                        expect_data_sequence : inverse;
                    }
                }
            }
        }
        Instruction (M5_R0) {
            operation_select : Read;
            AddressCommands {
                address_select : select_a;
                z_address : decrement;
                x1_address : decrement;
                y1_address : decrement;
                inhibit_last_address_count : on;
            }
            DataCommands {
                expect_data : data_reg;
            }
            NextConditions {
                z_end_count : on;
                x1_end_count : on;
                y1_end_count : on;
            }
        }
```

```
      }
    }
  }
}
```

Fault Coverage

The faults detected by the LVMarchLA algorithm are identified in Table C-1.

Specification

To test SRAMs using the LVMarchLA algorithm, specify Algorithm LVMarchLA in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

Usage Conditions

The following usage conditions apply to the LVMarchLA algorithm:

- Operations named Write, ReadModifyWrite, WriteRead, and Read must exist or be mapped to another operation in OperationSet of the memory TCD file.

- For soft programmable controller usage, the DftSpecification/MemoryBist/Controller/ AlgorithmResourceOptions/soft_instruction_count property must be greater than or equal to five.

LVRowBar Algorithm

The LVRowBar algorithm is a test algorithm that is available for loading into the memory controller to perform a row bar algorithm. The row bar algorithm is performed as follows:

1. Write background of D-data to even columns and $\overline{D}$-data to odd columns incrementing from address minimum to address maximum for a single bank.

2. Read D-data from even columns and $\overline{D}$-data from odd columns incrementing from address minimum to address maximum for a single bank.

3. Re-run 1 and 2, incrementing the bank address from minimum to maximum.

4. Repeat 1 to 3 with inverted data.

Test Time

The time required for the memory BIST controller to test your design using the LVRowBar algorithm is outlined in Table C-2.

TestRegisterSetup

The test register setup describes the controller test register values to be initialized for the address and data prior to execution of the algorithm.

AddressRegisterA

The AddressRegisterA segments are configured to sequence the address counting with fast columns as follows:

$$Z \qquad X1 \Leftarrow Y1$$

where:

- The column address segment Y1 counts when instructed.

- The row address segment X1 counts when instructed and a carry out from the Y1 address segment is generated. An y1_carry_out is generated when:

    o The Y1 address segment is incrementing and has reached the maximum OR.

    o The Y1 address segment is decrementing and has reached the minimum.

- The bank address segment Z counts when instructed.

DataRegister

The logical data pattern D is loaded into both the write and expect data registers and is a word of all zeroes.

The logical data pattern applied is inverted with each column. That is the logical data pattern applied is D for even columns and $\overline{D}$ for odd columns.

Algorithm Sequence

Table C-15 describes the LVRowBar algorithm sequence.

Table C-15. Description of LVRowBar Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast column	WD_e, $\overline{D}$_o	Write	Write background of D-data to even column addresses and $\overline{D}$-data to odd column addresses.
1	1	-	-	max to min, fast column	RD_e, $\overline{D}$_o	Read	Read background of D-data from even column addresses and $\overline{D}$-data from odd column addresses.

Table C-15. Description of LVRowBar Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
2	2	-	-	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses. After all bank addresses have been accessed branch to instruction 0 and repeat one time as follows with RepeatLoopB: • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing
3	0	-	Repeat #1	min to max, fast column	$W\overline{D}$_e, D_o	Write	Write background of $\overline{D}$-data to even column addresses and D-data to odd column addresses.
4	1	-	Repeat #1	max to min, fast column	$R\overline{D}$_e, D_o	Read	Read background of $\overline{D}$-data from even column addresses and D-data from odd column addresses.
5	2	-	Repeat #1	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses.

Example Algorithm Wrapper

Figure C-5 illustrates the Algorithm wrapper for the example memory. Two equivalent algorithms (TessentRowBarFastX and TessentRowBarFastY), that have been optimized to eliminate redundant operations, are also available in the *lib/technology/memory_bist* directory of the tool tree. These algorithms run slightly faster, but diagnosis needs to be performed in two steps as explained in the "Diagnosis Considerations" section.

Figure C-5. LVRowBar Example Algorithm Wrapper

```
MemoryOperationsSpecification {
  Algorithm (LVRowBar) {
   TestRegisterSetup {
       operation_set_select : Sync;
       AddressGenerator {
          AddressRegisterA {
              z_carry_in : none;
              x1_carry_in : y1_carry_out;
              y1_carry_in : none;
          }
       }
       DataGenerator {
          InvertDataWithColumnBit : c[0];
       }
   }
   MicroProgram {
       Instruction (W0) {
          operation_select : Write;
          AddressCommands {
              address_select : select_a;
              x1_address : increment;
              y1_address : increment;
          }
          DataCommands {
              write_data : zero;
          }
          NextConditions {
              x1_end_count : on;
              y1_end_count : on;
          }
       }
       Instruction (R0) {
          operation_select : Read;
          AddressCommands {
              address_select : select_a;
              x1_address : increment;
              y1_address : increment;
          }
          DataCommands {
              expect_data : zero;
          }
          NextConditions {
              x1_end_count : on;
              y1_end_count : on;
          }
       }
       Instruction (GOTO_NEXT_BANKADDRESS) {
          operation_select : NoOperation;
          AddressCommands {
              address_select : select_a;
              z_address : increment;
          }
          branch_to_instruction : W0;
          NextConditions {
              z_end_count : on;
```

```
                    RepeatLoopB {
                       branch_to_instruction: W0;
                       Repeat1 {
                          enable : on;
                          write_data_sequence: inverse;
                          expect_data_sequence: inverse;
                       }
                    }
                 }
              }
           }
        }
     }
```

Fault Coverage

The faults detected by the LVRowBar algorithm are identified in Table C-1. Stuck-at faults are detected correctly when there are no address decoder faults.

Specification

To test SRAMs using the LVRowBar algorithm, specify Algorithm LVRowBar in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

Usage Conditions

The following usage conditions apply to the LVRowBar algorithm:

- Operations named Write and Read must exist or be mapped to another operation in OperationSet of the memory TCD file.

- For soft programmable controller usage, the DftSpecification/MemoryBist/Controller/AlgorithmResourceOptions/soft_instruction_count property must be greater than or equal to three.

LVColumnBar Algorithm

The LVColumnBar algorithm is a test algorithm that is available for loading into the memory controller to perform a column bar algorithm. The column bar algorithm is performed as follows:

1. Write background of D-data to even rows and $\overline{D}$-data to odd rows incrementing from address minimum to address maximum for a single bank.

2. Read D-data from even rows and $\overline{D}$-data from odd rows incrementing from address minimum to address maximum for a single bank.

3. Re-run 1 and 2 incrementing the bank address from minimum to maximum.

4. Repeat 1 to 3 with inverted data.

Test Time

The time required for the memory BIST controller to test your design using the LVColumnBar algorithm is outlined in Table C-2.

TestRegisterSetup

The test register setup describes the controller test register values to be initialized for the address and data prior to execution of the algorithm.

AddressRegisterA

The AddressRegisterA segments are configured to sequence the address counting with fast rows as follows:

$$Z \qquad Y1 \Leftarrow X1$$

where:

- The row address segment X1 counts when instructed.

- The column address segment Y1 counts when instructed and a carry out from the X1 address segment is generated. An x1_carry_out is generated when:

  o The X1 address segment is incrementing and has reached the maximum OR.

  o The X1 address segment is decrementing and has reached the minimum.

- The bank address segment Z counts when instructed.

DataRegister

The logical data pattern D is loaded into both the write and expect data registers and is a word of all zeroes.

The logical data pattern applied is inverted with each row. That is the logical data pattern applied is D for even rows and $\overline{D}$ for odd rows.

Algorithm Sequence

Table C-16 describes the LVColumnBar algorithm sequence.

Table C-16. Description of LVColumnBar Algorithm

Phase	Instruction #	RepeatLoopA	RepeatLoopB	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast row	WD_e, $\overline{D}$_o	Write	Write background of D-data to even row addresses and $\overline{D}$-data to odd row addresses.
1	1	-	-	max to min, fast row	RD_e, $\overline{D}$_o	Read	Read background of D-data from even row addresses and $\overline{D}$-data from odd row addresses.
2	2	-	-	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses. After all bank addresses have been accessed branch to instruction 0 and repeat one time as follows with RepeatLoopB: • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing
3	0	-	Repeat #1	min to max, fast row	W$\overline{D}$_e, D_o	Write	Write background of $\overline{D}$-data to even row addresses and D-data to odd row addresses.
4	1	-	Repeat #1	max to min, fast row	R$\overline{D}$_e, D_o	Read	Read background of $\overline{D}$-data from even row addresses and D-data from odd row addresses.
5	2	-	Repeat #1	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses.

Example Algorithm Wrapper

illustrates the Algorithm wrapper for the example memory. Two equivalent algorithms (TessentColumnBarFastX and TessentColumnBarFastY), that have been optimized to eliminate redundant operations, are also available in the *lib/technology/memory_bist*

directory of the tool tree. These algorithms run slightly faster, but diagnosis needs to be performed in two steps as explained in the "Diagnosis Considerations" section.

Figure C-6. LVColumnBar Example Algorithm Wrapper

```
MemoryOperationsSpecification {
  Algorithm (LVColumnBar) {
   TestRegisterSetup {
      operation_set_select : Sync;
      AddressGenerator {
         AddressRegisterA {
            z_carry_in : none;
            y1_carry_in : x1_carry_out;
            x1_carry_in : none;
         }
      }
      DataGenerator {
         InvertDataWithRowBit : r[0];
      }
   }
   MicroProgram {
      Instruction (W0) {
         operation_select : Write;
         AddressCommands {
            address_select : select_a;
            x1_address : increment;
            y1_address : increment;
         }
         DataCommands {
            write_data : zero;
         }
         NextConditions {
            x1_end_count : on;
            y1_end_count : on;
         }
      }
      Instruction (R0) {
         operation_select : Read;
         AddressCommands {
            address_select : select_a;
            x1_address : increment;
            y1_address : increment;
         }
         DataCommands {
            expect_data : zero;
         }
         NextConditions {
            x1_end_count : on;
            y1_end_count : on;
         }
      }
      Instruction (GOTO_NEXT_BANKADDRESS) {
         operation_select : Nooperation;
         AddressCommands {
            address_select : select_a;
            z_address : increment;
         }
         branch_to_instruction : W0;
         NextConditions {
            z_end_count : on;
```

```
                   RepeatLoopB {
                      branch_to_instruction: W0;
                      Repeat1 {
                         enable : on;
                         write_data_sequence:  inverse;
                         expect_data_sequence: inverse;
                      }
                   }
                }
             }
          }
       }
    }
```

Fault Coverage

The faults detected by the LVColumnBar algorithm are identified in Table C-1. Stuck-at faults are detected correctly when there are no address decoder faults.

Specification

To test SRAMs using the LVColumnBar algorithm, specify Algorithm LVColumnBar in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

Usage Conditions

The following usage conditions apply to the LVColumnBar algorithm:

- Operations named Write and Read must exist or be mapped to another operation in OperationSet of the memory TCD file.

- For soft programmable controller usage, the DftSpecification/MemoryBist/Controller/ AlgorithmResourceOptions/soft_instruction_count property must be greater than or equal to three.

LVGalPat Algorithm

The LVGalPat algorithm is a test algorithm that is available for loading into the memory controller to perform a galloping pattern algorithm. The LVGalPat algorithm is performed as follows:

1. Write background of D-data from address minimum to address maximum for a single bank.

2. Write $\overline{D}$-data to the "home" cell addressed by AddressRegisterA.

3. Read D-data from all "away" cells in the same bank addressed by AddressRegisterB but following each read of the "away" cell read $\overline{D}$-data from the "home" cell.

4. Write D-data at the "home" cell to restore the data background.

5. Increment the "home" cell addressed by AddressRegisterA and re-run steps 2 to 5 until every cell in the bank has been a "home" cell.

6. Increment the bank address from minimum to maximum and re-run steps 1 to 6.

7. Repeat steps 1 to 6 with inverted data.

Test Time

The time required for the memory BIST controller to test your design using the LVGalPat algorithm is outlined in Table C-2.

TestRegisterSetup

The test register setup describes the controller test register values to be initialized for the address and data prior to execution of the algorithm.

AddressRegisterA

The AddressRegisterA segments are configured to sequence the address counting with fast rows as follows:

$$Z \qquad Y1 \Leftarrow X1$$

where:

- The row address segment X1 counts when instructed.

- The column address segment Y1 counts when instructed and a carry out from the X1 address segment is generated. An x1_carry_out is generated when:

  o The X1 address segment is incrementing and has reached the maximum OR.

  o The X1 address segment is decrementing and has reached the minimum.

- The bank address segment Z counts when instructed.

AddressRegisterB

The AddressRegisterB segments are configured to identical to AddressRegisterA.

DataRegister

The logical data pattern D is loaded into both the write and expect data registers and is a word of all zeroes.

The logical data pattern applied is inverted with each column. That is, the logical data pattern applied is D for even columns and D for odd columns.

Algorithm Sequence

Table C-17 describes the LVGalPat algorithm sequence.

Table C-17. Description of LVGalPat Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast row	WD	Write	Write background of D-data.
1	1	-	-	-	W$\overline{\text{D}}$	Write	Write $\overline{\text{D}}$-data to location addressed by AddressRegisterA.
2	2	-	-	-	R$\overline{\text{D}}$	Read	Read $\overline{\text{D}}$-data to location addressed by AddressRegisterA.
	3	-	-	AddressRegisterB min to max, fast row	RD	Read	Read D-data from all addresses using AddressRegisterB. When AddressRegisterB is equivalent to AddressRegisterA the read D-data is expected. Branch back to Instruction 2 and repeat for all AddressRegisterB row and column addresses.
	4	-	-	AddressRegisterA min to max, fast row	R$\overline{\text{D}}$WD	ReadModifyWrite	Read $\overline{\text{D}}$-data and write D-data at the location addressed by AddressRegisterA. Branch back to Instruction 2 and repeat for all AddressRegisterA row and column addresses.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Table C-17. Description of LVGalPat Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
3	5	-	-	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses. After all bank addresses have been accessed branch to instruction 0 and repeat one time as follows with RepeatLoopB: Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing.
0	0	-	Repeat #1	min to max, fast row	$W\overline{D}$	Write	Write background of D-data.
1	1	-	Repeat #1	-	WD	Write	Write $\overline{D}$-data to location addressed by AddressRegisterA.

Table C-17. Description of LVGalPat Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
2	2	-	Repeat #1	-	RD	Read	Read $\overline{D}$-data to location addressed by AddressRegisterA.
	3	-	Repeat #1	AddressRegisterB min to max, fast row	R$\overline{D}$	Read	Read D-data from all addresses using AddressRegisterB. When AddressRegisterB is equivalent to AddressRegisterA the read $\overline{D}$-data is expected. Branch back to Instruction 2 and repeat for all AddressRegisterB row and column addresses.
	4	-	Repeat #1	AddressRegisterA min to max, fast row	RDW$\overline{D}$	ReadModifyWrite	Read $\overline{D}$-data and write D-data at the location addressed by AddressRegisterA. Branch back to Instruction 2 and repeat for all AddressRegisterA row and column addresses.
3	5	-	Repeat #1	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses. After all bank addresses have been accessed repeat phase 0 and 1 with inverted write and expect data.

Example Algorithm Wrapper

illustrates the Algorithm wrapper for the example memory. Two equivalent algorithms (TessentGalPatFastX and TessentGalPatFastY), that have been optimized to eliminate redundant operations, are also available in the *lib/technology/memory_bist* directory of the tool tree. These algorithms run slightly faster, but diagnosis needs to be performed in two steps as explained in the "Diagnosis Considerations" section.

Figure C-7. LVGalPat Example Algorithm Wrapper

```
MemoryOperationsSpecification {
  Algorithm (LVGalPat) {
   TestRegisterSetup {
      operation_set_select : Sync;
      AddressGenerator {
         AddressRegisterA {
            z_carry_in : none;
            y1_carry_in : x1_carry_out;
            x1_carry_in : none;
         }
         AddressRegisterB {
            z_carry_in : none;
            y1_carry_in : x1_carry_out;
            x1_carry_in : none;
         }
      }
      DataGenerator {
         load_write_data  : all_zero;
         load_expect_data : all_zero;
      }
   }
   MicroProgram {
      Instruction (WRITE_BACKGROUND) {
         operation_select : Write;
         AddressCommands {
            address_select : select_a;
            x1_address : increment;
            y1_address : increment;
         }
         DataCommands {
            write_data : data_reg;
         }
         NextConditions {
            x1_end_count : on;
            y1_end_count : on;
         }
      }
      Instruction (WRITE_HOME_CELL) {
         operation_select : Write;
         AddressCommands {
            address_select : select_a;
         }
         DataCommands {
            write_data : inverse_data_reg;
         }
         NextConditions {
         }
      }
      Instruction (READ_HOME_CELL) {
         operation_select : Read;
         AddressCommands {
            address_select : select_a;
         }
         DataCommands {
            expect_data : inverse_data_reg;
```

```
                NextConditions {
                }
            }
            Instruction (READ_AWAY_CELL) {
                operation_select : Read;
                AddressCommands {
                    address_select : select_b;
                    x1_address : increment;
                    y1_address : increment;
                }
                DataCommands {
                    expect_data : data_reg;
                    address_a_equals_b : invert_expect_data;
                }
                branch_to_instruction : READ_HOME_CELL;
                NextConditions {
                    x1_end_count : on;
                    y1_end_count : on;
                }
            }
            Instruction (REWRITE_HOME_CELL_AND_ADVANCE) {
                operation_select : ReadModifyWrite;
                AddressCommands {
                    address_select : select_a;
                    x1_address : increment;
                    y1_address : increment;
                }
                DataCommands {
                    write_data : data_reg;
                    expect_data : inverse_data_reg;
                }
                branch_to_instruction : WRITE_HOME_CELL;
                NextConditions {
                    x1_end_count : on;
                    y1_end_count : on;
                }
            }
            Instruction (GOTO_NEXT_BANKADDRESS) {
                operation_select : NoOperation;
                AddressCommands {
                    address_select : select_a_copy_to_b;
                    z_address : increment;
                }
                branch_to_instruction : WRITE_BACKGROUND;
                NextConditions {
                    z_end_count : on;
                    RepeatLoopB {
                        branch_to_instruction: WRITE_BACKGROUND;
                        Repeat1 {
                            enable : on;
                            write_data_sequence:  inverse;
                            expect_data_sequence: inverse;
                        }
                    }
                }
            }
        }
    }
}
```

```
    }
```

Fault Coverage

The faults detected by the LVGalPat algorithm are identified in Table C-1.

Specification

To test SRAMs using the LVGalPat algorithm, specify Algorithm LVGalPat in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

Usage Conditions

The following usage conditions apply to the LVGalPat algorithm:

- Operations named Write, Read, ReadModifyWrite must exist or be mapped to another operation in OperationSet of the memory TCD file.

- For soft programmable controller usage, the DftSpecification/MemoryBist/Controller/ AlgorithmResourceOptions/soft_instruction_count property must be greater than or equal to three.

- If the MemoryBist/Controller/a_equals_b_command_allowed property is set to off, the LVGalPat algorithm is not available.

LVGalColumn Algorithm

The LVGalColumn algorithm is a test algorithm that is available for loading into the memory controller to perform a galloping column pattern algorithm. The LVGalColumn algorithm is performed as follows:

1. Write background of D-data from address minimum to address maximum for a single bank.

2. Write $\overline{D}$-data to the "home" cell addressed by AddressRegisterA.

3. Read D-data from all "away" cells in the same column (same Y address) of the same bank addressed by AddressRegisterB. Following each read of the "away" cell read $\overline{D}$-data from the "home" cell.

4. Write D-data at the "home" cell to restore the data background.

5. Increment the "home" cell addressed by AddressRegisterA and re-run steps 2 to 5 until every cell in the bank has been a "home" cell.

6. Increment the bank address from minimum to maximum and re-run steps 1 to 6.

7. Repeat steps 1 to 6 with inverted data.

Test Time

The time required for the memory BIST controller to test your design using the LVGalColumn algorithm is outlined in Table C-2.

TestRegisterSetup

The test register setup describes the controller test register values to be initialized for the address and data prior to execution of the algorithm.

AddressRegisterA

The AddressRegisterA segments are configured to sequence the address counting with fast rows as follows:

$$Z \qquad X1 \Leftarrow Y1$$

where:

- The column address segment Y1 counts when instructed.

- The row address segment X1 counts when instructed and a carry out from the Y1 address segment is generated. An y1_carry_out is generated when:

    o The Y1 address segment is incrementing and has reached the maximum OR.

    o The Y1 address segment is decrementing and has reached the minimum.

- The bank address segment Z counts when instructed.

AddressRegisterB

The AddressRegisterB segments are configured to count each of the segments individually where:

- The row address segment X1 counts when instructed.

- The column address segment Y1 counts when instructed.

- The bank address segment Z counts when instructed.

DataRegister

The logical data pattern D is loaded into both the write and expect data registers and is a word of all zeroes.

The logical data pattern applied is inverted with each column. That is, the logical data pattern applied is D for even columns and $\overline{D}$ for odd columns.

Algorithm Sequence

Table C-18 describes the LVGalColumn algorithm sequence.

Table C-18. Description of LVGalColumn Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast column	WD	Write	Write background of D-data.
1	1	-	-	-	W$\overline{D}$	Write	Write $\overline{D}$-data to location addressed by AddressRegisterA.
2	2	-	-	-	R$\overline{D}$	Read	Read $\overline{D}$-data to location addressed by AddressRegisterA.
	3	-	-	AddressRegisterB row address min to max	RD	Read	Read D-data from all row addresses in the same column using AddressRegisterB. When AddressRegisterB is equivalent to AddressRegisterA the read $\overline{D}$-data is expected. Branch back to Instruction 2 and repeat for all AddressRegisterB row addresses.
	4	-	-	-	-	NoOperation	Advance the column address by one.
	5	-	-	AddressRegisterA min to max, fast column	R$\overline{D}$WD	ReadModifyWrite	Read $\overline{D}$-data and write D-data at the location addressed by AddressRegisterA. Branch back to Instruction 2 and repeat for all AddressRegisterA row and column addresses.

Table C-18. Description of LVGalColumn Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
3	6	-	-	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses. After all bank addresses have been accessed branch to instruction 0 and repeat one time as follows with RepeatLoopB: • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing.
4	0	-	Repeat #1	min to max, fast column	WD	Write	Write background of D-data.
5	1	-	Repeat #1	-	W$\overline{D}$	Write	Write $\overline{D}$-data to location addressed by AddressRegisterA.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Table C-18. Description of LVGalColumn Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
6	2	-	Repeat #1	-	R$\overline{\text{D}}$	Read	Read $\overline{\text{D}}$-data to location addressed by AddressRegisterA.
	3	-	Repeat #1	AddressRegisterB row address min to max	RD	Read	Read D-data from all row addresses in the same column using AddressRegisterB. When AddressRegisterB is equivalent to AddressRegisterA the read $\overline{\text{D}}$-data is expected. Branch back to Instruction 2 and repeat for all AddressRegisterB row addresses.
	4	-	Repeat #1	-	-	NoOperation	Advance the column address by one.
	5	-	Repeat #1	AddressRegisterA min to max, fast column	R$\overline{\text{D}}$WD	ReadModifyWrite	Read $\overline{\text{D}}$-data and write D-data at the location addressed by AddressRegisterA. Branch back to Instruction 2 and repeat for all AddressRegisterA row and column addresses.
7	6	-	Repeat #1	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses. After all bank addresses have been accessed repeat phase 0 and 1 with inverted write and expect data.

Example Algorithm Wrapper

illustrates the Algorithm wrapper for the example memory.

Figure C-8. LVGalColumn Example Algorithm Wrapper

```
MemoryOperationsSpecification {
  Algorithm (LVGalColumn) {
   TestRegisterSetup {
      operation_set_select : Sync;
      AddressGenerator {
         AddressRegisterA {
            z_carry_in : none;
            x1_carry_in : y1_carry_out;
            y1_carry_in : none;
         }
         AddressRegisterB {
            z_carry_in : none;
            y1_carry_in : none;
            x1_carry_in : none;
         }
      }
      DataGenerator {
         load_write_data: all_zero;
         load_expect_data: all_zero;
      }
    }
   MicroProgram {
      Instruction (WRITE_BACKGROUND) {
         operation_select : Write;
         AddressCommands {
            address_select : select_a;
            x1_address : increment;
            y1_address : increment;
         }
         DataCommands {
            write_data : data_reg;
         }
         NextConditions {
            x1_end_count : on;
            y1_end_count : on;
         }
      }
      Instruction (WRITE_HOME_CELL) {
         operation_select : Write;
         AddressCommands {
            address_select : select_a;
         }
         DataCommands {
            write_data : inverse_data_reg;
         }
         NextConditions {
         }
      }
      Instruction (READ_HOME_CELL) {
         operation_select : Read;
         AddressCommands {
            address_select : select_a;
         }
         DataCommands {
            expect_data : inverse_data_reg;
```

```
        }
        NextConditions {
        }
    }
    Instruction (READ_AWAY_COLUMN) {
        operation_select : Read;
        AddressCommands {
            address_select : select_b;
            x1_address : increment;
        }
        DataCommands {
            expect_data : data_reg;
            address_a_equals_b : invert_expect_data;
        }
        branch_to_instruction : READ_HOME_CELL;
        NextConditions {
            x1_end_count : on;
        }
    }
    Instruction (ADVANCE_AWAY_COLUMN_POINTER) {
        operation_select : NoOperation;
        AddressCommands {
            address_select : select_b;
            y1_address : increment;
        }
        NextConditions {
        }
    }
    Instruction (REWRITE_HOME_CELL_AND_ADVANCE) {
        operation_select : Readmodifywrite;
        AddressCommands {
            address_select : select_a;
            x1_address : increment;
            y1_address : increment;
        }
        DataCommands {
            write_data : data_reg;
            expect_data : inverse_data_reg;
        }
        branch_to_instruction : WRITE_HOME_CELL;
        NextConditions {
            x1_end_count : on;
            y1_end_count : on;
        }
    }
    Instruction (GOTO_NEXT_BANKADDRESS) {
        operation_select : Nooperation;
        AddressCommands {
            address_select : select_a_copy_to_b;
            z_address : increment;
        }
        branch_to_instruction : WRITE_BACKGROUND;
        NextConditions {
            z_end_count : on;
            RepeatLoopB {
                branch_to_instruction: WRITE_BACKGROUND;
                Repeat1 {
                    enable : on;
```

```
                    write_data_sequence: inverse;
                    expect_data_sequence: inverse;
                }
            }
        }
    }
}
```

Fault Coverage

The faults detected by the LVGalColumn algorithm are identified in Table C-1. Only coupling faults occurring in the same column are detected.

Specification

To test SRAMs using the LVGalColumn algorithm, specify Algorithm LVGalColumn in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

Usage Conditions

The following usage conditions apply to the LVGalColumn algorithm:

- Operations named Write, Read, ReadModifyWrite must exist or be mapped to another operation in OperationSet of the memory TCD file.

- For soft programmable controller usage, the DftSpecification/MemoryBist/Controller/ AlgorithmResourceOptions/soft_instruction_count property must be greater than or equal to three.

- If the MemoryBist/Controller/a_equals_b_command_allowed property is set to off, the LVGalColumn algorithm is not available.

LVGalRow Algorithm

The LVGalRow algorithm is a test algorithm that is available for loading into the memory controller to perform a galloping row algorithm. The LVGalRow algorithm is performed as follows:

1. Write background of D-data from address minimum to address maximum for a single bank.

2. Write $\overline{D}$-data to the "home" cell addressed by AddressRegisterA.

3. Read D-data from all "away" cells in the same row (same X address) of the same bank addressed by AddressRegisterB. Following each read of the "away" cell read $\overline{D}$-data from the "home" cell.

4. Write D-data at the "home" cell to restore the data background.

5. Increment the "home" cell addressed by AddressRegisterA and re-run steps 2 to 5 until every cell in the bank has been a "home" cell.

6. Increment the bank address from minimum to maximum and re-run steps 1 to 6.

7. Repeat steps 1.) to 6.) with inverted data.

Test Time

The time required for the memory BIST controller to test your design using the LVGalRow algorithm is outlined in Table C-2.

TestRegisterSetup

The test register setup describes the controller test register values to be initialized for the address and data prior to execution of the algorithm.

AddressRegisterA

The AddressRegisterA segments are configured to sequence the address counting with fast rows as follows:

$$Z \qquad Y1 \Leftarrow X1$$

where:

- The row address segment X1 counts when instructed.

- The column address segment Y1 counts when instructed and a carry out from the X1 address segment is generated. An x1_carry_out is generated when:

    o   The X1 address segment is incrementing and has reached the maximum OR.

    o   The X1 address segment is decrementing and has reached the minimum.

- The bank address segment Z counts when instructed.

AddressRegisterB

The AddressRegisterB segments are configured to count each of the segments individually where:

- The row address segment X1 counts when instructed.

- The column address segment Y1 counts when instructed.

- The bank address segment Z counts when instructed.

DataRegister

The logical data pattern D is loaded into both the write and expect data registers and is a word of all zeroes.

The logical data pattern applied is inverted with each column. That is, the logical data pattern applied is D for even columns and D for odd columns.

Algorithm Sequence

describes the LVGalRow algorithm sequence.

Table C-19. Description of LVGalRow Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast row	WD	Write	Write background of D-data.
1	1	-	-	-	W$\overline{\text{D}}$	Write	Write $\overline{\text{D}}$-data to location addressed by AddressRegisterA.

Table C-19. Description of LVGalRow Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
2	2	-	-	-	$R\overline{D}$	Read	Read $\overline{D}$-data to location addressed by AddressRegisterA.
	3	-	-	AddressRegisterB column address min to max	RD	Read	Read D-data from all column addresses in the same row using AddressRegisterB. When AddressRegisterB is equivalent to AddressRegisterA the read $\overline{D}$-data is expected. Branch back to Instruction 2 and repeat for all AddressRegisterB column addresses.
	4	-	-	-	-	NoOperation	Advance the row address by one.
	5	-	-	AddressRegisterA min to max, fast row	$R\overline{D}WD$	ReadModifyWrite	Read $\overline{D}$-data and write D-data at the location addressed by AddressRegisterA. Branch back to Instruction 2 and repeat for all AddressRegisterA row and column addresses.
3	6	-	-	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses. After all bank addresses have been accessed branch to instruction 0 and repeat one time as follows with RepeatLoopB: • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing.

Table C-19. Description of LVGalRow Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
4	0	-	Repeat #1	min to max, fast row	WD	Write	Write background of D-data.
5	1	-	Repeat #1	-	W$\overline{D}$	Write	Write $\overline{D}$-data to location addressed by AddressRegisterA.
6	2	-	Repeat #1	-	R$\overline{D}$	Read	Read $\overline{D}$-data to location addressed by AddressRegisterA.
	3	-	Repeat #1	AddressRegisterB column address min to max	RD	Read	Read D-data from all column addresses in the same row using AddressRegisterB. When AddressRegisterB is equivalent to AddressRegisterA the read $\overline{D}$-data is expected. Branch back to Instruction 2 and repeat for all AddressRegisterB column addresses.
	4	-	Repeat #1	-	-	NoOperation	Advance the row address by one.
	5	-	Repeat #1	AddressRegisterA min to max, fast row	R$\overline{D}$WD	ReadModifyWrite	Read $\overline{D}$-data and write D-data at the location addressed by AddressRegisterA. Branch back to Instruction 2 and repeat for all AddressRegisterA row and column addresses.

Table C-19. Description of LVGalRow Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
7	6	-	Repeat #1	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses. After all bank addresses have been accessed repeat phase 0 and 1 with inverted write and expect data.

Example Algorithm Wrapper

Figure C-9 illustrates the Algorithm wrapper for the example memory.

Figure C-9. LVGalRow Example Algorithm Wrapper

```
MemoryOperationsSpecification {
  Algorithm (LVGalRow) {
   TestRegisterSetup {
      operation_set_select : Sync;
      AddressGenerator {
         AddressRegisterA {
            z_carry_in : none;
            y1_carry_in : x1_carry_out;
            x1_carry_in : none;
         }
      }
      DataGenerator {
         load_write_data: all_zero;
         load_expect_data: all_zero;
      }
   }
   MicroProgram {
      Instruction (WRITE_BACKGROUND) {
         operation_select : Write;
         AddressCommands {
            address_select : select_a;
            x1_address : increment;
            y1_address : increment;
         }
         DataCommands {
            write_data : data_reg;
         }
         NextConditions {
            x1_end_count : on;
            y1_end_count : on;
         }
      }
      Instruction (WRITE_HOME_CELL) {
         operation_select : Write;
         AddressCommands {
            address_select : select_a;
         }
         DataCommands {
            write_data : inverse_data_reg;
         }
         NextConditions {
         }
      }
      Instruction (READ_HOME_CELL) {
         operation_select : Read;
         AddressCommands {
            address_select : select_a;
         }
         DataCommands {
            expect_data : inverse_data_reg;
         }
         NextConditions {
         }
      }
      Instruction (READ_AWAY_ROW) {
```

```
            operation_select : Read;
            AddressCommands {
                address_select : select_b;
                y1_address : increment;
            }
            DataCommands {
                expect_data : data_reg;
                address_a_equals_b : invert_expect_data;
            }
            branch_to_instruction : READ_HOME_CELL;
            NextConditions {
                y1_end_count : on;
            }
        }
        Instruction (ADVANCE_AWAY_ROW_POINTER) {
            operation_select : Nooperation;
            AddressCommands {
                address_select : select_b;
                x1_address : increment;
            }
            NextConditions {
            }
        }
        Instruction (REWRITE_HOME_CELL_AND_ADVANCE) {
            operation_select : Readmodifywrite;
            AddressCommands {
                address_select : select_a;
                x1_address : increment;
                y1_address : increment;
            }
            DataCommands {
                write_data : data_reg;
                expect_data : inverse_data_reg;
            }
            branch_to_instruction : WRITE_HOME_CELL;
            NextConditions {
                x1_end_count : on;
                y1_end_count : on;
            }
        }
        Instruction (GOTO_NEXT_BANKADDRESS) {
            operation_select : Nooperation;
            AddressCommands {
                address_select : select_a_copy_to_b;
                z_address : increment;
            }
            branch_to_instruction : WRITE_BACKGROUND;
            NextConditions {
                z_end_count : on;
                RepeatLoopB {
                    branch_to_instruction: WRITE_BACKGROUND;
                    Repeat1 {
                        enable : on;
                        write_data_sequence: inverse;
                        expect_data_sequence: inverse;
                    }
                }
            }
```

```
            }
          }
        }
      }
```

Fault Coverage

The faults detected by the LVGalRow algorithm are identified in Table C-1. Only coupling faults occurring in the same row are detected.

Specification

To test SRAMs using the LVGalRow algorithm, specify Algorithm LVGalRow in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

Usage Conditions

The following usage conditions apply to the LVGalRow algorithm:

- Operations named Write, Read, ReadModifyWrite must exist or be mapped to another operation in OperationSet of the memory TCD file.

- For soft programmable controller usage, the DftSpecification/MemoryBist/Controller/ AlgorithmResourceOptions/soft_instruction_count property must be greater than or equal to three.

- If the MemoryBist/Controller/a_equals_b_command_allowed property is set to off, the LVGalRow algorithm is not available.

LVCheckerboard1X1 Algorithm

The LVCheckerboard1X1 algorithm is a test algorithm that is available for loading into the memory controller to perform a 1X1 Checkerboard algorithm.

The checkerboard 1x1 algorithm is described in Table C-20.

Test Time

The time required for the memory BIST controller to test your design using the LVCheckerboard1X1 algorithm is outlined in Table C-2.

TestRegisterSetup

The test register setup describes the controller test register values to be initialized for the address and data prior to execution of the algorithm.

AddressRegisterA

The AddressRegisterA segments are configured to sequence the address counting with fast rows as follows:

$$Z \qquad X1 \Leftarrow Y1$$

where:

- The column address segment Y1 counts when instructed.

- The row address segment X1 counts when instructed and a carry out from the Y1 address segment is generated. A y1_carry_out is generated when:

  o The Y1 address segment is incrementing and has reached the maximum OR.

  o The Y1 address segment is decrementing and has reached the minimum.

- The bank address segment Z counts when instructed.

DataRegister

The logical data pattern D is loaded into both the write and expect data registers and is a word of all zeroes.

The logical data pattern applied is inverted with each column and with each row. That is the logical data pattern applied is:

- D when column address bit 0 = 1'b0 and row address bit 0 = 1'b0

- $\overline{D}$ when column address bit 0 = 1'b1 and row address bit 0 = 1'b0

- $\overline{D}$ when column address bit 0 = 1'b0 and row address bit 0 = 1'b1

- D when column address bit 0 = 1'b1 and row address bit 0 = 1'b1

Algorithm Sequence

Table C-20 describes the LVCheckerboard1X1 algorithm sequence.

Table C-20. Description of LVCheckerboard1X1 Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast row	WD	Write	Write 1X1 checkerboard D-data background.
1	1	-	-	min to max, fast row	RD	Read	Read 1x1 checkerboard D-data.

Table C-20. Description of LVCheckerboard1X1 Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
2	5	-	-	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses. After all bank addresses have been accessed branch to instruction 0 and repeat one time as follows with RepeatLoopB: • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing.
3	0	Repeat #1	-	min to max, fast row	$W\overline{D}$	Write	Write 1X1 checkerboard $\overline{D}$-data background.
4	1	Repeat #1	-	min to max, fast row	$R\overline{D}$	Read	Read 1x1 checkerboard $\overline{D}$-data.
5	5	Repeat #1	-	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses.

Example Algorithm Wrapper

Figure C-10 illustrates the Algorithm wrapper for the example memory. Two equivalent algorithms (TessentCheckerboard1x1FastX and TessentCheckerboard1x1FastY), that have been optimized to eliminate redundant operations, are also available in the *lib/technology/ memory_bist* directory of the tool tree. These algorithms run slightly faster, but diagnosis needs to be performed in two steps as explained in the "Diagnosis Considerations" section.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Figure C-10. LVCheckerboard1X1 Example Algorithm Wrapper

```
MemoryOperationsSpecification {
  Algorithm (LVCheckerboard1X1){
   TestRegisterSetup {
      operation_set_select : Sync;
      AddressGenerator {
         AddressRegisterA {
            z_carry_in : none;
            x1_carry_in : y1_carry_out;
            y1_carry_in : none;
         }
      }
      DataGenerator {
         InvertDataWithRowBit: r[0];
         InvertDataWithColumnBit: c[0];
      }
   }
   MicroProgram {
      Instruction (W_1X1_CHECKERBOARD) {
         operation_select : Write;
         AddressCommands {
            address_select : select_a;
            x1_address : increment;
            y1_address : increment;
         }
         DataCommands {
            write_data : zero;
         }
         NextConditions {
            x1_end_count : on;
            y1_end_count : on;
         }
      }
```

```
            Instruction (R_1X1_CHECKERBOARD) {
                operation_select : Read;
                AddressCommands {
                    address_select : select_a;
                    x1_address : increment;
                    y1_address : increment;
                }
                DataCommands {
                    expect_data : zero;
                }
                NextConditions {
                    x1_end_count : on;
                    y1_end_count : on;
                }
            }
            Instruction (GOTO_NEXT_BANKADDRESS) {
                operation_select : Nooperation;
                AddressCommands {
                    address_select : select_a;
                    z_address : increment;
                }
                branch_to_instruction : W_1X1_CHECKERBOARD;
                NextConditions {
                    z_end_count : on;
                    RepeatLoopA {
                        branch_to_instruction: W_1X1_CHECKERBOARD;
                        Repeat1 {
                            enable : on;
                            write_data_sequence: inverse;
                            expect_data_sequence: inverse;
                        }
                    }
                }
            }
        }
    }
}
```

Fault Coverage

The faults detected by the LVCheckerboard1X1 algorithm are identified in Table C-1. Stuck-at faults are detected correctly when there are no address decoder faults.

Specification

To test SRAMs using the LVCheckerboard1X1 algorithm, specify Algorithm LVCheckerboard1X1 in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

Usage Conditions

The following usage conditions apply to the LVCheckerboard1X1 algorithm:

- The LVCheckerboard1X1 algorithm is only available if the memory controller is testing a memory with at least 1 row address bit and 1 column address bit. For memories that utilize only row address, the SMarchCHKB, SMarchCHKBci, SMarchCHKBcil and SMarchCHKBvcd algorithms can be used.

- Operations named Write, ReadModifyWrite, and Read must exist or be mapped to another operation in OperationSet of the memory TCD file.

- For soft programmable controller usage, the DftSpecification/MemoryBist/Controller/AlgorithmResourceOptions/soft_instruction_count property must be greater than or equal to three.

LVCheckerboard4X4 Algorithm

The LVCheckerboard4X4 algorithm is a test algorithm that is available for loading into the memory controller to perform a 4X4 Checkerboard algorithm.

The checkerboard 4x4 algorithm is described in Table C-21.

Test Time

The time required for the memory BIST controller to test your design using the LVCheckerboard4X4 algorithm is outlined in Table C-2.

TestRegisterSetup

The test register setup describes the controller test register values to be initialized for the address and data prior to execution of the algorithm.

AddressRegisterA

The AddressRegisterA segments are configured to sequence the address counting with fast rows as follows:

$$Z \qquad X1 \Leftarrow Y1$$

where:

- The column address segment Y1 counts when instructed.

- The row address segment X1 counts when instructed and a carry out from the Y1 address segment is generated. A y1_carry_out is generated when:

  o The Y1 address segment is incrementing and has reached the maximum OR.

> o The Y1 address segment is decrementing and has reached the minimum.

- The bank address segment Z counts when instructed.

DataRegister

The logical data pattern D is loaded into both the write and expect data registers and is a word of all zeroes.

The logical data pattern applied is inverted after every two columns and every two row. That is, the logical data pattern applied is:

- D when column address bit 1 = 1'b0 and row address bit 1 = 1'b0

- $\overline{D}$ when column address bit 1 = 1'b1 and row address bit 1 = 1'b0

- $\overline{D}$ when column address bit 1 = 1'b0 and row address bit 1 = 1'b1

- D when column address bit 1 = 1'b1 and row address bit 1 = 1'b1

Algorithm Sequence

Table C-21 describes the LVCheckerboard4X4 algorithm sequence.

Table C-21. Description of LVCheckerboard4X4 Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast row	WD	Write	Write 4X4 checkerboard D-data background.
1	1	-	-	min to max, fast row	RD	Read	Read 4X4 checkerboard D-data.
2	5	-	-	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses. After all bank addresses have been accessed branch to instruction 0 and repeat one time as follows with RepeatLoopB: • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing.

Table C-21. Description of LVCheckerboard4X4 Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
3	0	Repeat #1	-	min to max, fast row	W$\overline{D}$	Write	Write 4X4 checkerboard $\overline{D}$-data background.
4	1	Repeat #1	-	min to max, fast row	R$\overline{D}$	Read	Read 4X4 checkerboard $\overline{D}$-data.
5	5	Repeat #1	-	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses.

Example Algorithm Wrapper

Figure C-11 illustrates the Algorithm wrapper for the example memory. Two equivalent algorithms (TessentCheckerboard4x4FastX and TessentCheckerboard4x4FastY), that have been optimized to eliminate redundant operations, are also available in the *lib/technology/ memory_bist* directory of the tool tree. These algorithms run slightly faster, but diagnosis needs to be performed in two steps as explained in the "Diagnosis Considerations" section.

Figure C-11. LVCheckerboard4X4 Example Algorithm Wrapper

```
MemoryOperationsSpecification {
  Algorithm (LVCheckerboard4X4){
   TestRegisterSetup {
      operation_set_select : Sync;
      AddressGenerator {
         AddressRegisterA {
            z_carry_in : none;
            x1_carry_in : y1_carry_out;
            y1_carry_in : none;
         }
      }
      DataGenerator {
         invert_data_with_row_bit    : r[1];
         invert_data_with_column_bit: c[1];
      }
   }
  MicroProgram {
      Instruction (W_4X4_CHECKERBOARD) {
         operation_select : Write;
         AddressCommands {
            address_select : select_a;
            x1_address : increment;
            y1_address : increment;
         }
         DataCommands {
            write_data : zero;
         }
         NextConditions {
            x1_end_count : on;
            y1_end_count : on;
         }
      }
```

```
Instruction (R_4X4_CHECKERBOARD) {
    operation_select : Read;
    AddressCommands {
        address_select : select_a;
        x1_address : increment;
        y1_address : increment;
    }
    DataCommands {
        expect_data : zero;
    }
    NextConditions {
        x1_end_count : on;
        y1_end_count : on;
    }
}
Instruction (GOTO_NEXT_BANKADDRESS) {
    operation_select : Nooperation;
    AddressCommands {
        address_select : select_a;
        z_address : increment;
    }
    branch_to_instruction : W_4X4_CHECKERBOARD;
    NextConditions {
        z_end_count : on;
        RepeatLoopA {
            branch_to_instruction: W_4X4_CHECKERBOARD;
            Repeat1 {
                enable : on;
                write_data_sequence: inverse;
                expect_data_sequence: inverse;
            }
        }
    }
}
}
}
}
}
```

Fault Coverage

The faults detected by the LVCheckerboard4X4 algorithm are identified in Table C-1. Stuck-at faults are detected correctly when there are no address decoder faults.

Specification

To test SRAMs using the LVCheckerboard4X4 algorithm, specify Algorithm LVCheckerboard4X4 in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

Usage Conditions

The following usage conditions apply to the LVCheckerboard4X4 algorithm:

- The LVCheckerboard4X4 algorithm is only available if the memory controller is testing a memory with at least 2 row address bits and 2 column address bits. Memories with at least 1 row and 1 column address bit can utilize the LVCheckerboard1X1 algorithm as an alternative. Memories with only row address bits can utilize the SMarchCHKB, SMarchCHKBci, SMarchCHKBcil and SMarchCHKBvcd algorithms.

- Operations named Write, ReadModifyWrite, and Read must exist or be mapped to another operation in OperationSet of the memory TCD file.

- For soft programmable controller usage, the DftSpecification/MemoryBist/Controller/ AlgorithmResourceOptions/soft_instruction_count property must be greater than or equal to three.

- If the *max_data_inversion_address_bit_index* property is set to 0, the LVCheckerboard4X4 algorithm is not available.

LVWalkingPat Algorithm

The LVWalkingPat algorithm is a test algorithm that is available for loading into the memory controller to perform a walking pattern algorithm.

The LVWalkingPat algorithm is described in Table C-22.

Test Time

The time required for the memory BIST controller to test your design using the LVWalkingPat algorithm is outlined in Table C-2.

TestRegisterSetup

The test register setup describes the controller test register values to be initialized for the address and data prior to execution of the algorithm.

AddressRegisterA

The AddressRegisterA segments are configured to sequence the address counting with fast rows as follows:

$$Z \qquad Y1 \Leftarrow X1$$

where:

- The row address segment X1 counts when instructed.

- The column address segment Y1 counts when instructed and a carry out from the X1 address segment is generated. An x1_carry_out is generated when:

  o The X1 address segment is incrementing and has reached the maximum OR.

  o The X1 address segment is decrementing and has reached the minimum.

- The bank address segment Z counts when instructed.

AddressRegisterB

The AddressRegisterB segments are configured to identical to AddressRegisterA.

DataRegister

The logical data pattern D is loaded into both the write and expect data registers and is a word of all zeroes.

Algorithm Sequence

Table C-22 describes the LVWalkingPat algorithm sequence.

Table C-22. Description of LVWalkingPat Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast row	WD	Write	Write background of D-data.
1	1	-	-	-	W$\overline{\text{D}}$	Write	Write $\overline{\text{D}}$-data to location addressed by AddressRegisterA.
	2	-	-	AddressRegisterB min to max, fast row	RD	Read	Read D-data from the location addressed by AddressRegisterB. Read $\overline{\text{D}}$-data when AddressRegisterB equal AddressRegisterA.
	3	-	-	AddressRegisterB min to max, fast row	R$\overline{\text{D}}$WD	ReadModifyWrite	Read $\overline{\text{D}}$-data and Write D-data to the location addressed by AddressRegisterA. Branch back to Instruction 1 and repeat for all AddressRegisterA row and column addresses.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Table C-22. Description of LVWalkingPat Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
2	4	-	-	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses. After all bank addresses have been accessed branch to instruction 0 and repeat one time as follows with RepeatLoopB: • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing.
3	0	-	Repeat #1	min to max, fast row	W$\overline{\text{D}}$	Write	Write background of $\overline{\text{D}}$-data.
4	1	-	Repeat #1	-	WD	Write	Write D-data to location addressed by AddressRegisterA.
	2	-	Repeat #1	AddressRegisterB min to max, fast row	R$\overline{\text{D}}$	Read	Read $\overline{\text{D}}$-data from the location addressed by AddressRegisterB. Read D-data when AddressRegisterB equal AddressRegisterA.
	3	-	Repeat #1	AddressRegisterB min to max, fast row	RDW$\overline{\text{D}}$	ReadModifyWrite	Read D-data and Write $\overline{\text{D}}$-data to the location addressed by AddressRegisterA. Branch back to Instruction 1 for all AddressRegisterA row and column addresses.

Table C-22. Description of LVWalkingPat Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
5	4	-	Repeat #1	min to max, bank addresses	-	NoOperation	Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses. After all bank addresses have been accessed repeat phase 0,1, and 2 with inverted write and expect data.

Example Algorithm Wrapper

Figure C-12 illustrates the Algorithm wrapper for the example memory. Two equivalent algorithms (TessentWalkingPatFastX and TessentWalkingPatFastY), that have been optimized to eliminate redundant operations, are also available in the *lib/technology/memory_bist* directory of the tool tree. These algorithms run slightly faster, but diagnosis needs to be performed in two steps as explained in the "Diagnosis Considerations" section.

Figure C-12. LVWalkingPat Example Algorithm Wrapper

```
MemoryOperationsSpecification {
  Algorithm (LVWalkingPat) {
   TestRegisterSetup {
       operation_set_select : Sync;
       AddressGenerator {
           AddressRegisterA {
               z_carry_in : none;
               y1_carry_in : x1_carry_out;
               x1_carry_in : none;
           }
           AddressRegisterB {
               z_carry_in : none;
               y1_carry_in : x1_carry_out;
               x1_carry_in : none;
           }
       }
       DataGenerator {
           load_write_data: all_zero;
           load_expect_data: all_zero;
       }
   }
   MicroProgram {
       Instruction (WRITE_BACKGROUND) {
           operation_select : Write;
           AddressCommands {
               address_select : select_a;
               x1_address : increment;
               y1_address : increment;
           }
           DataCommands {
               write_data : data_reg;
           }
           NextConditions {
               x1_end_count : on;
               y1_end_count : on;
           }
       }
       Instruction (WRITE_HOME_CELL) {
           operation_select : Write;
           AddressCommands {
               address_select : select_a;
           }
           DataCommands {
               write_data : inverse_data_reg;
           }
           NextConditions {
           }
       }
       Instruction (READ_AWAY_CELL) {
           operation_select : Read;
           AddressCommands {
               address_select : select_b;
               x1_address : increment;
               y1_address : increment;
           }
```

```
        DataCommands {
            expect_data : data_reg;
            address_a_equals_b : invert_expect_data;
        }
        NextConditions {
            x1_end_count : on;
            y1_end_count : on;
        }
    }
    Instruction (REWRITE_HOME_CELL_AND_ADVANCE) {
        operation_select : ReadModifyWrite;
        AddressCommands {
            address_select : select_a;
            x1_address : increment;
            y1_address : increment;
        }
        DataCommands {
            write_data : data_reg;
            expect_data : inverse_data_reg;
        }
        branch_to_instruction : WRITE_HOME_CELL;
        NextConditions {
            x1_end_count : on;
            y1_end_count : on;
        }
    }
    Instruction (GOTO_NEXT_BANKADDRESS) {
        operation_select : NoOperation;
            address_select : select_a_copy_to_b;
            z_address : increment;
        branch_to_instruction : WRITE_BACKGROUND;
        NextConditions {
            z_end_count : on;
            RepeatLoopB {
                branch_to_instruction: WRITE_BACKGROUND;
                Repeat1 {
                    enable : on;
                    write_data_sequence: inverse;
                    expect_data_sequence: inverse;
                }
            }
        }
    }
}
}
}
```

Fault Coverage

The faults detected by the LVWalkingPat algorithm are identified in Table C-1.

Specification

To test SRAMs using the LVWalkingPat algorithm, specify Algorithm LVWalkingPat in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

Usage Conditions

The following usage conditions apply to the LVWalkingPat algorithm:

- Operations named Write, Read, ReadModifyWrite must exist or be mapped to another operation in OperationSet of the memory TCD file.

- For soft programmable controller usage, the DftSpecification/MemoryBist/Controller/ AlgorithmResourceOptions/soft_instruction_count property must be greater than or equal to three.

- If the *a_equals_b_command_allowed* property is set to off, the LVWalkingPat algorithm is not available.

LVBitSurroundDisturb Algorithm

The LVBitSurroundDisturb algorithm is a test algorithm that is available for loading into the memory controller to perform a bit surround disturb algorithm.

The LVBitSurroundDisturb algorithm is described in Table C-23.

Test Time

The time required for the memory BIST controller to test your design using the LVBitSurroundDisturb algorithm is outlined in Table C-2.

TestRegisterSetup

The test register setup describes the controller test register values to be initialized for the address and data prior to execution of the algorithm.

AddressRegisterA

The AddressRegisterA segments are configured to sequence the address counting with fast rows as follows:

$$Z \qquad Y1 \Leftarrow X1$$

where:

- The row address segment X1 counts when instructed.

- The column address segment Y1 counts when instructed and a carry out from the X1 address segment is generated. An x1_carry_out is generated when:

  o The X1 address segment is incrementing and has reached the maximum OR.

  o The X1 address segment is decrementing and has reached the minimum.

- The bank address segment Z counts when instructed.

- Initial value of a AddressRegisterA is Z = 0, Y = 0, X = 0.

AddressRegisterB

The AddressRegisterB segments are configured to independently count each address segment as follows:

$$Z \qquad X1 \qquad Y1$$

where:

- The row address segment X1 counts when instructed.

- The column address segment Y1 counts when instructed.

- The bank address segment Z counts when instructed.

- Initial value of a AddressRegisterB is: Z = 0, Y = 0, X = 0.

DataRegister

The logical data pattern D is loaded into both the write and expect data registers and is a word of all zeroes.

CounterA EndCount Register

The Counter A EndCount register is loaded with the binary equivalent to the decimal value '1'.

Algorithm Sequence

Table C-23 describes the LVBitSurroundDisturb algorithm sequence.

Table C-23. Description of LVBitSurroundDisturb Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast row	WD	Write	Write background of D-data.
1	1	-	-	-	-	NoOperation	Offset AddressRegisterB by decrementing one Row Address and decrementing one Column Address.

Table C-23. Description of LVBitSurroundDisturb Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
2	2	-	-	AddressRegisterB, increment column address	W$\overline{D}$	Write	Write $\overline{D}$-data at the location addressed by AddressRegisterB. Increment AddressRegisterB column address by one.
	3	-	-	AddressRegisterA	RD	Read	Read D-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice).
	4	-	-	AddressRegisterB, increment row address	W$\overline{D}$	Write	Write $\overline{D}$-data at the location addressed by AddressRegisterB. Increment AddressRegisterB row address by one.
	5	-	-	AddressRegisterA	RD	Read	Read D-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice). Repeat instructions 2-5 three times as follows with RepeatLoopB: • Repeat #1 - branch to instruction 2 and repeat instructions with inverted the address sequencing • Repeat #2 - branch to instruction 2 and repeat instructions with inverted the write data sequencing. • Repeat #3 -branch to instruction 2 and repeat instructions with inverted the address and write data sequencing.

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

Table C-23. Description of LVBitSurroundDisturb Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
3	2	-	Repeat #1	AddressRegisterB, decrement column address	W$\overline{\text{D}}$	Write	Write $\overline{\text{D}}$-data at the location addressed by AddressRegisterB. Decrement AddressRegisterB column address by one.
	3	-	Repeat #1	AddressRegisterA	RD	Read	Read D-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice).
	4	-	Repeat #1	AddressRegisterB, decrement row address	W$\overline{\text{D}}$	Write	Write $\overline{\text{D}}$-data at the location addressed by AddressRegisterB. Decrement AddressRegisterB row address by one.
	5	-	Repeat #1	AddressRegisterA	RD	Read	Read D-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counter counts from zero to CounterA endcount (twice). Repeat instructions 2-5 three times as follows with RepeatLoopB: • Repeat #1 - branch to instruction 2 and repeat instructions with inverted the address sequencing • Repeat #2 - branch to instruction 2 and repeat instructions with inverted the write data sequencing. • Repeat #3 -branch to instruction 2 and repeat instructions with inverted the address and write data sequencing.

Table C-23. Description of LVBitSurroundDisturb Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
4	2	-	Repeat #2	AddressRegisterB, increment column address	WD	Write	Write D-data at the location addressed by AddressRegisterB. Increment AddressRegisterB column address by one.
	3	-	Repeat #2	AddressRegisterA	RD	Read	Read D-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice).
	4	-	Repeat #2	AddressRegisterB, increment row address	WD	Write	Write D-data at the location addressed by AddressRegisterB. Increment AddressRegisterB row address by one.
	5	-	Repeat #2	AddressRegisterA	RD	Read	Read D-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice). Repeat instructions 2-5 three times as follows with RepeatLoopB: - Repeat #1 - branch to instruction 2 and repeat instructions with inverted the address sequencing - Repeat #2 - branch to instruction 2 and repeat instructions with inverted the write data sequencing. - Repeat #3 -branch to instruction 2 and repeat instructions with inverted the address and write data sequencing.

Table C-23. Description of LVBitSurroundDisturb Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
5	2	-	Repeat #3	AddressRegisterB, decrement the column address	WD	Write	Write D-data at the location addressed by AddressRegisterB. Decrement AddressRegisterB column address by one.
	3	-	Repeat #3	AddressRegisterA	RD	Read	Read D-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice).
	4	-	Repeat #3	AddressRegisterB, decrement the row address	WD	Write	Write D-data at the location addressed by AddressRegisterB. Decrement AddressRegisterB row address by one.
	5	-	Repeat #3	AddressRegisterA	RD	Read	Read D-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice). Repeat instructions 2-5 three times as follows with RepeatLoopB: • Repeat #1 - branch to instruction 2 and repeat instructions with inverted the address sequencing • Repeat #2 - branch to instruction 2 and repeat instructions with inverted the write data sequencing. • Repeat #3 -branch to instruction 2 and repeat instructions with inverted the address and write data sequencing.

Table C-23. Description of LVBitSurroundDisturb Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
6	6	-	-	Address RegisterA, increment min to max, fast rows	-	NoOperation	Increment AddressRegisterA and copy the result to AddressRegisterB. Branch back to Instruction 1 and re-run phases 1 through 5 for all AddressRegisterA row and column addresses.
7	7	-	-	Address RegisterA, bank addresses, min to max	-	NoOperation	Increment the bank address for AddressRegisterA and copy the result to AddressRegisterB. Branch back to Instruction 0 and re-run phases 1 through 7 for all bank addresses. Repeat instructions 0-7 once as follows with RepeatLoopA: • Repeat #1 - branch to instruction 0 and repeat instructions with inverted the write and expect data sequencing
0	0	Repeat #1	-	min to max, fast row	$W\overline{D}$	Write	Write background of $\overline{D}$-data.
1	1	Repeat #1	-	-	-	NoOperation	Offset AddressRegisterB by decrementing one Row Address and decrementing one Column Address.

Table C-23. Description of LVBitSurroundDisturb Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
2	2	Repeat #1	-	AddressRegisterB, increment column address	WD	Write	Write D-data at the location addressed by AddressRegisterB. Increment AddressRegisterB column address by one.
	3	Repeat #1	-	AddressRegisterA	R$\overline{D}$	Read	Read $\overline{D}$-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice).
	4	Repeat #1	-	AddressRegisterB, increment row address	WD	Write	Write D-data at the location addressed by AddressRegisterB. Increment AddressRegisterB row address by one.
	5	Repeat #1	-	AddressRegisterA	R$\overline{D}$	Read	Read $\overline{D}$-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice). Repeat instructions 2-5 three times as follows with RepeatLoopB: • Repeat #1 - branch to instruction 2 and repeat instructions with inverted the address sequencing • Repeat #2 - branch to instruction 2 and repeat instructions with inverted the write data sequencing. • Repeat #3 -branch to instruction 2 and repeat instructions with inverted the address and write data sequencing.

Table C-23. Description of LVBitSurroundDisturb Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
3	2	Repeat #1	Repeat #1	AddressRegisterB, decrement column address	WD	Write	Write D-data at the location addressed by AddressRegisterB. Decrement AddressRegisterB column address by one.
	3	Repeat #1	Repeat #1	AddressRegisterA	R$\overline{D}$	Read	Read $\overline{D}$-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to counterA endcount (twice).
	4	Repeat #1	Repeat #1	AddressRegisterB, decrement row address	WD	Write	Write D-data at the location addressed by AddressRegisterB. Decrement AddressRegisterB row address by one.
	5	Repeat #1	Repeat #1	AddressRegisterA	R$\overline{D}$	Read	Read $\overline{D}$-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice). Repeat instructions 2-5 three times as follows with RepeatLoopB: • Repeat #1 - branch to instruction 2 and repeat instructions with inverted the address sequencing • Repeat #2 - branch to instruction 2 and repeat instructions with inverted the write data sequencing. • Repeat #3 -branch to instruction 2 and repeat instructions with inverted the address and write data sequencing.

Table C-23. Description of LVBitSurroundDisturb Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
4	2	Repeat #1	Repeat #2	AddressRegisterB, increment column address	W$\overline{D}$	Write	Write $\overline{D}$-data at the location addressed by AddressRegisterB. Increment AddressRegisterB column address by one.
	3	Repeat #1	Repeat #2	AddressRegisterA	R$\overline{D}$	Read	Read $\overline{D}$-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice).
	4	Repeat #1	Repeat #2	AddressRegisterB, increment row address	W$\overline{D}$	Write	Write $\overline{D}$-data at the location addressed by AddressRegisterB. Increment AddressRegisterB row address by one.
	5	Repeat #1	Repeat #2	AddressRegisterA	R$\overline{D}$	Read	Read $\overline{D}$-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice). Repeat instructions 2-5 three times as follows with RepeatLoopB: • Repeat #1 - branch to instruction 2 and repeat instructions with inverted the address sequencing • Repeat #2 - branch to instruction 2 and repeat instructions with inverted the write data sequencing. • Repeat #3 -branch to instruction 2 and repeat instructions with inverted the address and write data sequencing.

Table C-23. Description of LVBitSurroundDisturb Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
5	2	Repeat #1	Repeat #3	AddressRegisterB, decrement column address	W$\overline{D}$	Write	Write $\overline{D}$-data at the location addressed by AddressRegisterB. Decrement AddressRegisterB column address by one.
	3	Repeat #1	Repeat #3	AddressRegisterA	R$\overline{D}$	Read	Read $\overline{D}$-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice).
	4	Repeat #1	Repeat #3	AddressRegisterB, decrement row address	W$\overline{D}$	Write	Write $\overline{D}$-data at the location addressed by AddressRegisterB. Decrement AddressRegisterB row address by one.
	5	Repeat #1	Repeat #3	AddressRegisterA	R$\overline{D}$	Read	Read $\overline{D}$-data at the location addressed by AddressRegisterA. Increment CounterA. Branch back to Instruction 2 and repeat until CounterA counts from zero to CounterA endcount (twice). Repeat instructions 2-5 three times as follows with RepeatLoopB: • Repeat #1 - branch to instruction 2 and repeat instructions with inverted the address sequencing • Repeat #2 - branch to instruction 2 and repeat instructions with inverted the write data sequencing. • Repeat #3 -branch to instruction 2 and repeat instructions with inverted the address and write data sequencing.

Table C-23. Description of LVBitSurroundDisturb Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
6	6	Repeat #1	-	Address RegisterA, increment min to max, fast rows	-	NoOperation	Increment AddressRegisterA and copy the result to AddressRegisterB. Branch back to Instruction 1 and re-run phases 1 through 5 for all AddressRegisterA row and column addresses.
7	7	Repeat #1	-	Address RegisterA, bank addresses, min to max	-	NoOperation	Increment the bank address for AddressRegisterA and copy the result to AddressRegisterB. Branch back to Instruction 0 and re-run phases 1 through 7 for all bank addresses. Repeat instructions 0-7 once as follows with RepeatLoopA: • Repeat #1 - branch to instruction 0 and repeat instructions with inverted the write and expect data sequencing

Example Algorithm Wrapper

illustrates the Algorithm wrapper for the example memory.

Figure C-13. LVBitSurroundDisturb Example Algorithm Wrapper

```
MemoryOperationsSpecification {
  Algorithm (LVBitSurroundDisturb) {
   TestRegisterSetup {
      operation_set_select : Sync;
      LoadCounterA_EndCount : 1;
      AddressGenerator {
         AddressRegisterA {
            z_carry_in : none;
            y1_carry_in : x1_carry_out;
            x1_carry_in : none;
         }
      }
      DataGenerator {
         load_write_data: all_zero;
         load_expect_data: all_zero;
      }
   }
   MicroProgram {
      Instruction (WRITE_BACKGROUND) {
         operation_select : Write;
         AddressCommands {
            address_select : select_a;
            x1_address : increment;
            y1_address : increment;
         }
         DataCommands {
            write_data : data_reg;
         }
         NextConditions {
            x1_end_count : on;
            y1_end_count : on;
         }
      }
      Instruction (OFFSET_AWAY_CELL) {
         operation_select : NoOperation;
         AddressCommands {
            address_select : select_b;
            x1_address : decrement;
            y1_address : decrement;
         }
         NextConditions {
         }
      }
      Instruction (WRITE_AWAY_CELL1) {
         operation_select : Write;
         AddressCommands {
            address_select : select_b;
            y1_address : increment;
         }
         DataCommands {
            write_data : inverse_data_reg;
         }
         NextConditions {
         }
      }
```

```
Instruction (READ_HOME_CELL1) {
    operation_select : Read;
    CounterCommands {
        counter_a : increment;
    }
    AddressCommands {
        address_select : select_a;
    }
    DataCommands {
        expect_data : data_reg;
    }
    branch_to_instruction : WRITE_AWAY_CELL1;
    NextConditions {
        counter_a_end_count : on;
    }
}
Instruction (WRITE_AWAY_CELL2) {
    operation_select : Write;
    AddressCommands {
        address_select : select_b;
        x1_address : increment;
    }
    DataCommands {
        write_data : inverse_data_reg;
    }
    NextConditions {
    }
}
Instruction (READ_HOME_CELL2) {
    operation_select : read;
    CounterCommands {
        counter_a : increment;
    }
    AddressCommands {
        address_select : select_a;
    DataCommands {
        expect_data : data_reg;
    }
    branch_to_instruction : WRITE_AWAY_CELL2;
    NextConditions {
        counter_a_end_count : on;
        RepeatLoopB {
            branch_to_instruction: WRITE_AWAY_CELL1;
            Repeat1 {
                enable : on;
                address_sequence : inverse;
            }
            Repeat2 {
                enable : on;
                write_data_sequence : inverse;
            }
            Repeat3 {
                enable : on;
                address_sequence : inverse;
                write_data_sequence : inverse;
            }
        }
    }
```

```
        }
        Instruction (MOVE_HOME_CELL) {
            operation_select : NoOperation;
            AddressCommands {
                address_select : Select_A_Copy_To_B;
                x1_address : increment;
                y1_address : increment;
            }
            branch_to_instruction : OFFSET_AWAY_CELL;
            NextConditions {
                x1_end_count: on;
                y1_end_count: on;
            }
        }
        Instruction (GOTO_NEXT_BANKADDRESS) {
            operation_select : NoOperation;
            AddressCommands {
                address_select : select_a_copy_to_b;
                z_address : increment;
            }
            branch_to_instruction : WRITE_BACKGROUND;
            NextConditions {
                z_end_count : on;
                RepeatLoopA {
                    branch_to_instruction: WRITE_BACKGROUND;
                    Repeat1 {
                        enable : on;
                        write_data_sequence: inverse;
                        expect_data_sequence: inverse;
                    }
                }
            }
        }
    }
  }
}
```

Fault Coverage

The faults detected by the LVBitSurroundDisturb algorithm are identified in Table C-1.

Specification

To test SRAMs using the LVBitSurroundDisturb algorithm, specify Algorithm LVBitSurroundDisturb in the memory TCD file or the Controller(id)/AdvancedOptions of the DftSpecification or the related entry in the DefaultsSpecification.

Usage Conditions

The following usage conditions apply to the LVBitSurroundDisturb algorithm:

- Operations named Write, Read, ReadModifyWrite must exist or be mapped to another operation in OperationSet of the memory TCD file.

- For soft programmable controller usage, the DftSpecification/MemoryBist/Controller/ AlgorithmResourceOptions/soft_instruction_count property must be set to 8 if the Z address bank is defined, otherwise, it should be 7.

Appendix D
Memory Fault Types

This appendix describes the various memory fault models that have been developed to represent the effects of common memory defect mechanisms. This appendix contains the following information for each fault model:

- A general description of the fault behavior

- The test sequence needed to detect the fault

- Test algorithms provided by membistGenerate that cover the fault

This appendix describes the following fault types:

Address Decoder Faults

This model encompasses faults in the address decoder logic. Three different faulty behaviors are possible:

- ADa: a certain address results in no cell being accessed.

- ADb: a certain address simultaneously accesses multiple cells.

- ADc: a certain cell can be accessed by multiple addresses.

It has been shown that the above address decoder faults can in fact be mapped to faults in the memory cell array. Therefore covering the memory cell array faults results in these faults also being covered.

Detection Requirements

All March style tests detect an address decoder fault.

Algorithms

Address decoder faults are covered by the algorithms indicated in Table C-1.

Bit/Group/Global Write Enable Faults

These defects on the write enable ports—bit, group, or global—are stuck-active and short between the bit/byte write enable ports and the global write enable ports.

Detection Requirements

Stuck active faults on Bit/Group/Global Write Enables are detected by first writing a data value to memory with these inputs set to their active value, followed by an attempt at writing the opposite data value with these inputs set to their inactive value, then verifying that the first data value is still present.

Shorts between Bit/Group Write Enables are detected by performing write operations with inputs corresponding to an odd and even index of the bus set to opposite values. Similarly, shorts between Bit/Group Write Enables and the Global Write Enable are detected by performing write operations with the Bit/Group Write Enables and Global Write Enable set to opposite values.

Bit/Group/Global write enable faults are covered by the algorithms indicated in Table C-1.

Access Transistor Current Leakage Faults

Excessive leakage current from the access transistors into the bitlines and datalines can cause the content of a memory cell to be incorrectly read. The unusually high leakage current of the

access transistors between the data lines, as well as the leakage current of the cells on the bitlines, could reduce the differential voltage between the bitlines that can cause the differential amplifier to read the cell incorrectly. The worst case is when all the cells on the column have the same value except the pivot cell.

Detection Requirements

Detecting the excessive leakage current faults can best be achieved by sensitizing the worst case condition, such as when the pivot cell holds a value "1" while all cells in the same column hold the opposite value "0". The inverse is also true.

The detection of leakage faults does not depend on the content of the cells in adjacent columns.

Algorithm

Access transistor current leakage faults are covered by the algorithms indicated in Table C-1.

Data Retention Faults

A data retention fault is one where a cell loses its contents over time without being accessed. This is primarily a DRAM cell fault mechanism, resulting from an abnormally large leakage current. Leakage can occur between a cell and the substrate or between two cells. A retention fault can also occur in SRAM cells as the result of a defective pull-up device in the cell.

Detection Requirements

These faults are detected by writing a physical checkerboard pattern in memory, pausing for some time, and then reading it back. The process is repeated for the inverse checkerboard pattern. The SMarchCHKB* library algorithms are automatically split into three test patterns to facilitate the application on the tester. Always use one of these algorithms as the default algorithm to enable this feature.

Algorithms

Data retention faults are covered by the algorithms indicated in Table C-1.

Data Path Shorts

Data path shorts occur between even-numbered and odd-numbered elements of the data path, that are assumed to be physically adjacent. The shorts can be anywhere between the memory inputs or outputs, and the column multiplexers.

Detection Requirements

These faults are detected by writing and reading a checkerboard-like pattern (101010...) to all locations in the first row of every bank.

The Data Path shorts are covered by the algorithms indicated in Table C-1.

Destructive Read Faults

This fault can cause the contents of a memory cell to be changed during a read access. However, the value read after a first read access could be the correct value. This fault can occur as a result of a resistive defect in the pull-down path of a memory cell, for example.

This fault can also be called a deceptive destructive read fault.

Detection Requirements

To detect a destructive read fault, the cell under test must be initialized and then read multiple times in consecutive clock cycles. The minimum number of reads is two. The sequence must be repeated for both logic 1 and logic 0. Siemens EDA library algorithms use the Read operation to detect this fault, for which the default number of reads is two. However, some memories may require the number of reads to be increased, which is easily done by customizing the Read operation.

An example is shown below, where the original operation is augmented by four cycles. Note that the number of additional strobes should be kept to a minimum to avoid slowing down the diagnosis process because of the multiple identical failures being reported. In fact, the last strobe can be omitted in all SMarch* algorithms as the fault is detected in a subsequent algorithm phase.

```
Operation (Read) {
  Cycle {
    select              : on;
    write_enable        : off;
    read_enable         : on;
    output_enable       : on;
    ConcurrentPortSignals {
      read_enable       : on;
    }
  }
  Cycle {
      strobe_data_out   : on;
  }
  Cycle {        }
  Cycle {        }
  Cycle {        }
  Cycle {
      strobe_data_out   : on;    // Optional
  }
}
```

Algorithm

The destructive read fault defect is detected by the algorithms indicated in Table C-1.

Dynamic Coupling Faults

In this fault model, reading or writing a logic 0 or logic 1 to one memory cell (the coupling cell) forces the value in another cell (the base cell) to either a logic 0 or logic 1. A total of four dynamic coupling faults are therefore possible between two cells:

- dyCFa: reading or writing a 0 in the coupling cell forces a 0 in the base cell.

- dyCFb: reading or writing a 0 in the coupling cell forces a 1 in the base cell.

- dyCFc: reading or writing a 1 in the coupling cell forces a 0 in the base cell.

- dyCFd: reading or writing a 1 in the coupling cell forces a 1 in the base cell.

The two coupled cells can appear anywhere in the memory array.

Detection Requirements

To detect a dyCFa fault, the following sequence of events must occur:

1. The base cell must be storing a logic 1 and the coupling cell a logic 0.

2. The coupling cell must be read or a logic 0 value written to it.

3. The base cell must be read before any value is written to it.

To detect a dyCFb fault, the following sequence of events must occur:

1. Both the base and coupling cells must be storing a logic 0.

2. The coupling cell must be read or a logic 0 value written to it.

3. The base cell must be read before any value is written to it.

To detect a dyCFc fault, the following sequence of events must occur:

1. Both the base and coupling cells must be storing a logic 1.

2. The coupling cell must be read or a logic 1 value written to it.

3. The base cell must be read before any value is written to it.

To detect a dyCFd fault, the following sequence of events must occur:

1. The base cell must be storing a logic 0 and the coupling cell a logic 1.

2. The coupling cell must be read or a logic 1 value written to it.

3. The base cell must be read before any value is written to it.

Algorithms

Dynamic coupling faults are covered by the algorithms indicated in Table C-1.

Idempotent Coupling Faults

In this fault model, a logic 0 to logic 1 or logic 1 to logic 0 transition in one memory cell (the coupling cell) forces the value in another cell (the base cell) to either a logic 0 or logic 1. A total of four idempotent coupling faults are therefore possible between two cells:

- IdCFa: a 0 to 1 transition in the coupling cell forces a 0 in the base cell.

- IdCFb: a 0 to 1 transition in the coupling cell forces a 1 in the base cell.

- IdCFc: a 1 to 0 transition in the coupling cell forces a 0 in the base cell.

- IdCFd: a 1 to 0 transition in the coupling cell forces a 1 in the base cell.

The two coupled cells can appear anywhere in the memory array.

Detection Requirements

To detect an IdCFa fault, the following sequence of events must occur:

1. The base cell must be storing a logic 1 and the coupling cell a logic 0.

2. A logic 1 must be written to the coupling cell.

3. The base cell must be read before any value is written to it.

To detect an IdCFb fault, the following sequence of events must occur:

1. Both the base and coupling cells must be storing a logic 0.

2. A logic 1 must be written to the coupling cell.

3. The base cell must be read before any value is written to it.

To detect an IdCFc fault, the following sequence of events must occur:

1. Both the base and coupling cells must be storing a logic 1.

2. A logic 0 must be written to the coupling cell.

3. The base cell must be read before any value is written to it.

To detect an IdCFd fault, the following sequence of events must occur:

1. The base cell must be storing a logic 0 and the coupling cell a logic 1.

2. A logic 0 must be written to the coupling cell.

3. The base cell must be read before any value is written to it.

Algorithms

Idempotent coupling faults are covered by the algorithms indicated in Table C-1.

Inversion Coupling Faults

In this fault model, a logic 0 to logic 1 or logic 1 to logic 0 transition in one memory cell (the coupling cell) inverts the value in another cell (the base cell). Two inversion coupling faults are therefore possible between two cells:

- InCFa: a 0 to 1 transition in the coupling cell inverts the value in the base cell.

- InCFb: a 1 to 0 transition in the coupling cell inverts the value in the base cell.

The two coupled cells can appear anywhere in the memory array.

Note that for each of the above faults, both inversions of the base cell must occur. If only one occurs, then the fault reduces to one of the idempotent coupling faults.

Detection Requirements

To detect an InCFa fault, the following sequence of events must occur:

1. The coupling cell must be storing a logic 0 and the value stored in the base cell must be known.

2. A logic 1 must be written to the coupling cell.

3. The base cell must be read before any value is written to it.

To detect an InCFb fault, the following sequence of events must occur:

1. The coupling cell must be storing a logic 1 and the value stored in the base cell must be known.

2. A logic 0 must be written to the coupling cell.

3. The base cell must be read before any value is written to it.

Algorithms

Inversion coupling faults are covered by the algorithms indicated in Table C-1.

Memory Select Faults

This defect is a result of an internal memory select stuck active fault.

Detection Requirements

These faults are detected by the following sequence:

1. Write 1 to a first cell

2. Write 0 to a second cell in the same column

3. read 1 from the first cell

4. read 0 from the second cell

5. Attempt to read 1 from the first cell, while de-asserting the select input. The result of the read operation is 0 for a fault-free memory.

6. Attempt to write 0 to the first cell, while de-asserting the select input

7. Read 1 from the first cell

8. Attempt to read 0 from the second cell, while de-asserting the select input. The result of the read operation is 1 for a fault-free memory.

9. Attempt to write 1 to the second cell, while de-asserting the select input

10. Read 0 from the second cell

This sequence is applied to the first two rows of every bank. The sequence assumes that the memory holds the last value read while asserting the select input. If this assumption is not true, MemoryHoldWithInactiveSelect must be set to off in the memory library file to disable the test.

Algorithm

The memory select stuck active defect is detected by the algorithms indicated in Table C-1.

Multi-Port Interference Faults

This defect is primarily a result of high resistance ground connections to one of the N-channel source terminals in multi-port memories.

Detection Requirements

This type of fault can be detected by turning ON all access transistors connected to the drain of the N-channel and observing the result of the read on one of the ports. This can be done by reading the same row simultaneously from both ports. This maximizes the chances to detect the defect because the N-channel needs to transfer more charge from the bit lines.

Because single-port and multi-port memories can be tested during the same step, the same algorithm can be used. The only difference is that read operations at specific locations must occur concurrently on the inactive port while the algorithm is applied to the active port. The Siemens EDA library operation sets enable such operations, called shadow or concurrent operations, on inactive ports of multi-port memories. The detection of defective ground connections is maximized in the same steps as for the single-port bitline coupling faults.

Algorithm

Multi-port interference faults are covered by the algorithms indicated in Table C-1.

Multiport Synchronous Bitline Coupling Faults

This defect is mainly a result of slightly slower access transistors on one port. This causes the data read from one port to be coupled to the other ports' bitlines.

Detection Requirements

These faults are primarily detected by performing a read and a write simultaneously from different ports, in the same column, using opposite data values. The data values in the cells adjacent to the one being read has secondary detection effect.

Algorithm

Multi-port synchronous bitline coupling faults are covered by the algorithms indicated in Table C-1.

Neighborhood Pattern Sensitive Faults

A neighborhood pattern sensitive fault deals with memory faults that occur due to specific patterns in the neighborhood. The definition of neighborhood is not unique. Different neighborhoods are considered for specific faults in the Siemens EDA library algorithms.

A few examples of neighborhoods:

- **single-port bitline coupling faults** — The two cells adjacent to the cell under test located in the same row.

- **Access transistor leakage current faults** — All cells in the same column as the cell under test.

Custom algorithms might be required to test faults that need to consider different neighborhoods.

Parametric Faults

Parametric faults can be classified into DC and AC parametric faults. DC parametric faults encompass time independent voltage/current abnormalities. These faults include:

- high power consumption

- high current leakage

- high or low input voltage thresholds

AC parametric faults encompass time dependent voltage abnormalities. These faults include:

- output slow to rise or slow to fall times

- large input setup and hold times

- large output delay times

- large memory cycle times

Detection Requirements

Applying test patterns at system cycle speeds results in the detection of the AC parametric faults. This is one of the great advantages of the memory BIST approach. Detecting other defects, such as current leakage within a cell causing data retention faults, requires an application of pauses at specific points in the algorithm.

Read Disturb Faults

This model encompasses defects that cause a memory cell to change value (logic 0 or logic 1) when another cell is read. This normal read operation on one memory cell causes another cell to change value (logic 0 to logic 1 or logic 1 to logic 0).

Some definitions of read disturb faults found in literature also include the case where the aggressor and victim cells are the same. In Tessent MemoryBIST, these faults are defined as Destructive Read Faults.

Detection Requirements

These faults are typically detected by the same algorithms used to detect inversion coupling, as the write operation at the aggressor cell is usually preceded by a read operation.

Algorithms

Read disturb faults are covered by the algorithms indicated in Table C-1.

Read Enable Faults

These defects are a result of stuck-active faults of the memory read enable signal, including paths internal to the memory.

Detection Requirements

These faults are detected by the following sequence:

1. Write 1 to a first cell

2. Write 0 to a second cell in the same column

3. Read 1 from the first cell

4. Attempt to read 0 from the second cell, while de-asserting the read enable input. The result of the read operation is 1 for a fault-free memory.

5. Read 0 from the second cell

6. Attempt to read 1 from the first cell, while de-asserting the read enable input. The result of the read operation is 0 for a fault-free memory.

This sequence is applied to the first two rows of every bank. The sequence assumes that the memory holds the last value read while asserting the read enable input. If not, DataHoldWithInactiveReadEnable must be set to off in the memory library file to disable the test.

Algorithm

The read enable stuck active defects are detected by the algorithms indicated in Table C-1.

Single Port Bitline Coupling Faults

Signal coupling between bitlines of adjacent columns causes read errors when accessing cells with minor manufacturing defects. Worst-case coupling occurs when the data value of the two adjacent cells in the same row is chosen to interfere with the read operation. The data values depend on factors related to the memory layout namely bit grouping and cell orientation.

Detection Requirements

The fault is detected when reading the reference cell with adjacent cells containing specific values. All eight combinations of the three cells (reference and two neighbors) are applied, to take into account the mirroring of bit lines.

Algorithm

Single port bitline coupling faults are covered by the algorithms indicated in Table C-1.

Stuck-At Faults

In this model, a memory cell is permanently forced to a logic 0 (stuck-at-0 fault) or logic 1 (stuck-at-1 fault) value, irrespective of any value written to the cell. This is the most common fault, but also the easiest to detect.

Detection Requirements

A logic 1 must be read from the cell under test to detect the stuck-at-0 fault while a logic 0 must be read to detect the stuck-at-1 fault.

Algorithms

Stuck-at faults are covered by the algorithms indicated in Table C-1.

Stuck-Open Faults

In this model, a memory cell can't be accessed. As an example, this could be due to an open word line.

Detection Requirements

Stuck-open faults are detected as stuck-at faults when the sense amplifier does not contain a data latch. When a data latch is present, the following sequence must be applied:

1. The cell under test must be storing a logic x (either 0 or 1)

2. The inverse logic value must be written into the cell

3. The cell must be read and the value compared

Algorithms

Stuck-open faults are covered by the algorithms indicated in Table C-1.

Transition Faults

In this model, a memory cell fails to undergo a transition from a logic 0 to a logic 1 value (up transition fault) or from a logic 1 to a logic 0 value (down transition fault). These faults are special cases of stuck-at faults because of the fact that once the non-faulty transition occurs, the faulty cell can no longer transition and hence manifests stuck-at behavior.

In some cases, however, a coupling fault with another cell (see Idempotent Coupling Faults, Inversion Coupling Faults, and Dynamic Coupling Faults) can flip the value of the cell, thereby masking the stuck-at behavior. For this reason, transition faults must be considered separately from stuck-at faults.

Detection Requirements

To detect an up transition fault, the following sequence of events must occur:

1. The cell under test must be storing a logic 0.

2. A logic 1 must be written into the cell.

3. The cell must be read before a logic 0 is written to it.

To detect a down transition fault, the following sequence of events must occur:

1. The cell under test must be storing a logic 1.

2. A logic 0 must be written into the cell.

3. The cell must be read before a logic 1 is written to it.

Algorithms

Transition faults are covered by the algorithms as indicated in Table C-1.

Write Disturb Faults

Write disturb faults cause a memory cell to change value when performing a non-transition write to this cell. Some definitions of write disturb faults found in literature also include the case where the aggressor and victim cells are different.

In Tessent MemoryBIST, these faults are defined as Inversion Coupling Faults or Idempotent Coupling Faults.

Detection Requirements

These faults are detected by writing a value to a cell, repeating this operation at least once, then reading the cell to verify that the correct value is present. This sequence must be repeated for both the logic 0 and logic 1 value.

All five algorithms of the SMarch* family, plus the LVMarchLA algorithm detect these faults. For cases where the write needs to be repeated more than once, the TessentHammerWriteFastX and TessentHammerReadFastY algorithms are required. These algorithms are available from the */lib/technology/memory_bist/algo* directory in the release tree.

Algorithms

Write disturb faults are covered by the algorithms indicated in Table C-1.

Write Recovery Faults

A write recovery fault occurs when a value is read from a cell just after the opposite value has been written to a cell along the same column and the bitline precharge has not been performed correctly.

The resulting faulty behavior is that reading from cell A just after writing to cell B results in reading the value written to cell B.

Detection Requirements

To detect a write recovery fault, the following sequence of events must occur:

For a given column address:

1. Write 1 to a first cell in row address A

2. Write 0 to a second cell in row address B

3. Read 1 from the first cell

4. Repeat steps 1-3 with different data values

Algorithms

Write recovery faults are covered by the algorithms indicated in Table C-1.

A large portion, if not the majority, of a complete memory test is spent on static retention testing. For chips containing several embedded SRAMs, performing the static retention test on all of the SRAMs in parallel or in groups of controllers is highly beneficial. This appendix describes how to accomplish this when the embedded SRAMs are tested by different memory BIST controllers or when they are tested sequentially within different memory BIST controller steps. This appendix also describes how to sequentially perform parallel static retention testing on groups of controllers.

Before running parallel static retention testing (PSRT), we recommend that you first run BIST in HWDefault or RunTimeProg mode to provide the highest level of fault coverage. Then run the sub-phases of PSRT that specifically target testing of the cell retention time. When PSRT is enabled for a multi-port memory, only one R port and one W port or one RW port is enabled. In this case, PSRT does not run BIST on all ports. Performing PSRT from all ports is unnecessary; this only adds test time for the pause. Depending on the specified pause length, the additional test time could be significant and does not increase the coverage of the test.

Parallel Static Retention Testing Limitations

Parallel Static Retention Testing is supported by the following library algorithms:

- SMarch

- SMarchCHKB

- SMarchCHKBci

- SMarchCHKBcil

- SMarchCHKBvcd

Handling Multiple Controllers

The key to performing the static retention test in parallel to all SRAMs that are being tested by different memory BIST controllers is to synchronize the controllers. To accomplish this, the applicable library algorithms can be made to support three test sub-phases:

- **start_to_pause** — In this sub-phase, the controller loads a background pattern into all memories and stops. The background pattern is all zeros for the simple SMarch algorithm and a checkerboard pattern for the remaining algorithms.

- **pause_to_pause** — In this sub-phase, the controller reads the background pattern and loads an inverse background pattern into all memories and stops. The inverse background pattern is all ones for the simple SMarch and an inverse checkerboard pattern for the remaining algorithms.

- **pause_to_end** — In this sub-phase, the controller reads the inverse background pattern and exits.

Sequence for Test Sub-Phases

You should apply only one of the test sub-phases at a time. Perform PSRT in the following sequence:

1. Enable all controllers in parallel and make them each run the start_to_pause sub-phase.

2. When all controllers complete Step 1, apply the first static retention test pause.

3. Enable all controllers in parallel and make them each run the pause_to_pause sub-phase.

4. When all controllers complete Step 3, apply the second static retention test pause.

5. Enable all controllers in parallel and make them each run the pause_to_end sub-phase.

Figure E-1 shows the sequence for applying the test sub-phases.

Figure E-1. Parallel Static Retention Test Sequence

Sample PatternsSpecification Syntax for Test Sub-Phases

Using PatternsSpecification, you can create a testbench that contains the proper sub-phase and retention pause sequences.

The following is a sample that illustrates the sequence for the two-controller example in Figure E-1.

```
TestStep (0) {
  MemoryBist {
    parallel_retention_time: 10ms;
    Controller (A) {
    ...
    }
    Controller (B) {
    ...
    }
  }
}
```

When generating the pattern, process_patterns_specification automatically creates the three sub-phases and two retention pauses. The resulting pattern is equivalent to manually specifying the test sequence as shown in the following example:

```
TestStep (t1) {
  MemoryBist {
    AdvancedOptions {
      retention_test_phase: start_to_pause;
      preserve_bist_inputs : on;
    }
    Controller (A) {
    }
    Controller (B) {
    }
  }
}
ProcedureStep (p1) {
  wait_time : 10ms;
}
TestStep (t2) {
  MemoryBist {
    AdvancedOptions {
      retention_test_phase: pause_to_pause;
      preserve_bist_inputs : on;
    }
    Controller (A) {
    }
    Controller (B) {
    }
  }
}
ProcedureStep (p2) {
  wait_time : 10ms;
}
TestStep (t3) {
  MemoryBist {
    AdvancedOptions {
      retention_test_phase: pause_to_end;
    }
    Controller (A) {
    }
    Controller (B) {
    }
  }
}
```

Handling Memories Tested Sequentially

The three sub-phase approach, described in the "Handling Multiple Controllers" section, can also be used to provide a parallel static retention test of memories tested sequentially; that is, tested in different memory BIST controller steps. This approach is applicable because when a controller applies a given sub-phase, the controller applies the sub-phase within all steps before stopping.

Assume, for example, that Ctrl A in Figure E-1 contains three steps. As shown in Figure E-1, when this controller is instructed to apply the sub-phase start_to_pause, the controller applies this sub-phase sequentially within each of its three steps before stopping. As a result, all memories in all three steps contain background patterns and are ready for the first static retention test pause.

You can deduce from the above scenario that a parallel static retention test can be applied to any combination of SRAMs tested by different controllers and in different steps within these controllers.

Testing Controllers in Groups

PSRT also can be performed sequentially on subsets of controllers. This approach reduces the power consumption during PSRT by limiting the number of controllers that are executing at the same time. During PSRT, the controllers are organized into controller groups. All controllers that belong to the same controller group are run in parallel.

The three sub-phases described in the "Handling Multiple Controllers" section are applied in the following sequence:

1. start_to_pause

2. pause_to_pause

3. pause_to_end

Within a test sub-phase, each controller group is enabled sequentially. All controllers in a group are instructed to apply the same sub-phase. When the last controller group completes execution, the next PSRT sub-phase is performed. The controller groups are enabled in the same sequence for all three sub-phases.

Sequence for Test Sub-Phases

The sequence for applying PSRT on four controllers organized into two controller groups is described in this section. Controllers A and B are in Group 1. Controllers C and D are in Group 2.

PSRT is achieved using the following sequence shown in Figure E-2 and described below:

1. Enable Ctrl A and Ctrl B in parallel to run the start_to_pause sub-phase.

2. Enable Ctrl C and Ctrl D in parallel to run the start_to_pause sub-phase.

3. When Ctrl C and Ctrl D complete Step 2, apply the first static retention test pause.

4. Enable Ctrl A and Ctrl B in parallel to run the pause_to_pause sub-phase.

5. Enable Ctrl C and Ctrl D in parallel to run the pause_to_pause sub-phase.

6. When Ctrl C and Ctrl D complete Step 5, apply the second static retention test pause.

7. Enable Ctrl A and Ctrl B in parallel to run the pause_to_end sub-phase.

8. Enable Ctrl C and Ctrl D in parallel to run the pause_to_end sub-phase

Figure E-2. Parallel Static Retention Test with Controller Groups

Sample PatternsSpecification Syntax for Test Sub-Phases

Using PatternsSpecification, you can create a testbench that contains the sequence described in the previous section.

The following sample illustrates the sequence for the four-controller example in Figure E-2:

```
TestStep (0) {
  MemoryBist {
    parallel_retention_time: 2ms;
    Controller (A) {
      parallel_retention_group: 1;
    }
    Controller (B) {
      parallel_retention_group: 1;
    }
    Controller (C) {
      parallel_retention_group: 2;
    }
    Controller (D) {
      parallel_retention_group: 2;
    }
  }
}
```

The generated testbench always applies all three sub-phases, and the retention_test_phase property should not be specified. The same retention pause of 2ms is applied between sub-phases. Each controller group consists of all controllers with the same group number.

_____ **Note** _____
The specified value for the parallel_retention_group property is a label for the controller group. The controller group execution order within a sub-phase is determined by the appearance order of the group numbers.

Appendix F
Memory BIST Physical Mapping Examples

This appendix contains information about address data mapping in memory BIST and provides several examples.

This appendix covers the following topics:

Memory Cores

A memory core is built of memory cells that are arranged in one or more arrays (or blocks or banks). The arrays might further be divided into sub-arrays (or sub-blocks).

A memory core contains *w* words of *b* bits wide each, which are physically configured in a matrix of *nrow* rows by *ncol* columns. Memory wordlines, each of them selecting a row of memory cells, are in the same direction as the rows. The bitlines are in the same direction as columns. The bitlines are used to write data to the memory cells and read data from them.

w x *b* = *nrow* x *ncol*

In a sample memory core, there are one or more words on each row. The number of columns is a whole multiple of the number of bits (*b*) per word (*w*). When there is more than one word per row, the corresponding bits from all the words on a row get multiplexed onto a memory's dataline. For example, the 0th bit of all the words on the same row gets multiplexed onto Dout 0 of the memory.

The number of words per row is denoted as *ncolmux* (which is *ncol* / *b*). The *w* words are decoded by the *nadd* address inputs, where:

$2^{nadd} >= w > 2^{(nadd-1)}$

The address inputs are divided into *nra* row address inputs and *nca* column address inputs where:

$2^{nra} >= nrow > 2^{(nra-1)}$

$2^{nca} >= ncolmux > 2^{(nca-1)}$

Example 1

Consider a 1kx32, 1-port memory, with 4:1 column multiplexing. This memory has the following:

```
w = 1024, b = 32, ncolmux = 4
```

Therefore,

```
nrow = w / ncolmux = 1024 / 4 = 256,
ncol = b x ncolmux = 32 x 4 = 128,
nadd = log₂(w)  = log₂(1024) = 10,
nra = log₂(nrow)  = log₂(256)  = 8, and
nca = log₂(ncolmux)  = log₂(4)  =2.
```

A simple architecture for this memory is presented in Figure F-1.

Figure F-1. Simple Architecture for the Memory in Example 1

The two-column address bits are denoted as C_0 and C_1, which are select lines for the column multiplexers.

───── **Note** ─────
Only the data out signals are shown. Similarly, the data in lines get de-multiplexed onto the bit lines or columns using the column addresses.

To conclude, the following list summarizes the terms defined in this section:

- w = number of words in a memory,

- b = number of bits in a word,

- $ncolmux$ = number of words per row = $ncol / b$,

- $nrow$ = number of rows in memory array = $w / ncolmux$,

- *ncol* = number of columns in memory array = *b* x *ncolmux,*

- *nadd* = number of addresses $<= \log_2 w,$

- *nra* = number of row addresses $<= \log_2 nrow$, and

- *nca* = number of column addresses $<= \log_2 ncolmux$.

> **Note**
> For $\log_2 w$, $\log_2 nrow$, and $\log_2 ncolmux$, when the value is not a whole number, round the value up to the next whole number.

Logical and Physical Mapping

When looking externally at a memory, the words are stored consecutively with respect to the address values, and the data bits within each word are stored in the order of their sequential numbering.

This arrangement is called logical mapping of a memory, which is illustrated in Figure F-2.

Figure F-2. Logical Memory Cell Arrangement

In Figure F-2, the designators r, c, and d are defined as below:

- $r_\#$: Logical row address bit

- $c_\#$: Logical column address bit

- $d_\#$: Logical data bit

> **Note**
> The number of rows or columns are shown as a full power of 2, which is not the case for all memories.

In many cases, the physical arrangement of memory cells does not correspond to the assumed logical arrangement, as a result of different memory design requirements. Some reasons for these differences include the following:

- To deal with small memory cells, memory designers sometimes fit the periphery cells in the pitch of more than one memory cell. For instance, they lay out sense amplifiers in the pitch of 4, 8, or more memory cells, so they place the corresponding bits of different words next to each other in the memory core, to be able to multiplex these corresponding bits onto one common sense-amplifier circuit.

- To balance the load on different address lines or (pre)decoded lines, memory designers sometimes scatter the wordlines or bitlines.

- To minimize the size of address and column decoders, as well as the length and hence propagation times of row and column select lines, memory arrays are typically divided into several sub-arrays.

- To increase the yield for larger memories, spare (redundant) rows or columns are often implemented, which typically disrupt the physical address sequence.

The Tessent MemoryBIST controller assumes the logical arrangement of cells when generating patterns, particularly the checkerboard patterns. To physically preserve the patterns, it becomes necessary to describe the mapping between the physical and logical cell arrangements.

The differences between the logical and physical cell arrangements are typically because of a *scrambling* of the rows, columns, and data bit lines. This scrambling can be described by a logical transformation or mapping between the address and data signals required to access the logical memory cell arrangement and those signals required to provide the same data pattern in the physical memory cell arrangement.

To simplify, the goal is to start with a pattern corresponding to a logical address and data and then by logical operations, transform the addresses and the data to a state or value that places the same pattern in the corresponding physical location. For example, if the intention is to write a 1010...10 pattern into the nth logical row and mth logical column, you need to describe the physical row and column address bits as a function of the logical row and column addresses in such a way that a 1010...10 pattern is written in the nth physical row and mth physical column.

The following examples illustrate this mapping:

> **Note**
> As mentioned before, the lowercase designators r, c, and d correspondingly denote the logical row address, column address, and the data bits. The uppercase designators R, C, and D denote the physical row address, column address, and data pins of the memory under test.

Example 2

Consider the row address mapping example shown in the figure below.

Figure F-3. Row Address Mapping Example

The column on the left in Figure F-3(a) represents a logical row sequence of memory words where the bit values represented within each box correspond to the values for the address signals r_2, r_1 and r_0, to access each row. The column on the right in Figure F-3(a) illustrates a typical scrambling of memory words within a column. As shown, every second pair of rows are flipped.

To access each successive row in the column now requires that the generated address values correspond to the bit values shown within each word in the second column of Figure F-2. These represent the true physical address bit values represented by R_2, R_1 and R_0. Therefore, to achieve the sequence you want, it is necessary to transform r_2, r_1 and r_0 into R_2, R_1 and R_0.

Figure F-3 illustrates the transformation rather simply. A single XOR gate is needed to generate R_0 from r_0 and r_1. The following simple equations express this transformation or mapping.

```
R₀ = r₀ xor r₁
R₁ = r₁
R₂ = r₂
```

$$R_0 = r_0 \text{ xor } r_1$$
$$R_1 = r_1$$
$$R_2 = r_2$$

The corresponding memory library file syntax is as follows:

```
PhysicalAddressMap {
    RowAddress[0] : r[0] xor r[1];
    RowAddress[1] : r[1];
    RowAddress[2] : r[2];
}
```

Example 3

Consider a memory that has *w* words and each word is 3 bits wide. There are four words on each row of the memory array; therefore, two column address bits are illustrated.

The arrangement of the memory columns and their addresses are depicted in Figure F-4.

Figure F-4. Column Address Mapping Example

In Figure F-4, the 3 bits of each word on every row are broken apart and corresponding bits of each word are placed next to each other according to the column address mapping shown in the figure. To specify this data bit ordering, the Core/Memory wrapper requires a statement such as:

```
BitGrouping: 1;
```

because each data bit is placed in the array by itself. The logical column address assumes a sequential order of the columns, as illustrated by the values for c_1c_0 on the topmost row above the memory array in Figure F-4. The following equations express the generation of column address pin values as a function of the logical column address bits, to physically achieve the wanted pattern:

```
C₀ = c₁
C₁ = c₀ xor c₁
```

The corresponding memory library file syntax is as follows:

```
PhysicalAddressMap {
   ColumnAddress[0] : c[1];
   ColumnAddress[1] : c[0] xor c[1];
}
```

_____ **Note** _____
In this example, a one-to-one logical to physical mapping is assumed for the row address bits and the states and sequence of the data bits.
_____

The default value for the BitGrouping property is the number of bits in a word or _b_. So, for this example, when BitGrouping is not specified it defaults to 3.

To further clarify the BitGrouping concept, consider the memory arrangement presented in Figure F-4. The memory of Figure F-4 is extended so that each word now has 6 bits, every 2 bits are placed next to each other. In this case, the following statement is in place:

```
BitGrouping: 2;
```

In Figure F-4, the multiplexers before the data output pins are drawn to better illustrate the column multiplexing.

Example 4

Consider a memory that has _w_ words, each word 4 bits wide.

Figure F-5 shows a small portion of the memory array with its data arrangement.

Figure F-5. Data Mapping Example

(a)

(b)

This data pattern is repeated throughout the entire array. Notice that unlike the sequential ordering of logical data bits, here the data bits are not placed in the array consecutively. In

addition, an inverted data value, denoted as $\overline{\text{Data}}$, is written into half of the physical memory cells. The following equations express this data transformation:

```
D0 = d0 xor r1
D2 = not d1 xor r1
D1 = d2 xor r1
D3 = not d3 xor r1
```

The corresponding memory library file syntax is as follows:

```
PhysicalDataMap {
     Data[0] : d[0] xor r[1];
     Data[2] : not d[1] xor r[1];
     Data[1] : d[2] xor r[1];
     Data[3] : not d[3] xor r[1];
}
```

To obtain the equation for memory data pin 0 (D_0), the logical data bit 0 (d_0) can be mapped to D_0 with one difference—when r_1 is 1, the data bit value is flipped. Furthermore, D_2, which is physically placed as the second column of the memory array, needs logical d_1, but here data is flipped when r_1 is 0. That is why the NOT of r_1 XORed with d_1 is used. A similar explanation is in place for D_1 and D_3.

Now, consider the data arrangement for a portion of the memory array shown in Figure F-5. The first 4 columns of the array are the same as the array in Figure F-4, while columns 4-7 are the mirror image of columns 0-3 with respect to the vertical line. When the column address c_0 is 1, the data arrangement is reversed. Hence, the equations above have been modified to express the data address pins for this memory as follows:

```
D0 = (d0 xor r1) xor c0
D2 = (not d1 xor r1) xor c0
D1 = (d2 xor r1) xor c0
D3 = (not d3 xor r1) xor c0.
```

The corresponding memory library file syntax is as follows:

```
PhysicalDataMap {
     Data[0] : d[0] xor r[1] xor c[0];
     Data[2] : not d[1] xor r[1] xor c[0];
     Data[1] : d[2] xor r[1] xor c[0];
     Data[3] : not d[3] xor r[1] xor c[0];
}
```

Example 5

Consider now the data mapping for the memory in the figure below.

This example illustrates portions of four columns of a memory array with a BitGrouping of 1.

Going from one column to the other, data inverts when c_0 is 1. Furthermore, the data pattern on each row is repeated while r_6 is 0. From row 64 onwards, however, the data pattern is inverted for the columns with c_0 equal to 0 because of bitline twist. Therefore, you can write the data mapping for D_0 in this example as:

```
D0 = d0 xor c0 xor (r6 and not c0)
```

The corresponding memory library file syntax is as follows:

```
PhysicalDataMap {
    Data[0] : d[0] xor c[0] xor (r[6] and not c[0]);
}
```

Figure F-6 only depicts a portion of the memory array to illustrate a data mapping using AND.

Figure F-6. Data Mapping Example That Incorporates AND

You can express virtually all address mappings as a series of XOR based equations similar to those in Example 2 and Example 3. Furthermore, you can express virtually all data mappings as a series of XOR and AND based equations, as illustrated in Example 5.

Note

You can use NOT for both types of mappings.

To simplify your memory logical and physical address and data mappings, it is suggested that, after understanding your memory architecture, you divide the mapping task into three parts:

- Row Address Mapping

- Column Address Mapping

- Data Mapping

Handle each part independently, as illustrated in Examples 2 through 5. Try to express your memory configuration in tables similar to the illustrations in this document.

The memory library file assumes a one-to-one mapping when you do not specify address and data mapping. For the mapping syntax in the memory library file, refer to the following:

- PhysicalAddressMap

- PhysicalDataMap

This appendix provides complete examples for implementing Built-In Self-Repair (BISR) using serial and parallel interfaces for a multi-segment memory with row and column repair. The memories used in the examples have row and column redundancy—a total of 4 spare rows and 4 spare columns. Each memory is divided into quadrants. The top hemisphere is the upper bank (Bank 0), and the bottom hemisphere is the lower bank (Bank 1). Each bank has two spare rows and two spare columns. For each bank, one redundant column is assigned to the left side and one to the right side.

This appendix covers the following topics:

Parallel BISR Interface Example

The following example outlines the memory library file for a memory with a parallel BISR interface.

Figure G-1 illustrates an example of a memory with a parallel BISR interface.

Figure G-1. Example Memory With Parallel BISR Interface

Figure G-2 shows an example of the memory library file syntax for a parallel repair interface. The example defines the repair analysis capability with built-in self-repair and soft repair to illustrate the example memory in Figure G-1.

Figure G-2. Memory Library File Sample Syntax for Parallel BISR Interface

```
// MemoryRepair ports for Bank B0
Port(B0_REN0) {
    Function: BisrParallelData;
    Direction: input;
}
Port(B0_RR0[3:0]) {
    Function: BisrParallelData;
    Direction: input;
}
Port(B0_RR1[3:0]) {
    Function: BisrParallelData;
    Direction: input;
}
Port(B0_REN1) {
    Function: BisrParallelData;
    Direction: input;
}
Port(B0_CRAR[3:0]) {
    Function: BisrParallelData;
    Direction: input;
}
Port(B0_CRAL[3:0]) {
    Function: BisrParallelData;
    Direction: input;
}
// MemoryRepair ports for Bank B1
Port(B1_REN0) {
    Function: BisrParallelData;
    Direction: input;
}
Port(B1_RR0[3:0]) {
    Function: BisrParallelData;
    Direction: input;
}
Port(B1_RR1[3:0]) {
    Function: BisrParallelData;
    Direction: input;
}
Port(B1_REN1) {
    Function: BisrParallelData;
    Direction: input;
}
Port(B1_CRAR[3:0]) {
    Function: BisrParallelData;
    Direction: input;
}
Port(B1_CRAL[3:0]) {
    Function: BisrParallelData;
    Direction: input;
}
```

```
RedundancyAnalysis {
    RowSegmentRange {
        SegmentAddress[0]: AddressPort(Address[10]);
    }
    RowSegment (Bank0){
        NumberOfSpareElements: 2;
        RowSegmentCountRange [1'b0:1'b0]; // Bank 0
        FuseSet {
            Fuse[3]: AddressPort(Address[9]);
            Fuse[2]: AddressPort(Address[8]);
            Fuse[1]: AddressPort(Address[7]);
            Fuse[0]: AddressPort(Address[0]);
        }
        PinMap {
            SpareElement {
                RepairEnable: B0_REN0;
                Fuse[0]: B0_RR0[0];
                Fuse[1]: B0_RR0[1];
                Fuse[2]: B0_RR0[2];
                Fuse[3]: B0_RR0[3];
            }
            SpareElement {
                RepairEnable: B0_REN1;
                Fuse[0]: B0_RR1[0];
                Fuse[1]: B0_RR1[1];
                Fuse[2]: B0_RR1[2];
                Fuse[3]: B0_RR1[3];
            }
        }
    }
    RowSegment (Bank1){
        NumberOfSpareElements: 2;
        RowSegmentCountRange [1'b1:1'b1]; // Bank 1
        FuseSet {
            Fuse[3]: AddressPort(Address[9]);
            Fuse[2]: AddressPort(Address[8]);
            Fuse[1]: AddressPort(Address[7]);
            Fuse[0]: AddressPort(Address[0]);
        }
        PinMap {
            SpareElement {
                RepairEnable: B1_REN0;
                Fuse[0]: B1_RR0[0];
                Fuse[1]: B1_RR0[1];
                Fuse[2]: B1_RR0[2];
                Fuse[3]: B1_RR0[3];
            }
            SpareElement {
                RepairEnable: B1_REN1;
                Fuse[0]: B1_RR1[0];
                Fuse[1]: B1_RR1[1];
                Fuse[2]: B1_RR1[2];
                Fuse[3]: B1_RR1[3];
            }
        }
    }
    ColumnSegment (Bank0_Left){
        RowSegmentCountRange [1'b0:1'b0]; // Bank0
```

```
            ShiftedIORange: QO[7:0]; // Left
            FuseSet {
                FuseMap[3:0]{
                    ShiftedIO(QO[0]): 4'b0001;
                    ShiftedIO(QO[1]): 4'b0010;
                    ShiftedIO(QO[2]): 4'b0011;
                    ShiftedIO(QO[3]): 4'b0100;
                    ShiftedIO(QO[4]): 4'b0101;
                    ShiftedIO(QO[5]): 4'b0110;
                    ShiftedIO(QO[6]): 4'b0111;
                    ShiftedIO(QO[7]): 4'b1000;
                }
            }
            PinMap {
                SpareElement {
                    FuseMap[0]: B0_CRAL[0];
                    FuseMap[1]: B0_CRAL[1];
                    FuseMap[2]: B0_CRAL[2];
                    FuseMap[3]: B0_CRAL[3];
                }
            }
        }
        ColumnSegment (Bank0_Right){
            RowSegmentCountRange [1'b0:1'b0]; // Bank 0
            ShiftedIORange: QO[15:8]; // Right
            FuseSet {
                FuseMap[3:0]{
                    ShiftedIO(QO[8]): 4'b0001;
                    ShiftedIO(QO[9]): 4'b0010;
                    ShiftedIO(QO[10]): 4'b0011;
                    ShiftedIO(QO[11]): 4'b0100;
                    ShiftedIO(QO[12]): 4'b0101;
                    ShiftedIO(QO[13]): 4'b0110;
                    ShiftedIO(QO[14]): 4'b0111;
                    ShiftedIO(QO[15]): 4'b1000;
                }
            }
            PinMap {
                SpareElement {
                    FuseMap[0]: B0_CRAR[0];
                    FuseMap[1]: B0_CRAR[1];
                    FuseMap[2]: B0_CRAR[2];
                    FuseMap[3]: B0_CRAR[3];
                }
            }
        }
        ColumnSegment (Bank1_Left){
            RowSegmentCountRange [1'b1:1'b1]; // Bank 1
            ShiftedIORange: QO[7:0]; // Left
            FuseSet {
                FuseMap[3:0]{
                    ShiftedIO(QO[0]): 4'b0001;
                    ShiftedIO(QO[1]): 4'b0010;
                    ShiftedIO(QO[2]): 4'b0011;
                    ShiftedIO(QO[3]): 4'b0100;
                    ShiftedIO(QO[4]): 4'b0101;
                    ShiftedIO(QO[5]): 4'b0110;
                    ShiftedIO(QO[6]): 4'b0111;
```

```
            ShiftedIO(QO[7]): 4'b1000;
        }
    }
    PinMap {
        SpareElement {
            FuseMap[0]: B1_CRAL[0];
            FuseMap[1]: B1_CRAL[1];
            FuseMap[2]: B1_CRAL[2];
            FuseMap[3]: B1_CRAL[3];
        }
    }
}
ColumnSegment (Bank1_Right){
    RowSegmentCountRange [1'b1:1'b1]; // Bank 1
    ShiftedIORange: QO[15:8]; // Right
    FuseSet {
        FuseMap[3:0]{
            ShiftedIO(QO[8]): 4'b0001;
            ShiftedIO(QO[9]): 4'b0010;
            ShiftedIO(QO[10]): 4'b0011;
            ShiftedIO(QO[11]): 4'b0100;
            ShiftedIO(QO[12]): 4'b0101;
            ShiftedIO(QO[13]): 4'b0110;
            ShiftedIO(QO[14]): 4'b0111;
            ShiftedIO(QO[15]): 4'b1000;
        }
    }
    PinMap {
        SpareElement {
            FuseMap[0]: B1_CRAR[0];
            FuseMap[1]: B1_CRAR[1];
            FuseMap[2]: B1_CRAR[2];
            FuseMap[3]: B1_CRAR[3];
        }
    }
}
}
```

The sample syntax of the memory library file in Figure G-2 specifies to Tessent MemoryBIST the following information for row and column segments:

- The sample memory has two banks (or row segments). Each bank has two spare rows and two spare columns available for repair. The two spare row elements per bank can repair a failure occurring only within that bank. The FuseSet sub-wrapper defines the address bits to be logged for each repairable element. Because each row segment has two spare columns, a total of four (4) ColumnSegment wrappers are needed:

  o Two ColumnSegment wrappers for *Bank0*

  o Two ColumnSegment wrappers for *Bank1*

  Each ColumnSegment has a RowSegmentCountRange property that assigns it to the appropriate range.

- Each of the two row and column segments is located in the address space defined by the address port Address[*10*]. The RowSegment(*Bank0*) and ColumnSegment(*Bank0_xxx*) are defined within the address space, whereby Address[*10*] is 1'b0. The RowSegment(*Bank1*) and ColumnSegment(*Bank1_xxx*) are defined within the address space, whereby Address[*10*] is 1'b1.

- This syntax specifies a segment range where the SegmentAddress bit 0 is defined as memory address port Address[*10*].

- Each row segment consists of two spare elements as defined by the NumberOfSpareElements property. Therefore, two fuse registers are required for RowSegment(*Bank0*), each of which logs the value of the ports Address[*9:7*] for the defective portion of memory within the row segment. Similarly for RowSegment(*Bank1*), two fuse registers log the value of the ports Address[*9:7*] when faults are detected within the portion of the memory covered by the range of this row segment.

- When implementing Column and IO repair, FuseMap registers capture the ShiftedIO fuse map values specified by the FuseSet/FuseMap/ShiftedIO properties for a given ColumnSegment. The value of the FuseMap register captured by the BIRA module identifies the memory IO on which a fault has been detected.

- When implementing Column repair, the FuseMap register described previously is needed to identify the faulty IO. However, extra information is required to identify the faulty column for a given IO. The PinMap/FuseMap property specifies the column address to log when a failure is detected.

- For each fuse and FuseMap register defined inside the FuseSet wrapper, a corresponding FuseMap property must be specified inside the PinMap wrapper. The PinMap wrapper maps the BISR spare element fuse registers to the corresponding spare element allocation ports on the memory.

Serial BISR Interface Example

The example in this section uses the same memory repair scheme as in the parallel BISR interface example, which has four redundant rows and four redundant columns. However, the following example uses a serial BISR interface.

Figure G-3 shows the serial BISR interface ports and the BISR register inside the memory module.

Figure G-3. Example Memory With Serial BISR Interface

The memory template associated with the memory illustrated in Figure G-3 is shown in
Figure G-4. The memory template for this example is similar to the parallel BISR interface
example but with two differences:

- The BisrParallelData port functions have been replaced by the serial BISR interface port
  functions (BisrClock, BisrSerialData, and BisrReset).

- The properties inside the PinMap/SpareElement wrapper now specify the order of the
  BISR register chain using the special RepairRegister[<x>] value instead of a port name.

Figure G-4. Memory Library File Sample Syntax for Serial BISR Interface

```
      // MemoryRepair ports for Bank B0
      Port(BISR_CLK) {
         Function: BisrClock;
         Direction: input;
      }
      Port(BISR_RST) {
         Function: BisrReset;
         Direction: input;
      }
      Port(BISR_SI) {
         Function: BisrSerialData;
         Direction: input;
      }
      Port(BISR_SO) {
         Function: BisrSerialData;
         Direction: output;
      }
```

```
RedundancyAnalysis {
    RowSegmentRange {
        SegmentAddress[0]: AddressPort(Address[10]);
    }
    RowSegment (Bank0){
        NumberOfSpareElements: 2;
        RowSegmentCountRange [1'b0:1'b0]; // Bank 0
        FuseSet {
            Fuse[3]: AddressPort(Address[9]);
            Fuse[2]: AddressPort(Address[8]);
            Fuse[1]: AddressPort(Address[7]);
            Fuse[0]: AddressPort(Address[0]);
        }
        PinMap {
            SpareElement {
                RepairEnable: RepairRegister[35];
                Fuse[0]: RepairRegister[31];
                Fuse[1]: RepairRegister[32];
                Fuse[2]: RepairRegister[33];
                Fuse[3]: RepairRegister[34];
            }
            SpareElement {
                RepairEnable: RepairRegister[30];
                Fuse[0]: RepairRegister[26];
                Fuse[1]: RepairRegister[27];
                Fuse[2]: RepairRegister[28];
                Fuse[3]: RepairRegister[29];
            }
        }
    }
    RowSegment (Bank1){
        NumberOfSpareElements: 2;
        RowSegmentCountRange [1'b1:1'b1]; // Bank 1
        FuseSet {
            Fuse[3]: AddressPort(Address[9]);
            Fuse[2]: AddressPort(Address[8]);
            Fuse[1]: AddressPort(Address[7]);
            Fuse[0]: AddressPort(Address[0]);
        }
        PinMap {
            SpareElement {
                RepairEnable: RepairRegister[17];
                Fuse[0]: RepairRegister[13];
                Fuse[1]: RepairRegister[14];
                Fuse[2]: RepairRegister[15];
                Fuse[3]: RepairRegister[16];
            }
            SpareElement {
                RepairEnable: RepairRegister[12];
                Fuse[0]: RepairRegister[8];
                Fuse[1]: RepairRegister[9];
                Fuse[2]: RepairRegister[10];
                Fuse[3]: RepairRegister[11];
            }
        }
    }
    ColumnSegment (Bank0_Left){
        RowSegmentCountRange [1'b0:1'b0]; // Bank0
```

```
                    ShiftedIORange: QO[7:0]; // Left
                    FuseSet {
                        FuseMap[3:0]{
                            ShiftedIO(QO[0]): 4'b0001;
                            ShiftedIO(QO[1]): 4'b0010;
                            ShiftedIO(QO[2]): 4'b0011;
                            ShiftedIO(QO[3]): 4'b0100;
                            ShiftedIO(QO[4]): 4'b0101;
                            ShiftedIO(QO[5]): 4'b0110;
                            ShiftedIO(QO[6]): 4'b0111;
                            ShiftedIO(QO[7]): 4'b1000;
                        }
                    }
                    PinMap {
                        SpareElement {
                            FuseMap[0]: RepairRegister[22];
                            FuseMap[1]: RepairRegister[23];
                            FuseMap[2]: RepairRegister[24];
                            FuseMap[3]: RepairRegister[25];
                        }
                    }
                }
                ColumnSegment (Bank0_Right){
                    RowSegmentCountRange [1'b0:1'b0]; // Bank 0
                    ShiftedIORange: QO[15:8]; // Right
                    FuseSet {
                        FuseMap[3:0]{
                            ShiftedIO(QO[8]): 4'b0001;
                            ShiftedIO(QO[9]): 4'b0010;
                            ShiftedIO(QO[10]): 4'b0011;
                            ShiftedIO(QO[11]): 4'b0100;
                            ShiftedIO(QO[12]): 4'b0101;
                            ShiftedIO(QO[13]): 4'b0110;
                            ShiftedIO(QO[14]): 4'b0111;
                            ShiftedIO(QO[15]): 4'b1000;
                        }
                    }
                    PinMap {
                        SpareElement {
                            FuseMap[0]: RepairRegister[18];
                            FuseMap[1]: RepairRegister[19];
                            FuseMap[2]: RepairRegister[20];
                            FuseMap[3]: RepairRegister[21];
                        }
                    }
                }
                ColumnSegment (Bank1_Left){
                    RowSegmentCountRange [1'b1:1'b1]; // Bank 1
                    ShiftedIORange: QO[7:0]; // Left
                    FuseSet {
                        FuseMap[3:0]{
                            ShiftedIO(QO[0]): 4'b0001;
                            ShiftedIO(QO[1]): 4'b0010;
                            ShiftedIO(QO[2]): 4'b0011;
                            ShiftedIO(QO[3]): 4'b0100;
                            ShiftedIO(QO[4]): 4'b0101;
                            ShiftedIO(QO[5]): 4'b0110;
                            ShiftedIO(QO[6]): 4'b0111;
```

```
                ShiftedIO(QO[7]): 4'b1000;
            }
        }
        PinMap {
            SpareElement {
                FuseMap[0]: RepairRegister[4];
                FuseMap[1]: RepairRegister[5];
                FuseMap[2]: RepairRegister[6];
                FuseMap[3]: RepairRegister[7];
            }
        }
    }
    ColumnSegment (Bank1_Right){
        RowSegmentCountRange [1'b1:1'b1]; // Bank 1
        ShiftedIORange: QO[15:8]; // Right
        FuseSet {
            FuseMap[3:0]{
                ShiftedIO(QO[8]): 4'b0001;
                ShiftedIO(QO[9]): 4'b0010;
                ShiftedIO(QO[10]): 4'b0011;
                ShiftedIO(QO[11]): 4'b0100;
                ShiftedIO(QO[12]): 4'b0101;
                ShiftedIO(QO[13]): 4'b0110;
                ShiftedIO(QO[14]): 4'b0111;
                ShiftedIO(QO[15]): 4'b1000;
            }
        }
        PinMap {
            SpareElement {
                FuseMap[0]: RepairRegister[0];
                FuseMap[1]: RepairRegister[1];
                FuseMap[2]: RepairRegister[2];
                FuseMap[3]: RepairRegister[3];
            }
        }
    }
}
}
```

The sample memory library file syntax shown in Figure G-4 shows how to use the
RepairRegister syntax. The RepairRegister index matches the order of the memory BISR chain
where RepairRegister[*0*] is closest to BISR_SO, and RepairRegister[*35*] is closest to BISR_SI.
Each repair register must be used and must correctly map to the corresponding Fuse, FuseMap,
or RepairEnable BIRA fuse information.

Appendix H
Functional Debug Memory Access

This appendix explains the functional debug memory access feature, which provides a mechanism to access memories in the context of functional system debug using existing memory BIST infrastructure.

The feature provides read and write access to any supported BISTable memory with negligible additional area. Refer to the "Requirements, Assumptions and Limitations" section for the currently supported memories.

Three basic memory access modes are:

- Read a single location of a single memory with a nondestructive scan-out that enables many algorithm executions with a single scan-in. The scan-out maintains the controller state in this access mode and therefore only requires a single initialization.This recommended usage is described in the first row of Table H-1 and within this appendix.

- Write single or multiple locations. The data pattern is selectable at run time. When writing multiple locations in a single execution, a custom algorithm controls the address sequence, and the data pattern can be modified as a function of the address. This mode can be used to initialize a block of locations with regular data patterns.

- Read a single location of many memories with destructive scan-out. Table H-1 describes this mode in rows two through four, where the scan-out destroys the controller state and you re-initialize the controller before beginning another algorithm execution.

_____ **Note** _____
The hardware disables the memory when the address from the controller goes out of range. The algorithm can cause an out-of-range condition because the controller step defines the address range used by the algorithm, not the selected memory. This is true even if you select a single memory to run the algorithm. Refer to the "Requirements, Assumptions and Limitations" section for details on when this may occur.

This appendix covers the following topics:

Feature Description

The Functional Debug Memory Access feature is an improvement over what currently is possible with soft programmable controllers, which is significantly more expensive than with the hard programmable controller.

The improvement consists of starting a hardcoded algorithm from an address loaded directly in the address register (register A, B, or both) via the setup chain. Normally, the algorithm's default values would override this value during the initialization phase of the controller. The initialization is suppressed via a control bit located on the setup chain or a controller input that is set at run time using the incremental_test_mode property in the DftSpecification/MemoryBist/Controller/AdvancedOptions wrapper.

_____ **Note** _____

When incremental_test_mode is On, algorithms that are used to read and write the memory must be hard-coded, or for soft programmable controller, downloaded at runtime..

Algorithm Description

Custom algorithms that access a single memory location and from a range of memory locations are described in this section.

The algorithm may be included during the Design Loading step so that it is hard-coded into the controller. For a soft programmable controller, the algorithm may also be downloaded during the Process Patterns Specification step.

The custom algorithm used to read a single location of memory is shown in Figure H-1. AddressRegisterA is initialized to value 0 and the expected data value is set to all zeros.When the default all zeros value of expected data is used, the content of the memory location is stored in the GO_ID registers for RAMs or in the MISR registers for ROMs. Compare failures are indicated on the GO signal, GO_ID registers and MISR registers when they are scanned out, unless the memory content matches the expected default value.

Figure H-1. Example Read Algorithm for a Single Location

```
MemoryOperationsSpecification {
  Algorithm (MemReadAlgo) {
   TestRegisterSetup {
      OperationSetSelect: SYNC;
      AddressGenerator {
         AddressRegisterA {
            load_row_address: min_row; //actual address loaded at run time
            load_column_address: min_column; //actual column address also
            load_bank_address: min_bank ; //actual bank address also
            // Recommended for GO_ID setup mode to increment address
            z_carry_in: x1_carry_out;
            x1_carry_in: y1_carry_out;
            y1_carry_in: none;
         }
      }
      DataGenerator {
         load_expect_data: all_zero; // data pattern can be changed at
                                     // run time
      }
   }
   MicroProgram {
      Instruction (MEM_READ) {
         OperationSelect: Read;
         AddressCommands {
            address_select: select_a;
            // Recommended for GO_ID setup mode to increment address
            x1_address: increment;
            y1_address: increment;
            z_address: increment;
         }
         DataCommands {
            expect_data: data_reg;
         }
         NextConditions {
            // Unconditional exit. Only one address read.
            // Address counter incremented by 1
         }
      }
   }
  }
}
```

To use Functional Debug Memory Access mode, the patterns should be constructed in the following sequence.

- Apply the initialization pattern once (Init).

- Apply the execution pattern in a loop (Exec).

- Iterate the Exec pattern until all locations of interest have been accessed.

The Init TestStep configures the controller to apply the selected algorithm to a single memory. Note that the Init TestStep does not run the algorithm and is a preparation step for the following Exec TestSteps. Therefore, the Init TestStep runtime is very short. Additionally, and important

for MissionMode usage where resources are more constrained, you only need the Init TestStep when a configuration change is necessary for the algorithm operating in the following Exec TestStep. For example, you can repeat the Exec TestStep without an intervening Init TestStep if an algorithm prepares for the next execution by incrementing the address. The Exec pattern is also shared by all memories on a controller step, which may cause out-of-range conditions on the single memory being tested. You need more Exec patterns in this case to fully cover the locations of interest. For further information, refer to the "Requirements, Assumptions and Limitations" section.

To create the Init TestStep, specify the following PatternsSpecification properties in the Patterns wrapper:

```
Patterns(func_debug_mode) {
  TestStep(init) {
    MemoryBist {
      run_mode: run_time_prog;
      Controller(instance) {
        AdvancedOptions {
          incremental_test_mode : on;
          freeze_step : <n>;
          enable_memory_list : <memory_id>;
          freeze_test_port : <n>;
          apply_algorithm : <algorithm_name>;
        }
      }
    }
  }
}
```

You may also override the initial settings of the address registers, data registers and CounterA that were defined in the algorithm. Specify new values for these parameters in the Controller/AlgorithmSetupOverrides wrapper.

The Exec TestStep runs the previously programmed algorithm without serially configuring the controller. Upon completion of the algorithm, the content of the memory location can be extracted by shifting out the GO_ID or MISR register. The hardware is built with a new GO_ID setup chain mode that enables scanning out the GO_ID or MISR register of the selected memory. This setup mode greatly reduces test time and preserves the controller state registers. Then, subsequent runs of the Exec TestStep can repeat the algorithm without reconfiguration.

To create the Exec TestStep, specify the following PatternsSpecification properties in the Patterns wrapper:

```
Patterns(func_debug_mode) {
  TestStep(exec) {
    MemoryBist {
      run_mode: hw_default;
      Controller(instance) {
        AdvancedOptions {
          incremental_test_mode : on;
        }
      }
    }
  }
}
```

Figure H-1 shows an example algorithm that can be used to read a single location and potentially be used for the GO_ID setup mode. Notice that this read algorithm increments the address by one to prepare the address registers for the next iteration of the Exec TestStep.

Compare failures during read access can be avoided if the expected data for this memory location is known and specified at run time. Then, scanning out the GO_ID register is unnecessary, which saves a significant amount of scan-in and scan-out time. However, large test data registers (ExpectDataRegister and WriteDataRegister) in the controller are needed to do so, and a corresponding number of connections are required between the controller and memories.

The write algorithm for a single location is essentially the same as the read algorithm. The read operation is replaced by a write operation, and the write data register is used instead of the expect data register. The standard Sync or SyncWR operation sets can be used to perform the operations or any other suitable operation set.

The full flexibility of the controller can be used to perform operations on multiple memory locations. Therefore, the expect and write data pattern can be modified based on the address, repeat loops can be used, address register segments can be linked to count fast column or fast row, and so on. The algorithms described previously can be generalized further to cover a range of locations instead of a single location. Figure H-2 shows an example.

Figure H-2. Example Write Algorithm for a Block of Locations

```
MemoryOperationsSpecification {
  Algorithm (MemWriteAlgo) {
   TestRegisterSetup {
       OperationSetSelect: SYNC;
       AddressGenerator {
          AddressRegisterA { // Address A is used in this example.
                           // Load default initial row and column addresses
             load_row_address: min_row ;
             load_column_address: min_column ;
             x1_carry_in: y1_carry_out;
             y1_carry_in: none;
          }
       }
       DataGenerator {
          load_write_data: all_zero; // initial data pattern
          // Data pattern can be manipulated as part of the algorithm
       }
       load_counter_a_end_count: 31; // 5 CounterA bits
       // Set the end count value to one less the largest block size that
       // can be accessed. Here, defaults to 32 locations.
    }


    MicroProgram {
       Instruction (MEM_BLOCK_WRITE) {
          OperationSelect: Write;
          branch_to_instruction: MEM_BLOCK_WRITE;
          AddressCommands {
             address_select: select_a;
             x1_address: increment;
             y1_address: increment;
          }
          CounterCommands {
             counter_a: increment;
          }
          DataCommands {
             write_data: data_reg;
          }
          NextConditions {
             counter_a_end_count: on;
          }
       }
     }
   }
  }
}
```

The data patterns are limited by the size of the test data registers. Most test algorithms only require 2-bit registers. However, this size would need to be increased to the maximum data path width if more flexibility is required, especially during write operations. Memory pipelining options would also multiply n times the number of registers, where n is the number of pipeline stages. Both the active area of the registers and routing to the memories must be considered in making the decision of increasing the data register size. Note that both test data registers (ExpectDataRegister and WriteDataRegister) have the same width in the controller.

An additional feature might be useful when performing read operations on multiple locations during a single test. You can compare a single data bit of each data word by specifying the following property:

```
Controller (instance) {
   MemoryInterface(memory_id) { // repeatable
      comparator_id_select: all | none | integer;
   }
}
```

This quickly verifies if a block of memory locations or all memory locations have the selected bit set to a specific value.

Verification of Read and Write Access Functions

The basic read and write access functions are verified by initializing a block of locations and reading it back. The freeze_step and enable_memory_list properties must be used to access only a single memory. For a list of memory ID names, refer to the DftSpecification of the TSDB.

As indicated previously, the memory access mode is applied using a series of TestSteps. Figure H-3 shows an outline of the PatternsSpecification used for this verification. Note that the outline does not show all the recommended configurations as outlined in the "Implementation" section. All locations are initialized to 0s (all_zero). Note that an Init TestStep is not needed prior to the mem_write_zeros_exec Exec TestStep because the default settings for the write algorithm are used. Next, two arbitrary locations are changed to 1s (all_one). Note that in the initialization TestSteps for writing these two arbitrary locations, the start count is specified to be the same as the Write algorithm's end count. This is done so that only a single write occurs, rather than the block write of 32 locations the algorithm would normally perform.

> **Note**
> For algorithms hard-coded into the memory BIST controller, only the initial value of CounterA can be specified in the PatternsSpecification at run time, and this counter only counts up. If an algorithm requires execution of X number of operations and CounterA is used to track the number of iterations, the value to be loaded into CounterA using the AlgorithmSetupOverrides/load_counter_a_start_count property is the algorithm's load_counter_a_end_count + 1 - X.

Lastly, the final content of all 32 locations is read back. The following example uses an 8-bit address register with column address aligned at bits [2:0].

Figure H-3. Test Steps Used for Verification of Debug Capability

```
Patterns(verify_func_debug_mode) {
  // Initialize the first 32 locations to all_zero
  TestStep(mem_write_zeros_exec) {
  }

  // Write all_one to first arbitrary location 8'h01
  TestStep(mem_write_init_1) {
    MemoryBist {
      Controller {
        AlgorithmSetupOverrides {
          load_counter_a_start_count : 31;
          DataGenerator {
            load_write_data : all_one;
          }
          AddressGenerator {
            AddressRegisterA {
              load_column_address : 3'b001;
            }
          }
        }
      }
    }
  }
}
```

```
TestStep(mem_write_exec_1) {
}
// Write all_one to second arbitrary location 8'h04
TestStep(mem_write_init_2) {
  MemoryBist {
    Controller {
      AlgorithmSetupOverrides {
        load_counter_a_start_count : 31;
        DataGenerator {
          load_write_data : all_one;
        }
        AddressGenerator {
          AddressRegisterA {
            load_column_address : 3'b100;
          }
        }
      }
    }
  }
}
TestStep(mem_write_exec_2) {
}

// Read 32 locations and compare content to all_zero.
// GO status for locations 8'h01 and 8'h04 will indicate a fail.
TestStep(mem_read_init) {
}
TestStep(mem_read_exec_1) {
}
TestStep(mem_read_exec_2) {
}
...
TestStep(mem_read_exec_32) {
}
}
```

Requirements, Assumptions and Limitations

The following are the requirements, assumptions and limitations in utilizing functional debug memory access in Tessent Shell:

- This feature is supported by hard and soft programmable Memory BIST controllers.

- You must hard code a custom algorithm that can be used to read from a single location. For a soft programmable controller, you may download the algorithm at run time. For an example of the read algorithm, refer to Figure H-1.

- This feature is incompatible with Parallel Static Retention Testing and memory reset modes that only operate with library algorithms.

- This feature supports all memory types. Although DRAMs are supported, some restrictions concerning their refresh operations may apply.

- All memory operations (read or write) are initiated through the IJTAG protocol.

- The completion time for each memory access requiring controller setup, partial or complete, depends on a number of factors. The length of the setup chain is variable based on the controller options selected. For example, using local comparators causes the setup chain to be significantly longer than when using shared comparators. The execution time of the custom algorithm performing the memory access is also affected by the options selection. For example, the depth of the pipeline to and from the memory delays the completion of the custom algorithm. For hard programmable controllers, several NoOperation operations are performed after completion of a custom algorithm.

- The memory address must be decomposed in terms of its physical bank, row, and column address before loading into the controller. If present, address mapping is applied automatically. You can find this information about the memory structure in the Core Memory description.

- The switch from functional to BIST/debug mode and vice-versa must be done in such a way that the memory content is not corrupted, which you can achieve by setting the memory select and write enable pins to an inactive state when switching occurs. The switch can occur while clocks are free-running or when clocks are gated. However, the Memory BIST controller and memories still must receive the active clock signal. Tessent Shell MemoryBIST does not check for this condition.

- The writing of arbitrary data patterns to any memory tested by a controller requires setting the data_register_bits property in the DftSpecification/MemoryBist/Controller/ AlgorithmResourceOptions wrapper to the maximum memory data bus width of all memories. Please review the usage conditions of this controller property because in some cases, this can have an important impact on area.

- In a hard programmable controller, only the initial value of CounterA can be specified in the PatternsSpecification at run time, and this counter only counts up. If an algorithm requires execution of X number of operations, and CounterA is used to track the number of iterations, the value to be loaded in CounterA using AlgorithmSetupOverrides/ load_counter_a_start_count must be the algorithm's load_counter_a_end_count + 1 - X. Another option is to use a segment of the second address counter. The address counter can be loaded with X - 1 and count down until reaching 0.

- If the address is higher than the tested memory's maximum address, either incremented by the algorithm or scanned in with the pattern file, the GO_ID registers remain at 0. The hardware disables the memory when the address from the controller goes out of range. The algorithm can cause an out-of-range condition because the controller step defines the address range used by the algorithm, not the selected memory. This is true even if you select a single memory to run the algorithm. There are three situations where this might happen:

    a. **Multiple memories of different sizes in a step:** The tool defines the maximum address as the largest bank address, row address, and column address found in any of the memories. For example, if memory M1 has two column address bit and M2 has four columns, M1 is de-selected half of the time as the algorithm always counts columns from 0 to 3.

b. **Multi-port memories with an odd number of rows:** The tool rounds up the number of rows for the step to the next even value when the memory allows concurrent/shadow read operations.

c. **Multi-port and Single-port memories with an odd number of rows:** The tool rounds up the number of rows for the step to the next even value when a memory allows operations with strobes in consecutive cycles.

Implementation

This section describes how to implement and verify the Functional Debug Memory access hardware. The process is the same as usual except for the steps described in the following sections.

DftSpecification

To enable the hardware and pattern capabilities, the incremental_test_mode property must be specified for the MemoryBIST controller when configuring the DFT Specification:

```
DftSpecification {
  MemoryBist {
    Controller(id) {
      AdvancedOptions {
          .
          .
        incremental_test_mode: on ;
          .
          .
      }
    }
  }
}
```

When incremental_test_mode is set to on, Tessent MemoryBIST generates the necessary setup chain muxing and controller ports to enable functional system debug. You may also automatically populate the default DFT specification by issuing the following command prior to generating the DFT Specification using the create_dft_specification command:

```
set_defaults_value DftSpecification/MemoryBist/ControllerOptions/
incremental_test_mode on
```

PatternsSpecification

The recommended settings and usage for Functional Debug Memory Access mode requires preparing and applying an initialization (Init) TestStep and a series of execution (Exec) TestSteps.

The Init TestStep configures the controller to apply the selected algorithm to a single memory. The custom algorithm default address, expect data and write data can be overridden. The Exec

TestStep runs the previously programmed algorithm. Upon completion of the algorithm, the content of the memory location can be extracted by shifting out the GO_ID or MISR register. The Exec pattern is repeated until all locations of interest have been accessed.

The recommended usage configurations, as well as additional usage configurations for Functional Debug Mode, are outlined in Table H-1 and the Table Notes that follow. All configurations have the following TestStep settings in common:

- preserve_bist_inputs : on ; // Necessary to avoid corruption of the memory between TestSteps

- incremental_test_mode : on ; // Enables using the AlgorithmSetupOverrides wrapper to specify the address

- compare_go_id : on ; // Necessary to annotate the pattern with the name of the GO_ID_REG registers

- freeze_test_port : *int* ; // Not required, but it can save test time since it is not necessary to read the same address location from all test ports

If your goal is to read the content of multiple memories within the same TestStep, one of the following conditions should exist:

- Memories have local comparators, OR

- The MemoryBIST controller has a single Step

Table H-1. Functional Debug Mode Usage Configurations

comparator_location	freeze_step	enable_memory_list	apply_algorithm[1]	PatternsSpec[2]	Chain	Comments
X	Yes	Yes[4]	Read + Incr Addr	Init + (N*Exec)	GOID of selected memory only	Recommended settings and documented usage in this Appendix
X	Yes	No	Read	N* (Init+Exec)	Entire short setup chain	Pattern patching required[3]
shared_in_controller	No	X	Read	N* (Init+Exec)	Entire short setup chain	Pattern patching required[3] and the results supply only the contents of the memory in the last Step

Table H-1. Functional Debug Mode Usage Configurations (cont.)

comparator_location	freeze_step	enable_memory_list	apply_algorithm[1]	PatternsSpec[2]	Chain	Comments
per_interface	No	X	Read	N* (Init+Exec)	Entire short setup chain	Pattern patching required[3]

Table Notes:

1. The applied algorithm operations are described below:

   o **Read + Incr Addr** — The algorithm reads one address location and increments the address. This algorithm can only be used if freeze_step is active (set to an integer value) and a single memory is enabled.

   o **Read** — The algorithm only reads one address location. This is necessary to read the same location in all memories of all controller steps.

2. The PatternsSpecification TestStep ordering definitions are as follows:

   o **Init** — A TestStep where you specify the run_mode as run_time_prog and the controller is configured to apply the selected algorithm. The custom algorithm default address, expect data and write data can be overridden as shown in the first TestStep of Figure H-5.

   o **Exec** — A TestStep where the controller runs in the hw_default run_mode.

   o **N** — The number of address locations to read.

3. Pattern patching indicates that the Init TestStep must be modified for each address location that is to be read.

4. Enable a single memory only.

For the recommended configuration usage, the example shown in Figure H-4 shows how to set up a functional debug write access utilizing the custom algorithm shown in Figure H-2 that would already be processed during the Design Loading step and hard-coded into the controller.

In the Init TestStep, the MemWriteAlgo algorithm is selected and is applied to memory m1 in controller step 1. By default, the custom algorithm writes zeros to the first 32 locations. The AlgorithmSetupOverrides wrapper modifies the starting column, row and bank addresses as well as the write data pattern. The data written to the memory is inverted when the column or row address is odd. In the Exec TestStep, the modified custom algorithm is applied to 32 memory locations.

Figure H-4. Functional Debug Mode Write Example

```
PatternsSpecification(core,rtl,signoff) {

  Patterns(Functional_debug_mode_write) { // {{{
    ClockPeriods {
      clk : 12.0ns;
    }
    TestStep(Functional_debug2_init) {
      MemoryBist {
        run_mode : run_time_prog;
        AdvancedOptions {
```

```
                    preserve_bist_inputs : on;
                }
                Controller(c1_inst) {
                  MemoryInterface(m1) {
                    comparator_id_select : all;
                  }
                  AdvancedOptions {
                    enable_memory_list : m1;
                    incremental_test_mode : on;
                    test_execution_cycles : 10;
                    apply_algorithm : MemWriteAlgo;
                    freeze_step : 1;
                  }
                  AlgorithmSetupOverrides {
                    AddressGenerator {
                      AddressRegisterA {
                        load_bank_address : 1'b1;
                        load_column_address : 2'b00;
                        load_row_address : 4'b1111;
                      }
                    }
                    DataGenerator {
                      load_write_data : 16'b0000000000000011;
                      invert_data_with_row_bit : r[0];
                      invert_data_with_column_bit : c[0];
                    }
                  }
                }
              }
            }
            TestStep(Functional_debug2_exec) {
              MemoryBist {
                run_mode : hw_default;
                AdvancedOptions {
                  preserve_bist_inputs : on;
                }
                Controller(c1_inst) {
                  AdvancedOptions {
                    incremental_test_mode : on;
                    test_execution_cycles : 1000;
                  }
                }
              }
            }
          }
        } // }}}
      }
```

The example shown in Figure H-5 shows how to set up a functional debug read access, utilizing the custom algorithm shown in Figure H-1, that would already be processed during the Design Loading step and hard-coded into the controller.

In the Init TestStep, the MemReadAlgo algorithm is selected and is applied to memory m1 in controller step 1. By default, the custom algorithm reads one location, compares the content to zeros then increments the address. The AlgorithmSetupOverrides wrapper modifies the starting column row and bank addresses to read from. The three subsequent Exec TestSteps run the

modified custom algorithm and extracts the memory content starting at column address 0, row address 5 and bank address 0.

Figure H-5. Functional Debug Mode Read Example

```
Patterns(Functional_debug_mode_read) { // {{{
  ClockPeriods {
    clk : 12.0ns;
  }
  TestStep(Functional_debug3_init) {
    MemoryBist {
      run_mode : run_time_prog;
      AdvancedOptions {
        preserve_bist_inputs : on;
      }
      Controller(c1_inst) {
        MemoryInterface(m1) {
          comparator_id_select : all;
        }
        AdvancedOptions {
          enable_memory_list : m1;
          incremental_test_mode : on;
          test_execution_cycles : 10;
          apply_algorithm : MemReadAlgo;
          freeze_step : 1;
        }
        AlgorithmSetupOverrides {
          AddressGenerator {
            AddressRegisterA {
              load_bank_address : 1'b0;
              load_column_address : 2'b00;
              load_row_address : 4'b0101;
            }
          }
          DataGenerator {
          }
        }
      }
    }
  }
}
```

```
      TestStep(Functional_debug3_exec1) {
      MemoryBist {
        run_mode : hw_default;
        AdvancedOptions {
          preserve_bist_inputs : on;
        }
        Controller(c1_inst) {
          AdvancedOptions {
            incremental_test_mode : on;
            test_execution_cycles : 500;
          }
          DiagnosisOptions {
            compare_go_id : on;
          }
        }
      }
    }
    TestStep(Functional_debug3_exec2) {
      MemoryBist {
        run_mode : hw_default;
        AdvancedOptions {
          preserve_bist_inputs : on;
        }
        Controller(c1_inst) {
          AdvancedOptions {
            incremental_test_mode : on;
            test_execution_cycles : 500;
          }
          DiagnosisOptions {
            compare_go_id : on;
          }
        }
      }
    }
    TestStep(Functional_debug3_exec3) {
      MemoryBist {
        run_mode : hw_default;
        AdvancedOptions {
          preserve_bist_inputs : on;
        }
        Controller(c1_inst) {
          AdvancedOptions {
            incremental_test_mode : on;
            test_execution_cycles : 500;
          }
          DiagnosisOptions {
            compare_go_id : on;
          }
        }
      }
    }
} // }}}
```

Appendix I
Advanced BAP Memory Access

The advanced BAP enables certain feature overrides in the hw_default operating mode of memory BIST controllers attached to the BAP. Test time can be reduced significantly by eliminating shift cycles to serially configure the controllers, at the cost of additional connections between the BAP and the controllers.

The portion of the advanced BAP memory access feature that interacts with the memory BIST controllers is configurable through the IJTAG protocol. This controller configuration may be used for manufacturing test within the ATE environment, as well as for in-system test through a Tessent MissionMode controller. Using a MissionMode controller for in-system test is a general solution enabling access to all instruments connected to the IJTAG network, including memory BIST controllers. All options of the PatternsSpecification are supported when generating patterns for the MissionMode controller.

The advanced BAP memory access feature also provides the ability of invoking memory tests through system signals connected to the BAP direct access interface, rather than serially shifting test configuration data and results by the IJTAG network. The direct access interface supports a low-latency protocol to configure the MemoryBIST controller, perform Go/NoGo tests, and monitor the pass/fail status. However, some modes of operation such as detailed diagnosis are not available.

This appendix describes the advanced BAP memory access method in the topics listed below. These topics provide information on the direct access interface architecture, interaction with the system control logic, procedures for inserting the direct access interface and generating simulation workbench for verification.

BAP Direct Access Interface Feature Description

The BAP module is created for each MBIST DftSpecification insertion pass and services one or more MemoryBIST controllers. The BAP direct access interface feature creates a set of ports that can be connected to system logic.

The direct access interface supports a low latency protocol to configure the MemoryBIST controller, perform Go/NoGo tests and monitor the pass/fail status. The controllers attached to the BAP may operate on the same or different clock domains depending on the functional clock sources of the memories under test. There are two different implementations of the feature, offering a trade-off between gate count and latency when multiple functional clocks are involved. Timing considerations are also slightly different.

The sections listed below provide the system-level details of the BAP direct access feature.

BAP Architecture

A block diagram for the advanced BAP, with both BAP and controller direct access is shown in the figure below.

Figure I-1. Advanced BAP Memory Access Diagram (single sequencer)

The advanced BAP memory access feature is comprised of two components. The BAP direct access interface ports are shown on the left side of the BAP below the IJTAG network interface ports. These connections are user-configurable and connect to the functional logic. On the right side, connections from the BAP to the MemoryBIST controllers are completed automatically and are not user configurable. The signals to the MemoryBIST controllers are programmed by the IJTAG TDRs added into the BAP or, if present, configured through ports of the BAP direct access interface. The ports on the MemoryBIST controllers can be implemented independently of the BAP direct access interface. The advanced BAP enables certain feature overrides, such as algorithm, for all memories without serially configuring the controllers. By operating the memory BIST controllers in hw_default mode, test time can be reduced by eliminating shift cycles to setup the memory test in the IJTAG protocol.

The direct access interface provides the ability of invoking memory tests through system signals connected to the BAP. The sequencer within the BAP implements the low latency protocol to start and stop the memory test. In this implementation, a single sequencer is used to minimize the gate count of the BAP. The clock of the sequencer, sys_clock, should have a relatively low frequency, comparable to ijtag_tck, to make sure that the control signals generated arrive in the correct order at all MBIST controllers.

The direct access interface offers basic options to configure the memory test. You can change the test algorithm and operation set to be applied on the memories. You can choose which controllers, step and memory to run based on your in-system test requirement. For example, if the allocated test time is limited, you can take advantage of this flexibility through the selection of a short algorithm with few instructions, or to only perform memory BIST on one memory.

Repair analysis can also be performed through the direct access interface. The results of this analysis can be used to perform either soft or hard incremental repair by accessing the BISR controller via the Tessent Mission Mode controller or the direct access interface of the BISR controller.

The output ports of the direct access interface indicate the global pass/fail and completion status of all active controllers. The execution time of each controller is typically different, so the pass/fail and completion status of individual controllers can also be monitored.

Figure I-2 shows a different implementation of the BAP shown in Figure I-1, with a sequencer for each functional clock. This implementation enables to further reduce the time required to start and stop the controllers, and more accurately constrains the critical paths from the sequencers to the MBIST controllers sharing the same functional clock. The cost consists of dedicated input/output control signals, as well as a sequencer for each functional clock.

Figure I-2. Advanced BAP Memory Access Diagram (multiple sequencers)

BAP Direct Access Interface Pins

The BAP direct access interface pins that are created depend upon the features configured in the DftSpecification.

Specifying the DftSpecification direct_access property to "on" enables creating the BAP direct access interface. By default, the BAP direct access interface contains the ports necessary to support the controller features enabled within the Controller/DirectAccessOptions/ ExecutionSelections wrapper. The ports related to controller selection are not present if there is only a single memory BIST controller. Refer to the Inserting the BAP Direct Access Interface section for more information on implementation of the BAP direct access interface.

Table I-1 describes the direct access interface ports that the tool creates on the BAP.

_____ **Note** _____
All MBIST controller port names described in Table I-1 that automatically connect to the BAP direct access interface are prefixed by "MBISTPG_". The exceptions are the controller ports connected to the BIST_SETUP, CHECK_REPAIR_NEEDED, and PRESERVE_FUSE_REGISTER signals.

Table I-1. BAP Direct Access Interface Ports

Port Name	Type	Description
User Connected BAP Signals		
sys_reset[_<clock_domain>]	Input	An active low signal that asynchronously resets the registers inside the advanced BAP for the memory access mode.
sys_clock[_<clock_domain>]	Input	System clock.
sys_test_init[_<clock_domain>]	Input	An active high signal releasing the asynchronous reset and enabling the clock of all memory BIST controllers. This input is used when multiple memory BIST controller runs are performed and results need to be preserved from one run to the other. Set this input to 1, together with sys_test_start, for the first controller run and maintain this value for all subsequent runs. Set this input to 0 at the end of all controller runs to disable the clock and assert the asynchronous reset of all memory BIST controllers. This input can be tied to 0 if it is not necessary to maintain the state of memory BIST controllers between controller runs.

Table I-1. BAP Direct Access Interface Ports (cont.)

Port Name	Type	Description
sys_test_start[_<clock_domain>]	Input	There is only one sys_test_start input when direct_access_clock_source is set to common. There is one such input per BIST clock domain when direct_access_clock_source is set to per_bist_clock_domain. When the corresponding sys_test_init signal is 0, setting sys_test_start to 1 performs the following operations: • 1. Release the asynchronous reset of all memory BIST controllers. • 2. Enable the clock of all memory BIST controllers. • 3. Enable all controllers. • 4. Start all controllers. After the memory test, setting sys_test_start to 0 performs the operations in the reverse order: • 4. Stop all controllers. • 3. Disable all controllers. • 2. Disable the clock of all memory BIST controllers. • 1. Assert the asynchronous reset of all memory BIST controllers. When the corresponding sys_test_init signal is 1, only operations 3 and 4 are performed.
sys_test_pass[_<clock_domain>]	Output	Memory test pass/fail status. Asserted to 0 when at least one controller detects a failure.
sys_test_done[_<clock_domain>]	Output	Memory test completion status. Asserted to 1 when all controllers have completed their execution.
sys_ctrl_pass[*msb*:*lsb*]	Output	Pass/fail status per controller. One-hot encoding indicates which controller(s) detected a failure.
sys_ctrl_done[*msb*:*lsb*]	Output	Completion status per controller. One-hot encoding indicates which controller(s) completed their execution.
sys_ctrl_select[*msb*:*lsb*]	Input	Individual enable signal per controller. Select the controller(s) to run by setting the corresponding bit to 1.

Table I-1. BAP Direct Access Interface Ports (cont.)

Port Name	Type	Description
sys_algo_select[*msb:lsb*]	Input	Test algorithm selection code.
sys_select_common_algo	Input	Enable the algorithm selection. When set to 1, the algorithm associated with the code specified on the sys_algo_select port is used. Otherwise, the default algorithm encoded at RTL generation is run.
sys_opset_select[*msb:lsb*]	Input	Operation set selection code.
sys_select_common_opset	Input	Enable the operation set selection. When set to 1, the operation set associated with the code specified on the sys_opset_select port is used. Otherwise, the default operation set encoded at RTL generation is run.
sys_step_select[*msb:lsb*]	Input	Controller step selection code.
sys_step_select_en	Input	Enable the controller step selection. When set to 1, the step associated with the code specified on the sys_step_select port is used. Otherwise, all the steps are run.
sys_memory_select[*msb:lsb*]	Input	Individual enable signal per memory within a controller step. Select the memories to run by setting the corresponding bit to 1.
sys_memory_select_en	Input	Enable the memory selection. When set to 1, the memories associated with the signals specified on the sys_memory_select port are tested. Otherwise, all the memories are tested.
sys_test_port_select[*msb:lsb*]	Input	Memory test port selection code.
sys_test_port_select_en	Input	Enable controller test port selection. When set to 1, the test port associated with the signals specified on the sys_test_port_select port are used. Otherwise, all the test ports are tested.
sys_data_inv_col_add_bit_select[*msb:lsb*]	Input	Select the column address bit that inverts the applied write and expect data registers. The select width is set by the AlgorithmResourceOptions/max_data_inversion_address_bit_index property, which defaults to 0 and corresponds to c[0].

Table I-1. BAP Direct Access Interface Ports (cont.)

Port Name	Type	Description
sys_data_inv_col_add_bit_select_en	Input	Enable data inversion based on column address bit selection. When set to 1, data inversion is determined by the signal present on sys_data_inv_col_add_bit_select.
sys_data_inv_row_add_bit_select[*msb:lsb*]	Input	Select the row address bit that inverts the applied write and expect data registers. The select width is set by the AlgorithmResourceOptions/ max_data_inversion_address_bit_index property, which defaults to 0 and corresponds to r[0].
sys_data_inv_row_add_bit_select_en	Input	Enable data inversion based on row address bit selection. When set to 1, data inversion is determined by the signal present on sys_data_inv_row_add_bit_select.
sys_mbistcfg_en	Input	Override the configuration data for the memory clusters. When set to 1, the value specified on the sys_mbistcfg_interface<id> port is applied to the associated memory cluster interface. Otherwise, the default value encoded at RTL generation is used.
sys_mbistcfg_interface*<id>*	Input	Configuration data for memory cluster interface <id>.
sys_*<ctrl_id>*_*<cluster_id>*_ *<cluster_reset_port_name><x>*	Input	Reset signal for each memory cluster instance. The memory cluster TCD defines the port name (reset) and its active polarity. The port specifies the InterfaceReset function of the cluster module. *ctrl_id* and *cluster_id* are respective DftSpecification wrapper names.
sys_*<ctrl_id>*_*<cluster_id>*_ *<cluster_request_port_name><x>*	Input	Request signal for each memory cluster instance. The memory cluster TCD defines the port name (request) and its active polarity. The port specifies the BistOn function of the cluster module. If the request signal is multi-bit, the BAP input is a bus port. *ctrl_id* and *cluster_id* are respective DftSpecification wrapper names.
sys_retention_test_phase[1:0]	Input	Retention test phase selection code for SMarchCHKB based library algorithms.

Table I-1. BAP Direct Access Interface Ports (cont.)

Port Name	Type	Description
sys_preserve_test_inputs	Input	An active high signal that maintains the memories under BIST control. For retention testing, use this signal to avoid corrupting the memory content during the retention pause.
sys_bira_en	Input	An active high signal that enables repair analysis for the repairable memories under BIST control.
sys_check_repair_needed	Input	An active high signal that enables examination of the repair analysis status register after BIST completion.
sys_preserve_fuse_register	Input	An active high signal that enables the preservation of repair analysis register values from a previous test step. This inhibits the loading of the BISR register values into the repair analysis registers when a new test step begins.
Controller Auto-Connected Signals		
BIST_ASYNC_RESET	Output	An active low asynchronous reset signal that connects to the ASYNC_RESETN input of the MBIST controller.
bistEn[*msb:lsb*]	Output	An active high enable signal per controller used in the start/stop protocol. Connects to the EN input of the MBIST controller.
BIST_SETUP	Output	Run mode signal used in the start/stop protocol. Connects to the BIST_SETUP input of the MBIST controller.
MBISTPG_GO[*msb:lsb*]	Input	Pass/fail status per controller. Connects to the GO output of the MBIST controller.
MBISTPG_DONE[*msb:lsb*]	Input	Completion status per controller. Connects to the DONE output of the MBIST controller.
BIST_ALGO_SEL[x:y]	Output	Test algorithm selection code. Connects to the ALGO_SEL input of the MBIST controller.
BIST_SELECT_COMMON_ALGO	Output	An active high signal that enables the test algorithm selection code. Connects to the SELECT_COMMON_ALGO input of the MBIST controller.
BIST_OPSET_SEL[x:y]	Output	Operation set selection code. Connects to the OPSET_SEL input of the MBIST controller.

Table I-1. BAP Direct Access Interface Ports (cont.)

Port Name	Type	Description
BIST_SELECT_COMMON_OPSET	Output	An active high signal that enables the operation set selection code. Connects to the SELECT_COMMON_OPSET input of the MBIST controller.
BIST_STEP_SEL[x:y]	Output	Controller step selection code. Connects to the STEP_SELECT input of the MBIST controller.
BIST_STEP_SELECT_EN	Output	An active high signal that enables the controller step selection code. Connects to the STEP_SELECT_EN input of the MBIST controller.
BIST_MEM_SEL[x:y]	Output	Individual selection signal per memory within a controller step. Connects to the MEM_SELECT input of the MBIST controller.
BIST_MEM_SELECT_EN	Output	An active high signal that enables the memory selection. Connects to the MEM_SELECT_EN input of the MBIST controller.
BIST_ALGO_MODE0 BIST_ALGO_MODE1	Output	Retention test phase selection code for library algorithms. Connects to the ALGO_MODE[1:0] inputs of the MBIST controller.
BIST_TEST_PORT_SEL[x:y]	Output	Controller test port selection code. Connects to the TEST_PORT_SELECT input of the MBIST controller.
BIST_TEST_PORT_SELECT_EN	Output	An active high signal that enables the controller test port selection code. Connects to the TEST_PORT_SELECT_EN input of the MBIST controller.
BIST_SELECT_TEST_DATA	Output	An active high signal that maintains memories under BIST control. Connects to the TESTDATA_SELECT input of the MBIST controller.
BIST_DATA_INV_COL_ADD_BIT_SEL[x:y]	Output	Controller column address data inversion selection. Connects to the DATA_INV_COL_ADD_BIT_SELECT input of the MBIST controller.

Table I-1. BAP Direct Access Interface Ports (cont.)

Port Name	Type	Description
BIST_DATA_INV_COL_ADD_BIT_SELECT_EN	Output	An active high signal that enables the column address data inversion selection. Connects to the DATA_INV_COL_ADD_BIT_SELECT_EN input of the MBIST controller.
BIST_DATA_INV_ROW_ADD_BIT_SEL[x:y]	Output	Controller row address data inversion selection. Connects to the DATA_INV_ROW_ADD_BIT_SELECT input of the MBIST controller.
BIST_DATA_INV_ROW_ADD_BIT_SELECT_EN	Output	An active high signal that enables the row address data inversion selection. Connects to the DATA_INV_ROW_ADD_BIT_SELECT_EN input of the MBIST controller.
BIRA_EN	Output	An active high signal that enables controller repair analysis. Connects to the BIRA_EN input of the MBIST controller.
CHECK_REPAIR_NEEDED	Output	An active high signal that enables repair analysis status register examination after BIST completion. Connects to the CHECK_REPAIR_NEEDED input of the MBIST controller.
PRESERVE_FUSE_REGISTER	Output	An active high signal that enables preserving repair analysis register values from the previous test step. Inhibits the loading of the BISR register values into the repair analysis registers when a new test step begins. Connects to the MBIST_RA_PRSRV_FUSE_VA input of the MBIST controller.
sel	Input	Reserved for future use. Tied to 0.

BAP Shared Bus Direct Access Interface Pins

An example advanced BAP configuration for Shared Bus access is provided below to highlight the direct access interface ports that are created and how they connect to the cluster instances.

The memory cluster is described by the following DftSpecification, resulting in the connections shown in Figure I-3:

```
DftSpecification {
  MemoryBist {
    Controller(c1) {
      MemoryCluster(cluster1) {
        instance_name: c1_inst1;
      }
    }
    ReusedController(c2) {
      reused_controller_id: c1;
      MemoryCluster(cluster1) {
        // same cluster module as c1
        instance_name: c1_inst2;
      }
    }
    Controller(c3) {
      MemoryCluster(cluster1) {
        instance_name: c1_inst3;
      }
    }
  }
}
```

Figure I-3. Advanced BAP Shared Bus Access Diagram

Note that the direct access interface cluster configuration ports are shared between all the cluster types (modules) and their instances, while each memory cluster has dedicated direct access input ports for reset and request.

_____ **Note** _____
The timing protocol for the memory cluster reset and request inputs are different than the other direct access interface input signals. The protocol is determined by the core provider and the user is responsible for its proper implementation.

_____ **Note** _____
The direct access interface system connections for the cluster reset and request inputs, as well as the BAP configuration_data inputs, are not automated at this time with the BistAccessPort/Connections wrapper in the DftSpecification.

System Logic Interaction and Timing

The following sections provide details for system logic connections and the direct access interface timing:

Timing Diagram

The general access protocol for in-system testing using advanced BAP memory access with single and multiple sequencers is described in this section.

Figure I-4 shows the protocol for the BAP implementation case described in Figure I-1, where a low frequency functional clock is used for the sequencer generating control signals for all MBIST controllers attached to the BAP. The timing diagram illustrates memory tests being initiated on memory controllers 0 and 2, and each runs algorithm 4.

Determining the codes used for controller and algorithm selection shown in this diagram, as well as selection codes for other controller features, is described in the Selection Codes section.

Figure I-4. BAP Direct Access Interface Timing Protocol

The sys_reset input is the asynchronous reset for the sequencer, and all BAP outputs are forced to their inactive value when this input is 0. Once the sequencer reset is released (that is, a 0 to 1

transition), the sys_test_start input is used to initiate a test. This input goes through a synchronizer before being applied to the sequencer in the event that the system inputs (sys_*) are not generated by sys_clock.

_____ **Note** _____

Starting with the 2020.3 release, a synchronizer cell from the cell library is implemented inside of the BAP, rather than a double flop implementation. Depending on what type of synchronizer cell the designer selects from the cell library, the timing of some BAP direct access interface signals may need to be adjusted. Synchronizer cells with a reset input do not need adjustment. For synchronizer cells without a reset, the sys_test_init and sys_test_start BAP direct access inputs must be set active for at least two functional clock cycles, before the sys_reset signal is set inactive. This ensures that the synchronizer cell is properly initialized, or flushed, before initiating a test.

The sequence of events shown in Figure I-4 during test initiation consists of the following:

1. Release reset of all MBIST controllers — 0 to 1 transition on BIST_ASYNC_RST

2. Prepare run mode for MBIST controllers — 00 to 10 transition on BIST_SETUP

3. Start selected MBIST controllers — 000 to 101 transition on BISTen

During execution, the sys_test_pass output remains high until the test is done (0 to 1 transition on sys_test_done) unless a fault is detected by one of the selected MBIST controllers as indicated by the red dotted line.

Once the test is done, the sys_test_start input can be set to 0. The sequence of events described earlier is then performed in the reverse order. A minimum of 6 clock cycles should be allowed before re-asserting sys_test_start to initiate a new test with the same or different inputs.

Figure I-5 shows a timing diagram for the BAP implementation case described in Figure I-2 with multiple sequencers. The same algorithm and controllers are selected as indicated by the sys_select_common_algo, sys_algo_select and syc_ctrl_select inputs. These shared inputs are applied ahead of time to the BAP to leave enough time for them to propagate to the MBIST controllers. Controllers are started per clock domain using the dedicated sys_test_start_CKA and sys_test_start_CKB BAP inputs, which are assumed to be synchronized to clock sys_clock_CKA and sys_clock_CKB respectively.

Figure I-5. BAP Direct Access Interface Timing Protocol with Multiple Sequencers

Clocking Schemes

The BAP direct access interface supports two different clocking schemes. Choosing between the two schemes depends on how the in-system test is operated.

If the system control logic operates on a slow functional clock, the BAP direct access interface can use the same clock source to manage the in-system test. One sequencer is implemented in the BAP to operate all controllers. The absolute value of the delays from the BAP outputs to the MBIST controllers are not important. However, the signals should arrive in the order described in the "Timing Diagram" section. This means that the difference in propagation delays cannot exceed one period of sys_clock. Selecting a relatively slow speed clock, similar to TCK (typically at tens of MHz), makes it relatively easy to meet this requirement without having to analyze the detailed timing.

If the system control logic operates on fast functional clocks, the BAP direct access interface can be built to manage the in-system test on a per-clock domain basis. Multiple sequencers are implemented in the BAP and each sequencer operates the controllers belonging to the same clock domain. This implementation further reduces the time required to start and stop the controllers, and more accurately constrains the critical paths from the sequencers to the MBIST controllers sharing the same functional clock. In addition to the sequencer clock input, other signals such as reset, memory test pass/fail, and completion status, are dedicated per clock domain as shown in Table I-1.

Sampling Pass/Done Signals

Sampling Pass/Done BAP outputs is done slightly differently for the single and multiple sequencer implementations.

In the single sequencer case, shown in Figure I-1, the global sys_test_pass and sys_test_done status signals, as well as the per-controller sys_ctrl_pass and sys_ctrl_done status signals are not registered outputs of the BAP direct access interface. These signals are sourced from memory

BIST controller DONE and GO outputs, and can be from various memory BIST clock domains. Because of this variation, consideration on how the functional control logic captures these signals is necessary to avoid potential glitches in sampling.

There are three methods available for avoiding potential glitches when functional logic captures pass/done status from the BAP direct access interface, depending on the exact requirements:

1. Wait a fixed time duration until all controllers complete test execution. This is the method employed during manufacturing test. Test time needs to be known for each run, and can be determined from the verification testbench as described in the "Test Execution Time Determination" section.

2. Synchronize the sys_test_done BAP output using a synchronizer cell clocked by sys_test_clock or other appropriate functional clock. The synchronized sys_test_done output can then be used to sample the sys_test_pass as well as the sys_ctrl_pass BAP outputs. This method is useful as the system does not need to store all possible test time values, and yet samples test results as soon as they are available. However, a time out value corresponding to the longest possible test time should be stored by the system in case a controller does not reach the Done state.

3. Also synchronize the sys_test_pass BAP output as described in method #2. This is only necessary if the system needs to be notified as soon as possible of a failure on any of the controllers.

In the multi-sequencer case of Figure I-2, synchronizers of method #2 and #3 are not necessary as the sys_test_done and sys_test_pass outputs are generated for each clock domain.

BAP Requirements, Assumptions and Limitations

The following are the requirements, assumptions and limitations for implementing and utilizing advanced BAP memory access in Tessent Shell:

- Applying in-system memory tests is a destructive process. The memory content, prior to the memory being taken off-line, is overwritten.

- This feature only supports simulation patterns with controllers using a run_mode of hw_default, running Go/NoGo tests (compare_go:on). The pass/fail and completion status of the test and individual controllers are reported. Operating modes that require shifting the controller setup chain are not supported.

- The BAP ports must be connected to the system control logic before performing logic synthesis and scan insertion.

- When automating the BAP connections to the system control logic through the DftSpecification, the system side port or pin must be pre-existing in the design.

- The tool does not create patterns to configure the system control logic nor to operate the BAP module for in-system use. For sign-off verification, a testbench can be created through the PatternsSpecification to illustrate the protocol that must be implemented by the system control logic.

- The BAP direct access interface selection input ports are created when at least one controller implements the corresponding selection feature in the controller DftSpecification ExecutionSelections wrapper.

- When the BistAccessPort/DirectAccessOptions/direct_access_clock_source property is set to "common", the system logic clock connected to the direct access interface sys_clock input port should be at least four times slower than the lowest memory BIST clock frequency. The slower clock has no impact on test time and avoids having to constrain additional paths.

Advanced BAP Implementation

The following sections provide details on implementing the advanced BAP memory access feature in hardware, and verifying it in simulation. Information on ensuring timing closure for synthesis with the implementation of this feature is also provided.

Inserting the BAP Direct Access Interface

The following sections describe how to configure the DftSpecification to insert the BAP direct access interface and make connections to the system control logic:

BAP and Controller DftSpecification Options

The BAP module is generated and inserted in the current DFT insertion pass, together with the MemoryBIST instruments. For inclusion of the BAP direct access interface, configure the DirectAccessOptions wrappers in the MemoryBist section of the DFT Specification. The material that follows outlines various options for implementations that range from basic to more advanced.

In a basic setup, the BAP is automatically equipped with additional ports to perform in-system memory test and to report the test status to the system control logic. The memory test is performed using the default settings defined in the memory library and DFT specification. You are able to test all memories at once, or a subset of memories based on controller assignment. A single sequencer is implemented to manage the test for all controllers attached to the BAP. The single clock to the sequencer must be at least four times slower than the lowest memory BIST clock frequency.

```
DftSpecification {
  MemoryBist {
    BistAccessPort {
      DirectAccessOptions {
        direct_access: on;
      }
    }
  }
}
```

For flexibility in configuring the in-system test, you can specify controller features that are directly configured from the BAP. These controller features enable applying the test on a subset of memories and changing the test algorithm and operation set. You can configure individual controllers using the ExecutionSelections wrapper in the MemoryBist/Controller/ DirectAccessOptions section of the DFT specification. In the following example, the BAP direct access interface is equipped to select a group of memories tested concurrently or a single memory within controller c1. Additional IJTAG TDRs are added into the BAP for control during manufacturing test. Additional input ports are added on the BAP for control during in-system test.

```
DftSpecification {
  MemoryBist {
    BistAccessPort {
      DirectAccessOptions {
        direct_access: on;
      }
    }
    Controller(c1) {
      DirectAccessOptions {
        ExecutionSelections {
          step: on;
          memory: on;
        }
      }
    }
  }
}
```

The selection features at the controller level can be used for applications independent of the
BAP direct access interface. Because IJTAG TDRs are added into the BAP, they can be
programmed during manufacturing test or during in-system test using the MissionMode
controller. The advantage of implementing the additional IJTAG TDRs are test time reduction.
Selecting the controller step, memory, algorithm or operation set can be achieved by
programming the BAP TDR instead of serially loading the configuration chain through the
controllers.

Depending on your test methodology, the system side ports of the BAP may not be wanted. You
can disable the BAP direct access interface by setting direct_access: off in the MemoryBist/
BistAccessPort/DirectAccessOptions section of the DFT specification. In the following
example, the BAP direct access interface is not implemented, but is equipped with an IJTAG
TDR to select the controller step and memory within controller c1.

```
DftSpecification {
  MemoryBist {
    BistAccessPort {
      DirectAccessOptions {
        direct_access: off;
      }
    }
    Controller(c1) {
      DirectAccessOptions {
        ExecutionSelections {
          step: on;
          memory: on;
        }
      }
    }
  }
}
```

For advanced in-system usage, you can use the ExecutionSelections wrapper in the
MemoryBist/BistAccessPort of the DFT specification to omit unneeded system-side ports from
the BAP direct access interface. In the following example, the BAP direct access interface is

able to perform in-system memory test on all memories at once, without the ability to select controller, algorithm or operation set. The BAP is built with IJTAG TDR to program the algorithm or operation set.

```
DftSpecification {
  MemoryBist {
    BistAccessPort {
      DirectAccessOptions {
        direct_access: on;
        ExecutionSelections {
          controller: off;
          algorithm: off;
          operation_set: off;
        }
      }
    }
    Controller(c1) {
      DirectAccessOptions {
        ExecutionSelections {
          algorithm: on;
          operation_set: on;
        }
      }
    }
  }
}
```

Clock Connections

The BAP direct access interface supports two different clocking schemes, each with a different impact on the connection and signaling requirements of sys_clock.

The clocking scheme used in the BAP direct access interface is specified with the DftSpecification BistAccessPort/DirectAccessOptions/direct_access_clock_source property. The clock connections are configured within the Connections/DirectAccess/ClockDomain wrapper.

When direct_access_clock_source is specified as common, a single test sequencer is created in the BAP that manages memory testing for all attached controllers. For this setting, the sys_clock connection to the system logic must be specified as follows:

```
DftSpecification {
  MemoryBist {
    BistAccessPort {
      Connections {
        DirectAccess {
          ...
          ClockDomain(-) {               // domain_label must be "-"
            clock : port_pin_name ;   // System logic clock
            ...
          }
        }
      }
    }
  }
}
```

The system logic clock frequency must be at least four times slower than the lowest memory BIST clock frequency. The slower clock frequency has no impact on test time because the clock is only used by the BAP test sequencer. The slower clock is necessary to provide enough time between different sequencer events and signal propagation to the controller, without having to constrain the timing paths.

When direct_access_clock_source specified as per_bist_clock_domain, a separate test sequencer is created in the BAP direct access interface for each memory BIST clock domain. The sys_clock port for each sequencer is automatically connected to the clock corresponding to the specified memory BIST clock domain label. For this setting, the clock connections are specified as follows:

```
DftSpecification {
  MemoryBist {
    BistAccessPort {
      Connections {
        DirectAccess {
          ...
          ClockDomain(mbist_clk_domain_1) { // Repeatable for each domain
              // The following is optional. An automatic connection is
              // made to the matching memory BIST controller
              // clock_domain_label clock
            clock : port_pin_name ;          // Optional
            ...
          }
          ClockDomain(mbist_clk_domain_2) {
            ...
          }
        }
      }
    }
  }
}
```

In either clocking scheme, the system logic driving sys_test_start can be from any clock domain because this BAP direct access interface input goes through a synchronizer, clocked by the respective BAP sequencer clock. All other BAP direct access interface inputs driven by the system logic should be in a stable state at the interface prior to sys_test_start being asserted.

System Connections

Connections from the BAP direct access interface to the system control logic can be automated to occur along with the generation and insertion of the BAP module during the DFT insertion pass.

The only exceptions for availability of automated connections are the reset and request inputs for memory clusters. Refer to the "BAP Shared Bus Direct Access Interface Pins" section for further information.

The system connections are configured within the Connections wrapper in the MemoryBist section of the DFT specification. To use the Connections wrapper, the BistAccessPort DirectAccessOptions/direct_access property must be set to on.

By default, the BAP direct access interface contains the system logic connection ports that are necessary to support the controller features enabled in the MemoryBist ExecutionSelections wrapper. If there is only a single memory BIST controller attached to the BAP, the controller selection ports are not present on the direct access interface.

The repeatable ClockDomain wrapper within the Connections wrapper, configures BAP direct access interface connections that can originate from different clock domains, as described in the Clock Connections section. The BAP direct access interface contain a set of ports for system logic connection, for each ClockDomain wrapper specified.

When making system logic connections, ensure that the Connections wrapper connections for controller_done (sys_ctrl_done), controller_pass (sys_ctrl_pass), ClockDomain/test_done (sys_test_done) and ClockDomain/test_pass (sys_test_pass) are to internal input pins or output ports of the current design. All other connections must be to internal output pins or input ports of the current design.

It is not allowed to duplicate usage for the specified design ports and pins. If a system control logic port or pin is wider than its counterpart on the BAP direct access interface, only the number of bits equaling the width of the direct access interface are connected to the control logic and the remaining control logic bits are unused. An error is issued if the system control logic port or pin is smaller in width than its counterpart on the BAP direct access interface

A warning is issued during DFT insertion if a BAP direct access interface output (such as controller_done, controller_pass, test_done, or test_pass) connects to a system control logic port or pin that already has a drive source. The tool disconnects the driver in this situation.

Verifying the BAP Direct Access Interface

The following sections describe how to create a simulation testbench that can be used to verify the BAP direct access interface using the appropriate selection codes for the incorporated features.

PatternsSpecification Options

The in-system memory test is controlled from the functional logic. The system control logic can be implemented and operated in various architectures and protocols. For example, it might be a standard bus or a CPU based interface. The tool does not create patterns to configure the system control logic or to operate the BAP module for in-system use. It is recommended to ensure the correct operation of the in-system test as part of your design verification flow.

To facilitate sign-off verification in the Tessent Shell flow, a simulation testbench can be created using the process_patterns_specification command to mimic the protocol that must be implemented by the system control logic. The testbench targets the system-side inputs and outputs of the BAP direct access interface. A series of Verilog force statements are applied to launch the selected controllers, wait for the test execution, and sample the memory test pass/fail and completion status outputs of the BAP.

You can generate such a testbench using the MemoryBist wrapper in the Patterns/TestStep section of the patterns specification. The access_protocol property instructs the tool to utilize the BAP direct access interface instead of the IJTAG network. In the following example, the memories associated with controllers c1 and c2 are tested using the default settings. The MBIST controllers are clocked at the functional period of 10ns. Assume the system control logic operates on a slow functional clock. One sequencer is implemented in the BAP. The BAP direct access interface is clocked at 40ns, which is four times slower than the lowest MBIST clock frequency.

```
PatternsSpecification(chip,rtl,direct) {
  Patterns (pat1) {
    ClockPeriods {
      clka : 10ns;
    }
    TestStep(test1) {
      MemoryBist {
        access_protocol : parallel;
        run_mode : hw_default;
        Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
          DiagnosisOptions {
            compare_go : on;
          }
        }
        Controller(chip_rtl_tessent_mbist_c2_controller_inst) {
          DiagnosisOptions {
            compare_go : on;
          }
        }
      }
    }
  }
}
```

Selection of algorithms, operation sets, controllers, steps, or memories can be configured through the patterns specification using the same syntax as for manufacturing test. The compatible memory BIST pattern generation options are shown in Table I-2.

Table I-2. Pattern Generation Options

Property/Wrapper	Selection
AdvancedOptions/apply_algorithm	A test algorithm for all memories.
AdvancedOptions/apply_operation_set	An operation set for all memories.
Controller wrapper	One or more active controllers. Enables a group of memories based on controller.
AdvancedOptions/freeze_step	One controller step for all controllers. Enables a group of memories tested concurrently within a controller.
AdvancedOptions/enable_memory_list	One or more memories within a controller step. Enables specific memory instances.
MemoryBist/AdvancedOptions/ retention_test_phase	Apply one of three phases for retention testing using the SMarchCHKB based library algorithms.
MemoryBist/AdvancedOptions/ preserve_bist_inputs	Maintain memories under BIST control throughout the current TestStep.
AdvancedOptions/ MemoryClusterOptions/ configuration_data	Override the configuration data value applied to the associated Shared Bus memory cluster interface.

In the following example, the BAP direct access interface only enables controller c1. Furthermore, a custom algorithm is applied to one specific memory. The memory under test is activated in the second controller step of controller c1.

```
PatternsSpecification(chip,rtl,direct) {
  Patterns (pat2) {
    ClockPeriods {
      clka : 10ns;
    }
    TestStep(test1) {
      MemoryBist {
        access_protocol : parallel;
        run_mode : hw_default;
        Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
          DiagnosisOptions {
            compare_go : on;
          }
          AdvancedOptions {
            apply_algorithm : hammer;
            freeze_step : 1;
            enable_memory_list : m2;
          }
        }
      }
    }
  }
}
```

When the execution selections at the controller level are implemented, the necessary controller ports are created and connected to the IJTAG TDR in the BAP. You can override the algorithm and operation set, and select step and memory by setting the controller inputs in parallel. The selections are global for all active controllers. The main advantage of the parallel broadcast is reducing the time to configure the memory test. For example, you can change the algorithm applied to all memories by configuring only the BAP. Without the advanced BAP memory access feature, the override must be programmed by serially shifting through all controller configuration chains.

You can achieve test time reduction for in-system test with the MissionMode controller and manufacturing test because both applications are based on the IJTAG protocol. During operation through the IJTAG network, the system-side ports of the BAP direct access interface are unused. When the access_protocol property is set to ijtag, you can utilize the parallel broadcast by setting run_mode to hw_default and specifying one or more AdvancedOptions properties of Table I-2 in the patterns specification. In the following example, the IJTAG pattern runs controllers c1 and c2. Furthermore, a custom algorithm is applied to all memories associated to both controllers. Because run_mode is hw_default, the algorithm is configured in the BAP only. The configuration chains of controllers c1 and c2 are not accessed.

```
PatternsSpecification(chip,rtl,direct) {
  Patterns (pat3) {
    ClockPeriods {
      clka : 10ns;
    }
    TestStep(test1) {
      MemoryBist {
        run_mode : hw_default;
        Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
          DiagnosisOptions {
            compare_go : on;
            compare_memory_go : on;
          }
          AdvancedOptions {
            apply_algorithm : movi;
          }
        }
        Controller(chip_rtl_tessent_mbist_c2_controller_inst) {
          DiagnosisOptions {
            compare_go : on;
            compare_memory_go : on;
          }
          AdvancedOptions {
            apply_algorithm : movi;
          }
        }
      }
    }
  }
}
```

Test Execution Time Determination

After launching the memory test, the system control logic must wait for the controllers to complete the algorithm execution and report their test pass/fail and completion status.

For a basic algorithm, the wait time can be manually calculated from the test sequence of the memories, the memory sizes, and the algorithm order (number of times that each memory location is assessed). Alternatively, the wait time can be obtained by examining the simulation testbench used to verify the BAP direct access interface. The procedure is described using the patterns specification example shown below.

```
PatternsSpecification(chip,rtl,direct) {
  Patterns (pat4) {
    ClockPeriods {
      clka : 10ns;
    }
    TestStep(test1) {
      MemoryBist {
        access_protocol : parallel;
        run_mode : hw_default;
        Controller(chip_rtl_tessent_mbist_c2_controller_inst) {
          DiagnosisOptions {
            compare_go : on;
          }
        }
      }
    }
  }
}
```

When the patterns specification is processed, each Patterns wrapper results in a simulation testbench file stored within the patterns directory of the TSDB. The Patterns wrapper name, pat4, forms the testbench file name:

```
tsdb_outdir/patterns/chip_rtl.patterns_direct/pat4.v
```

For Patterns(pat4) in the patterns specification, the TestStep(test1) wrapper translates to a Verilog task statement in the simulation testbench. The Verilog task statement implements the

protocol to operate the BAP direct access interface according to the TestStep settings. The TestStep wrapper name, test1, forms the prefix of the Verilog task name:

```
`timescale 1ns / 100ps
module TB
...
task test1_direct_access_test;
begin
  $display ($realtime,"ns: Start testing the parallel access protocol of
   BAP chip_rtl_tessent_mbist_bap_inst");
  force `SIM_INSTANCE_NAME.chip_rtl_tessent_mbist_bap_inst.sys_test_start = 1'b1;
  force `SIM_INSTANCE_NAME.chip_rtl_tessent_mbist_bap_inst.sys_test_init = 1'b1;
  force `SIM_INSTANCE_NAME.chip_rtl_tessent_mbist_bap_inst.sys_reset = 1'b1;
  force `SIM_INSTANCE_NAME.chip_rtl_tessent_mbist_bap_inst.sys_ctrl_select=2'b01;
  #(7*40.0);
  #737660.0;
  $display ($realtime,"ns: Check the testing results of BAP
    chip_rtl_tessent_mbist_bap_inst");
  if ( `SIM_INSTANCE_NAME.chip_rtl_tessent_mbist_bap_inst.sys_test_pass == 1'b1 )
    ...
  end
  if (`SIM_INSTANCE_NAME.chip_rtl_tessent_mbist_bap_inst.sys_test_done == 1'b1)
    ...
  end
  ...
end
endtask
...
end
endmodule
```

Within the Verilog task statement, test1_direct_access_test, a series of Verilog force statements activate the system-side inputs of the BAP. Once the test is started, the Verilog delay statements provide the wait time corresponding to the algorithm execution. After that, the status outputs of the BAP are compared. Running controller c2 requires 280 + 737660 time units. At the top of the testbench file, the timescale statement defines the time unit as 1 ns. Therefore, the test execution time is 737940 ns.

Selection Codes

System logic sources selection codes to the BAP direct access interface ports for the controller features that have been configured in the DftSpecification. These selection codes identify which controller, step, memory, algorithm and operation set are to be used for the in-system test. MemoryBIST controllers can be individually configured in the DftSpecification to support any combination of direct access step, memory, algorithm or operation set selection capabilities.

Once the DftSpecification is created, the selection codes are available within the file *design_name_design_id_*tessent_mbist_bap.direct_access_dictionary located in the TSDB in the tsdb_outdir/instruments/*design_name_design_id_*mbist.instrument folder.

An example of the dictionary content is shown in Example I-1 below. The BAP direct access interface has been configured for controller, step, algorithm, operation set and memory selection capability. Two controllers, identified as c1 and c2, have also been configured to support all of these selections. The third controller, c3, is a shared bus controller and is configured to support controller and step selections.

The controller selection codes are one-hot encoded in the order listed. For this example, controller c1 has the selection code 3'b001 applied to the direct access interface sys_ctrl_select input ports, and controller c2 is selected with 3'b010. All controllers are selected with the code 3'b111.

Similarly, memory selection codes are also one-hot encoded. In this example, assume controller c1 (3'b001) and step(1) (2'b01) are selected for in-system test. A memory selection code of 2'b01 applied to the sys_memory_select input ports select the memory identified with an index of "0", and a code of 2'b10 select the memory identified with an index of "1". Both memories in the step are selected with the code 2'b11.

The dictionary identifies the design instance name for selectable non-shared bus memories, and an instance id for selectable shared bus memories. The instance id is a concatenation of the shared bus memory cluster TCD MemoryBistInterface*(id)*, LogicalMemoryToInterfaceMapping*(logical_memory_id)*, and finally the logical memory TCD PhysicalToLogicalMapping*(physical_memory_id)* if the DftSpecification memory_access_level is set, or resolves, to physical. In this example, assume controller c3 (3'b100) and step(2) (2'10) are selected for in-system test. A memory selection code of 2'b01 applied to the sys_memory_select input ports select the memory identified with an index of "0", which has an instance id of *I1_LM_1_LEFT_MEM*. This corresponds to a MemoryBistInterface id of "*I1*", a logical memory id of "*LM_1*", and a physical memory id of "*LEFT_MEM*" in the shared bus memory TCD.

Selection codes for the hard-coded algorithms and operation sets available in each controller are listed as follows:

```
Algorithm(algorithm_name) selection_code
OperationSet(operationset_name) selection_code
```

Algorithm selection codes are applied to the sys_algo_select input ports of the direct access interface and OperationSet selection codes are applied to the sys_opset_select input ports.

Example I-1. BAP Direct Access Dictionary Contents

```
set bap_direct_access_dictionary {
  controller(c1) {
    ControllerSelectionBit 0
    Algorithm(WRITE_SPEED_1R1W_UX7LS) 2'b00
    Algorithm(Y_CHECKER_1R1W_UX7LS) 2'b01
    Algorithm(X_MARCH_1R1W_UX7LS) 2'b10
    Algorithm(SMARCHCHKBCI) 2'b11
    OperationSet(OPERATIONS_1R1W_UX7LS) 2'b00
    OperationSet(NEC_SYNC_1R1W) 2'b01
    OperationSet(SYNC) 2'b10
    step(0) {
      MemorySelectionBit(SYNC_1R1W_16x8_iA) 0
    }
    step(1) {
      MemorySelectionBit(SYNC_1RW_32x16_RC_BISR_iA) 0
      MemorySelectionBit(SYNC_1RW_32x16_RC_BISR_iA) 1
    }
    step(2) {
      MemorySelectionBit(SYNC_2R1W_16x8_iA) 0
    }
  }
  controller(c2) {
    ControllerSelectionBit 1
    Algorithm(WRITE_SPEED_1R1W_UX7LS) 2'b00
    Algorithm(Y_CHECKER_1R1W_UX7LS) 2'b01
    OperationSet(OPERATIONS_1R1W_UX7LS) 2'b00
    OperationSet(NEC_SYNC_1R1W) 2'b01
    step(0) {
      MemorySelectionBit(SYNC_1R1W_16x8_iA) 0
      MemorySelectionBit(SYNC_2R1W_16x8_iA) 1
    }
    step(1) {
      MemorySelectionBit(SYNC_1RW_32x16_RC_BISR_iA) 0
    }
  }
  controller(c3) {
    ControllerSelectionBit 2
    step(0) {
      MemorySelectionBit(I1_LM_0_LOWER_MEM) 0
    }
    step(1) {
      MemorySelectionBit(I1_LM_0_UPPER_MEM) 0
    }
    step(2) {
      MemorySelectionBit(I1_LM_1_LEFT_MEM) 0
      MemorySelectionBit(I1_LM_1_RIGHT_MEM) 1
    }
  }
}
```

If there is only one controller configured for advanced BAP memory access, the BAP direct access interface will not contain sys_ctrl_select input ports for controller selection.

Parallel Retention Testing

Through the system-side ports of the BAP, you can enable parallel retention test using the SMarchCHKB based library algorithms.

The BAP must be equipped with the control signals by setting the ExecutionSelections/ retention_test_phase property to on in the DFT Specification. Retention test is typically performed concurrently on all memories. Table I-3 shows the sequence of events and the test phase selection codes for the sys_retention_test_phase inputs. The "00" code disables retention testing.

Table I-3. Retention Test Phases

Step	Event	Library Algorithm Phase Execution	sys_retention_test_phase[1:0]
1	Load checkerboard background	start_to_pause	01
2	Apply first retention pause	n/a	n/a
3	Read checkerboard background. Load inverse checkerboard background.	pause_to_pause	10
4	Apply second retention pause	n/a	n/a
5	Read inverse checkerboard background	pause_to_end	11

In addition to setting the test phase selection code for each retention phase, the sys_test_start signal should be asserted during each retention phase execution, and deasserted during the retention pauses and at the conclusion of the test. Finally, the sys_test_init and sys_preserve_test_inputs signals should be set to 1 for the entire retention test. For more information, refer to the "Parallel Static Retention Testing" Appendix.

You can create a simulation testbench to run parallel retention test using the SMarchCHKB based library algorithms. In the following example, the BAP direct access interface applies the SMarchCHKBvcd algorithm to the memories associated with controller c1. The TestStep wrappers implement the retention test phases of steps 1, 3, and 5 in Table I-3 above. The ProcedureStep wrappers represent the retention pauses of steps 2 and 4 in the same table.

Figure I-6. Example BAP Direct Access Parallel Retention Test

```
Patterns (prt) {
  ClockPeriods {
    clka : 10ns;
  }
  TestStep(write_ckb) {
    MemoryBist {
      access_protocol : parallel;
      run_mode : hw_default;
      AdvancedOptions {
        retention_test_phase : start_to_pause; //sys_retention_test_phase= 01
      }
      Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
        DiagnosisOptions {
          compare_go : on;
        }
        AdvancedOptions {
          apply_algorithm : SMarchCHKBvcd;
        }
      }
    }
  }

  ProcedureStep(pause1) {
    wait_time : 500ns;
  }
  TestStep(read_ckb_write_inv_ckb) {
    MemoryBist {
      access_protocol : parallel;
      run_mode : hw_default;
      AdvancedOptions {
        retention_test_phase : pause_to_pause; //sys_retention_test_phase= 10
      }
      Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
        DiagnosisOptions {
          compare_go : on;
        }
        AdvancedOptions {
          apply_algorithm : SMarchCHKBvcd;
        }
      }
    }
  }
```

```
      ProcedureStep(pause2) {
        wait_time : 500ns;
      }
      TestStep(read_inv_ckb) {
        MemoryBist {
          access_protocol : parallel;
          run_mode : hw_default;
          AdvancedOptions {
            retention_test_phase : pause_to_end; //sys_retention_test_phase= 11
          }
          Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
            DiagnosisOptions {
              compare_go : on;
            }
            AdvancedOptions {
              apply_algorithm : SMarchCHKBvcd;
            }
          }
        }
      }
    }
```

Timing Closure Considerations

Insertion of the advanced BAP memory access feature may require some additional considerations to obtain timing closure, depending on the clock source implementation.

The clocking scheme used in the BAP direct access interface is specified with the DftSpecification BistAccessPort/DirectAccessOptions/direct_access_clock_source property. When specified as common, a single test sequencer is created in the BAP that manages memory testing for all attached controllers. For this setting, the absolute value of the delays from the BAP outputs to the MBIST controllers are not important. However, the signals should arrive in the order described in the "Timing Diagram" section. This means that the difference in propagation delays cannot exceed one period of sys_clock. Selecting a relatively slow speed clock similar to TCK (typically at tens of MHz) makes it relatively easy to meet this requirement without having to analyze the detailed timing. When specified as per_bist_clock_domain, a separate test sequencer is created in the BAP for each memory BIST clock domain.

The timing closure considerations for each of these settings is as follows:

direct_access_clock_source: common

- Cross domain paths exist from the BAP sequencer to the memory BIST circuits. Declare false paths from the branch of sys_clock in the BAP to the functional clocks used in the memory BIST logic.

- In the v2018.1 release, a persistent buffer is instantiated immediately after the sys_clock input port. The clock buffer is preserved through synthesis to apply the timing exception on the netlist for static timing analysis.

- In the v2018.2 release, the SDC file declares a generated clock at the clock buffer output, and includes a generic false path from the generated clock to the functional clocks reaching memory BIST controllers attached to the same BAP.

direct_access_clock_source: per_bist_clock_domain

- The BAP sequencer and memory BIST controller(s) are on the same clock domain. A timing path exists from the BAP sequencer, through the controller BIST_SO output and back to the capture flop in the BAP TDR. This timing path is false and is covered by the existing timing exception between the IJTAG TCK and BIST clock.

Implementing memory BIST using Tessent MemoryBIST products requires special memory models, often referred to as memory library files or memory TCD files. The abstract models are needed to describe test-specific memory attributes that enable the Tessent MemoryBIST tool to create built-in test hardware with high defect test coverage. The memory library files do not provide any behavioral representation of the memory. However, there is a need to verify that the memory attributes are correctly defined and are consistent with the memory functional behavior.

This section describes how you can automate the validation of the memory library files using the memlibCertify utility. The intention is to have memory providers (companies or groups within a company) that are responsible for development of embedded memories (compiled or custom) certify the creation of all memory library files before the designer tries to use them in a design. This certification process verifies that the memory test attributes are correctly defined and are consistent with the memory behavioral model.

Memory library files are presented to the tool in the form of Tessent Core Description (TCD) files. For further information on the memory TCD file structure and content, refer to the "Memory" topic in this manual.

_____ **Note** _____
The legacy LogicVision memory library format is supported natively and is automatically translated into the memory TCD format when read. Note that Tessent Shell requires the LogicVision MemoryTemplate name to match the specified CellName as shown below:

```
MemoryTemplate(mydram) {
      MemoryType:     SRAM;
      CellName:       mydram;
      .
      .
      .
}
```

_____ **Note** _____
You can also use the memlibCertify utility to validate custom user-defined algorithms (UDAs).

The following topics are covered in this section:

Certification Steps for Memories without Redundancy

This procedure describes the steps needed to validate the memory TCD files for RAM and ROM memories that do not incorporate memory repair. The process makes use of the memlibCertify utility to automate the validation as much as possible. The memlibCertify utility also provides physical mapping data validation between memory vendor bitmap files and matching memory TCD files in the Tessent Shell flow.

Prerequisites

- If a memory TCD is being certified for a ROM, you need to provide a ROM contents file. The contents of this file is described in the RomContentsFile property description of the Memory TCD wrapper.

- Memory TCD files and corresponding valid Verilog simulation models for all unique memory types that need to be validated.

- A memory vendor bitmap file for each memory module to validate the address and data mapping that is described in the PhysicalAddressMap and PhysicalDataMap wrappers of the memory TCD.

Procedure

1. **Prepare the Environment** — Create the necessary directory structures and copy all relevant memory library files and Verilog models to the corresponding directories.

   The memlibCertify utility is flexible in that it supports repeatable arguments that can point to a variety of memory library file folders and Verilog simulation model folders. This enables one to organize folders as necessary. A simplistic directory structure could be implemented as shown below, where all the memory library files are located in the

memlib directory, and the corresponding Verilog simulation models are located in the *verilog* directory.

2. **Generate Tessent Shell dofiles** — Run the memlibCertify utility to generate the dofiles that create the DFT hardware, patterns and testbenches needed, as well as run and check simulations to complete the verification for each memory. A makefile is created that automates the certification process.

   Refer to the memlibCertify utility documentation to determine the proper runtime options specific to your implementation.

   The following example shows a typical invocation of memlibCertify:

   ```
   % memlibc      -memLib memlib/singlePort/spram.tcd_memory
                  -memLib memlib/dualPort/dpram.lvlib
                  -simModelDir verilog/libs/90nm/rams
                  -simModelDir verilog/libs/90nm/roms
                  -extension v:vg
                  -romFileExtension romdata
                  -romFileDir memlib/roms/rom.tcd_memory
                  -bitMapFile memlib/singlePort/spram.bitmap
   ```

   This example instructs memlibCertify to process three memory library files (*spram.tcd_memory*, *dpram.lvlib* and *rom.tcd_memory*). Tessent Shell natively supports legacy LogicVision memory library format and these files can be processed by memlibCertify, as shown in the example with *dpram.lvlib*. The specified search paths and file extensions are appropriately forwarded into the generated dofiles.

   The ROM content files with the specified file extensions are searched for within the directories specified with the -romFileDir switch and the file names are loaded into the generated *dft_spec.do* dofile. If you do not specify the -romFileDir property, the utility searches for ROM content files with the specified file extensions within the directories specified by the -memLib properties. Refer to the "memlibCertify Output" section for further information on the generated dofiles and makefile. For more information on using ROM content files, refer to the memlibCertify -romFileExtension switch description as well as the tool help contents viewed with the -help switch.

> **Note**
>
> For ROMs, *file_path_name* entries for the RomContentsFile property in the Memory library file are ignored. The *file_path_name* is specified in the DftSpecification which is configured by the *dft_spec.do* dofile. For information on the proper formatting of a ROM contents file, refer to the RomContentsFile property description.

For this example, the memlibCertify invocation is also specifying the vendor memory bitmap file for the single port RAM with the -bitMapFile switch. This will enable physical mapping data validation between the PhysicalAddressMap and PhysicalDataMap defined in the memory library file and the bit cell arrangement defined in the vendor bitmap file for this memory. This validation is described further in the next step. Refer to the memlibCertify -bitMapFile, -bitMapDir, and -reportBitMapFileMatchingOnly switch descriptions for more information on physical mapping data validation options.

3. **Generate RTL** — Generate Tessent MemoryBIST hardware (RTL) based on the content of the provided memory library files with the *dft_spec.do* file that is generated by memlibCertify.

   After creating the Tessent MemoryBIST dofiles and makefile, the next step is to generate memory BIST hardware that is used to verify the memory library file.

   Either of steps 3.a or 3.b shown below can be utilized to generate the memory BIST hardware. If you are certifying ROM memory library files, you also need to complete step 3.c.

   a. **Generation from Tessent Shell Invocation**

      From the working directory, enter:

      ```
      %tessent -shell -dofile dft_spec.do
      ```

   b. **Generation from makefile**

      From the working directory, enter:

      ```
      %make gen
      ```

   c. **Additional Step for Handling ROM Errors**

      If a ROM memory library file is included in the certification and a corresponding ROM contents file was not located, the *dft_spec.do* execution will return an error. In this case, the user is prompted to edit the *dft_spec.do* dofile to add the appropriate memory library module name and ROM contents file path and name to the set_rom_content_files_list variable defined in the dofile. The dofile can then be re-run as described in steps a) or b) above.

Upon completion of Steps 3.a or 3.b, and 3.c for ROMs, the generated RTL contains the following:

- One memory BIST controller for all ROM memories

- One memory BIST controller for all RAM memories

- One BIST Step for each memory within the appropriate controller

The generation step invokes Tessent Memory BIST tools to perform syntax checking of the memory library files. The tools also apply specific rule checking on the memory library description. The Tessent Shell log file *tshell.log_dft_spec* is created in the working directory during this step and reports any errors or warnings that exist.

You must fix any syntax errors in the memory library file to enable the tools to generate the hardware.

The rule checks are necessary to ensure correct operation of the BIST controller. They include checking for missing writeEnables for RAMs, application of incompatible address buses of multi-port memories, and many others.

Some rules may trigger warnings too. These warnings are indications of something missing in the memory library files or property definitions that are incompatible with Tessent MemoryBIST. You must examine all warnings and investigate their sources before proceeding to the next step.

The generated hardware is placed in the Tessent Shell Data Base (TSDB) structured sub-directory named *tsdb_outdir* within the working directory.

The memory vendor bitmap file indicates the XY coordinates of the bit cells within the memory array. If the vendor bitmap files are provided to memlibCertify, this procedure step will correlate the PhysicalAddressMap and PhysicalDataMap equations with the arrangement defined in the vendor bitmap file. For a memory wrapper that instantiates a single memory macro, the validation will be performed on the vendor bitmap of the macro.

The validation results are reported into the Tessent Shell log file. If mismatches are found between the address calculated from the PhysicalAddressMap equations and the vendor bitmap file, they are listed in a file named

<memory_module_name>.address_map_mismatch in the current directory. An example mismatch report is shown below:

```
==================================================
|        Controller        |       Memory         |
|   Physical Address       |   Logical Address    |
|      (Decimal)           |    (Hexadecimal)     |
==================================================
|                        | Before   After    From  |
|                        | Physical Physical Vendor |
|                        | Address  Address  Bitmap |
|  Bank   Row   Column   |   Map      Map     File   |
==================================================
    1      0      0           8        e        8
    1      0      1           9        f        9
    1      1      0           a        c        a
    1      1      1           b        d        b
    1      2      0           c        a        c
    1      2      1           d        b        d
    1      3      0           e        8        e
    1      3      1           f        9        f
    1      4      0          18       1e       18
    1      4      1          19       1f       19
    1      5      0          1a       1c       1a
    1      5      1          1b       1d       1b
    1      6      0          1c       1a       1c
    1      6      1          1d       1b       1d
    1      7      0          1e       18       1e
    1      7      1          1f       19       1f
```

4. **Generate Testbenches and Patterns** — Generate the testbench and test patterns that are used to verify that the Tessent MemoryBIST hardware generated in Step 3 operates correctly when compared against the memory behavioral model.

   Either of steps 4.a or 4.b shown below can be utilized to generate the simulation testbenches and patterns that are used to run Verilog simulations:

   a. **Testbench Generation from Tessent Shell Invocation**

      From the working directory, enter:

      ```
      %tessent -shell -dofile patterns_spec.do
      ```

   b. **Testbench Generation from makefile**

      From the working directory, enter:

      ```
      %make testbench
      ```

   The Tessent Shell log file *tshell.log_patterns_spec* is created in the working directory during this step and reports any errors or warnings that were encountered while processing the patterns specification. This file must be checked to make sure the creation of the testbench and patterns were successful. The patterns folder is also created in the tsdb_outdir folder.

5. **Simulate and Verify** — Run the simulations to verify the Tessent MemoryBIST generated hardware against the memory behavioral model.

    Either of steps 5.a or 5.b shown below can be utilized to run the simulations and verify the memory library files included in the certification.

    a. **Run Simulations from Tessent Shell Invocation**

        From the working directory, enter:

        ```
        %tessent -shell -dofile simulations.do
        ```

    b. **Run Simulations from the makefile**

        From the working directory, enter:

        ```
        %make sim_rtl
        ```

    The Tessent Shell log file *tshell.log_simulations* is created in the working directory during this step and reports any errors or warnings that were encountered, as well as the simulation results reported by the check_testbench_simulations command. The simulation files are archived in new sub-folder named simulation_outdir in the working directory.

    The *tshell.log_simulations* file should be checked to make sure the simulations ran without any errors. The simulated testbenches should not contain any compare failures. A compare failure indicates that the memory library file is not correctly modeling the behavior of the memory model.

6. **Optional Shortcut**

    Steps 3 through 5 can be combined into a single execution to further automate the process. This assumes that any special handling has already been incorporated into the dofiles generated by memlibCertify, such as editing *dft_spec.do* to add the needed rom_contents_file property for ROMs.

    The following can be utilized to process all of the steps in one invocation:

    From the working directory, enter:

    ```
    %make all
    ```

Results

A successful certification shows a pass status with no miscompares reported for each pattern, as shown in the example below. This report is archived in the *tshell.log_simulations* file in the working directory after the completion of Step 5.

```
//   command: check_testbench_simulations -report -design_name ${design_name} -design_id rtl

// Simulation status for
//      ./simulation_outdir memlibc_memory_bist_assembly_rtl.simulation_signoff
// =========================================================================
// ---------------------------------  ------  -----------  -----------  -------
// Pattern Name                       Status  Unexpected   Missing      Date
//                                            Miscompares  Miscompares
// ---------------------------------  ------  -----------  -----------  -------
// ICLNetwork                         pass         0            0       Wed Jan 24 xx PST 2018
// MemoryBist_P1                      pass         0            0       Wed Jan 24 xx PST 2018
// MemoryBist_ParallelRetentionTest_P1  pass       0            0       Wed Jan 24 xx PST 2018
//
//   command: exit
```

Certification Steps for Memories with Redundancy

Certifying memories with redundancy (row, column, or row and column) requires extra steps.

For memories with redundancies, the memory library file contains a RedundancyAnalysis wrapper describing the repair scheme and fuse mapping information. The description of the redundancy analysis wrapper must be verified separately from the other memory library file contents that are verified using the same method outlined in the "Certification Steps for Memories without Redundancy" section. The verification of the redundancy analysis wrapper is based on simulations with Fault Injected memory models that exercise Tessent MemoryBIST Built-In Repair Analysis (BIRA) circuitry and validate that the information specified in the RedundancyAnalysis wrapper of the memory library file is accurate. Faults are injected at a particular address in the memory, and the repair status and fuse register contents are extracted and examined. You must make sure that the status register and fuse register contents are as expected and map to the address location where the fault was injected.

The steps described in this procedure enable the needed BIRA capabilities and make changes to the testbench simulation to inject faults into memories. The final step outlines the analysis of the failures.

Prerequisites

- Completion of the "Certification Steps for Memories without Redundancy" steps to verify proper function of the memory library file other than the RedundancyAnalysis wrapper. All files and folders created are retained to complete verification of the RedundancyAnalysis wrapper contents.

- A verilog simulation model for the memory being certified that includes fault injection features. The fault injection implementation in the memory model is vendor specific. For more details, consult the documentation from your memory provider.

  The example task below shows fault injection in a model that enables setting a fault on a particular address and IO. The task sets registers that are used in the behavioral logic to

inject the faults. These fault injection registers, when enabled, force a defect to be injected on the memory data input during the write cycle.

```
task injectSA;
  input [ROW_BITS + COL_BITS - 1:0] Add;
  input [IO-1:0] FIO;
  begin
    if ( FaultNum === 'bx )
      FaultNum = 0;
    $display("** Fault[%2d]::Injecting fault at address %d on IO(s):
           %d",FaultNum,Add,FIO);
    FaultAddrEn[FaultNum] = 1;
    FaultAddr[FaultNum] = Add;
    FaultIO[FaultNum] = FIO;
    FaultNum=FaultNum+1;
  end
endtask
```

•

Procedure

1. **Load the Design and PatternsSpecification** — The Tessent Shell environment from the prior certification run is loaded into a new session.

   In the working directory, invoke Tessent Shell and run the following commands to load the design and PatternsSpecification data from the prior certification run that covered the normal RAM functions in the memory library file:

   ```
   %tessent -shell
   SETUP>set_context patterns -ijtag
   SETUP>set design_name memlibc_memory_bist_assembly
   SETUP>set patterns_spec_file_path [get_tsdb_output_directory]/patterns/
   ${design_name}_rtl.patterns_spec_signoff
   SETUP>read_design ${design_name} -design_id rtl -no_hdl
   SETUP>set_current_design ${design_name}
   SETUP>read_config_data ${patterns_spec_file_path}
   ```

   For clarity, the actual text output from the Tessent Shell commands are not shown in the example above.

2. **Enable Extraction of Repair Information** — Changes are made to the patterns specification to enable the extraction of repair information after a fault is injected.

   The presence of a RedundancyAnalysis wrapper in the memory library file enables the creation of the RepairOptions wrapper for the MemoryBist controller in the custom algorithm's TestStep wrapper. The check_repair_status property is already specified in this wrapper and set to "on", which enables sampling of the repair analysis STATUS_SHADOW registers from the controller SHORT_SETUP chain. The STATUS_SHADOW register values are duplicated from the BIRA chain STATUS register values, and provide the repair analysis results of the memory.

The commands shown below add the extract_repair_fuse_map property that enables sampling of the repair analysis fuse registers, as well as the STATUS register values from the BIRA chain.

_____ **Note** _____

Because executing check_repair_status or extract_repair_fuse_map clears their respective chains, running check_repair_status enables for checking the repair status from the STATUS_SHADOW registers without losing the repair solution contained in the BIRA chain.

For the purposes of this memlibCertify procedure, running extract_repair_fuse_map and losing the repair information in the BIRA chain is not an issue because a repair is not performed.

For memories with redundant row and column elements, the spare_element_priority property can be optionally added in the same manner. This property specifies the type of spare element to be allocated first upon failures.

_____ **Note** _____

The wrappers and ids presented below are valid for a single memory being certified with default naming.

```
SETUP>set patterns_spec [get_config_elements PatternsSpecification]
SETUP>set ctrl_wrap [get_config_elements -in $patterns_spec
Patterns(MemoryBist_P1)/TestStep(run_time_prog)/MemoryBist/Controller]
SETUP>set_config_value RepairOptions/extract_repair_fuse_map -in $ctrl_wrap
on
SETUP>set_config_value RepairOptions/spare_element_priority -in $ctrl_wrap
row
```

The resulting PatternsSpecification modifications are shown below and can be displayed with the report_config_data command.

```
Patterns(MemoryBist_P1) {
    ClockPeriods {
      BIST_CLK : 10.0ns;
    }
    TestStep(run_time_prog) {
      MemoryBist {
        run_mode : run_time_prog;
        reduced_address_count : off;
         Controller(mem_container_inst_memlibc_memory_bist_
                   assembly_rtl_tessent_mbist_c1_controller_inst) {
          DiagnosisOptions {
            compare_go : on;
            compare_go_id : on;
          }
          RepairOptions {
            check_repair_status : on;
            extract_repair_fuse_map : on;
          }
        }
      }
    }
    ...
}
```

3. **Generate Testbench** — View the PatternsSpecification and generate the updated testbench and patterns with the following commands:

```
SETUP>report_config_data $pattern_spec
SETUP>process_patterns_specification
SETUP>exit
```

4. **Create the Verilog Fault Injection Module** — The verilog model is created that interacts with the memory simulation model and performs the fault injection.

As an example, a verilog module named *FI.vb* is created and saved in the working directory. The contents of this file are shown below and is based on the existence of a fault injection feature in the memory simulation model, such as that described in the prerequisites.

```
module FI();
   initial begin
     #1;
     // Injecting fault at address 127, IO 3
     TB.DUT_inst.mem_container_inst.m1_mem_inst.injectSA(127,3);
   end
endmodule
```

The simulation path shown in the example to the fault injection task is valid for any single memory certification implementation. The user modifies the task name to what is present in the memory simulation model and adjusts the input parameters as needed to validate the RedundancyAnalysis wrapper contents.

5. **Simulate** — The simulations are run with the inclusion of the fault injection module.

The *simulations.do* file utilized in Step 5 of "Certification Steps for Memories without Redundancy" for this memory library file is edited to specify the simulation options that are needed to run the fault injection module along with the patterns. The following example shows the changes in bold that are needed to invoke the module described in Step 4:

```
run_testbench_simulations -design_name ${design_name} -design_id
rtl -extra_verilog_files FI.vb -extra_top_modules FI
```

The simulations can then be re-run using either of the methods outlined in Steps 5.a or 5.b of "Certification Steps for Memories without Redundancy".

The simulation fails because the fault injection causes unexpected miscompares. The check_testbench_simulations command can be run to get the simulation summary before exiting the tool session. An example simulation error output as well as the check_testbench_simulations invocation and result are shown below.
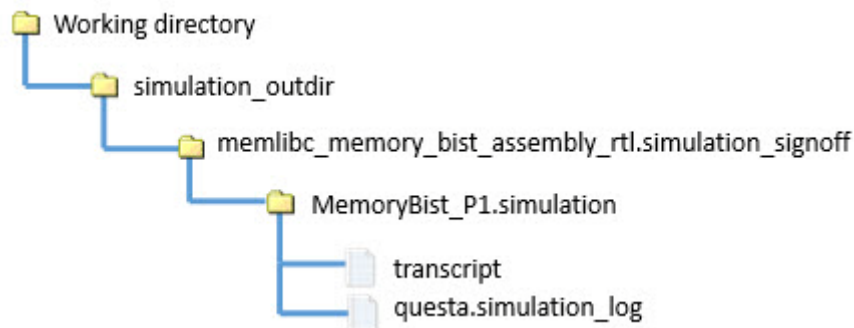
```
//  Error: 2 out of 4 simulations failed:
//        MemoryBist_P1 - 4 unexpected miscompares.
//        MemoryBist_ParallelRetentionTest_P1 - 6 unexpected miscompares.
// 'DOFile simulations.do' aborted at line 20
SETUP> check_testbench_simulations -report -design_name ${design_name} -design_id rtl

// Simulation status for ./simulation_outdir/
memlibc_memory_bist_assembly_rtl.simulation_signoff
// ==============================================================================
// -------------------------------   ------  -----------  -----------  ----------
// Pattern Name                      Status  Unexpected   Missing         Date
//                                           Miscompares  Miscompares
// -------------------------------   ------  -----------  -----------  ----------
// ICLNetwork                        pass        0            0           xxx
// MemoryBisr_BisrChainAccess        pass        0            0           xxx
// MemoryBist_P1                     fail        9            0           xxx
// MemoryBist_ParallelRetentionTest_P1  fail    6            0           xxx
//
SETUP> exit
%
```

6. **Analyze the Simulation Failure Results**

The failure results from the simulation are now analyzed against what is specified in the RedundancyAnalysis wrapper of the memory library file.

The Verilog simulation log files are located in the folder indicated in Figure J-1 for a certification performed on a single memory library file with a default tool setup. For Questa® SIM, either the *transcript* or *questa.simulation_log* files can be referenced.

Figure J-1. Verilog Simulation Log Files for Single Memory Certification

A portion of the Verilog simulation log for this example is shown in Figure J-2. For this TCD library certification case, the repair analysis engines are implemented in the memory interface. Each memory instance has a repair status register in the BIRA hardware named:

```
RA_INTERFACE_STATUS_REG[x]
```

In other usages, if the BIRA repair status registers were located at the controller, they would be named:

```
RA_<MemoryID>_STATUS_REG[x]
   where:
     <MemoryID> = DftSpecification MemoryInterface(<MemoryID>) or
                   the repair group name for repair sharing
                   implementations.
```

The repair status register specifies whether the memory requires a repair, is not repairable, or does not require a repair (when no failure is detected). The bit decode assignments for the repair status register are described in the table below:

Bit1 Bit0	Repair Status
00	No Repair Required
01	Repair Required
1x	Not Repairable

In this example, the simulation log shows that RA_INTERFACE_STATUS_REG[0] of the memory has a compare fail of expect 0 and simulated 1. This indicates that the memory repair status is "Repair Required".

Most repair analysis hardware have two registers for each spare element in each memory segment: the allocation register and the FuseSet register.

The allocation register contains a bit specifying whether a spare element is allocated or not. This register is always a single bit, where "0" represents NOT allocated, and "1" represents allocated. If allocated, the value in the FuseSet register represents the defective portion of the memory to be repaired.

Figure J-2. Partial Output of a Verilog Simulation Log

```
# Mismatch at pin          1 name ijtag_so, Simulated 1, Expected 0
# Previous scan out : pin ijtag_so =
mem_container_inst_m1_mem_inst_interface_inst.RA_STATUS_SHADOW_REG[0]
# Mismatch at pin          1 name ijtag_so, Simulated 1, Expected 0
# Previous scan out : pin ijtag_so =
mem_container_inst_m1_mem_inst_interface_inst.RA_INTERFACE_STATUS_REG[0]
# Mismatch at pin          1 name ijtag_so, Simulated 1, Expected 0
# Previous scan out : pin ijtag_so =
mem_container_inst_m1_mem_inst_interface_inst.RA_INTERFACE_rs_SPARE0_ALLOC_BIT[0]
# Mismatch at pin          1 name ijtag_so, Simulated 1, Expected 0
# Previous scan out : pin ijtag_so =
mem_container_inst_m1_mem_inst_interface_inst.RA_INTERFACE_rs_SPARE0_FUSE_ADD_REG[4]
# Mismatch at pin          1 name ijtag_so, Simulated 1, Expected 0
# Previous scan out : pin ijtag_so =
mem_container_inst_m1_mem_inst_interface_inst.RA_INTERFACE_rs_SPARE0_FUSE_ADD_REG[3]
# Mismatch at pin          1 name ijtag_so, Simulated 1, Expected 0
# Previous scan out : pin ijtag_so =
mem_container_inst_m1_mem_inst_interface_inst.RA_INTERFACE_rs_SPARE0_FUSE_ADD_REG[2]
# Mismatch at pin          1 name ijtag_so, Simulated 1, Expected 0
# Previous scan out : pin ijtag_so =
mem_container_inst_m1_mem_inst_interface_inst.RA_INTERFACE_rs_SPARE0_FUSE_ADD_REG[1]
# Mismatch at pin          1 name ijtag_so, Simulated 1, Expected 0
# Previous scan out : pin ijtag_so =
mem_container_inst_m1_mem_inst_interface_inst.RA_INTERFACE_rs_SPARE0_FUSE_ADD_REG[0]
```

For this example, as shown in Figure J-2 the
RA_INTERFACE_rs_SPARE0_ALOC_BIT indicates a spare element has been
allocated from the segment defined in the RowSegment(rs) wrapper of the memory
template, which is shown in Figure J-3. Similarly, the
RA_INTERFACE_rs_SPARE0_FUSE_ADD_REG[*] represents the fuse repair
address register defined in the FuseSet wrapper.

The fuse repair address register contains the fuse bits as specified by the FuseSet
wrapper for the segment. Each fuse bit logs a single memory address bit that all together
identify the defective memory element. In the example simulated,
FUSE_ADD_REG[4:0] are set to "1", which are mapped in the FuseSet wrapper to
memory address A[6:2]. For this memory, the column segment is determined by A[1:0],
so this row repair covers an overall address range of 10'b0001111100 to
10'b0001111111, or 124 to 127. The injected fault from Step 4 was to address 127,
which matches expectations and validates proper operation of the BIRA hardware.

Figure J-3. Example RedundancyAnalysis Wrapper (Partial)

```
RedundancyAnalysis {

    RowSegment(rs) {
        NumberOfSpareElements : 2;
        FuseSet {
            Fuse[0] : AddressPort(A[2]);
            Fuse[1] : AddressPort(A[3]);
            Fuse[2] : AddressPort(A[4]);
            Fuse[3] : AddressPort(A[5]);
            Fuse[4] : AddressPort(A[6]);
            Fuse[5] : AddressPort(A[7]);
            Fuse[6] : AddressPort(A[8]);
            Fuse[7] : AddressPort(A[9]);

        }
        PinMap {
            SpareElement {
                RepairEnable : RR0[8];
                Fuse[0]      : RR0[0];
                Fuse[1]      : RR0[1];
                Fuse[2]      : RR0[2];
                Fuse[3]      : RR0[3];
                Fuse[4]      : RR0[4];
                Fuse[5]      : RR0[5];
                Fuse[6]      : RR0[6];
                Fuse[7]      : RR0[7];

            }
            SpareElement {
                ...
            }
        }
    }
    ColumnSegment() {
        ShiftedIORange : Q[7:0];
        NumberOfSpareElements : 1;
        FuseSet {
            Fuse[0] : AddressPort(A[0]);
            Fuse[1] : AddressPort(A[1]);
          ...
        }
    }
}
```

7. **Repeat Steps 2 through 6 as needed to validate the RedundancyAnalysis wrapper contents**.

   Edit the contents of your fault injection module in Step 4 for interesting addresses and IO faults. For memories with redundant row and column elements, changing the spare_element_priority property setting in Step 2 may provide additional validation coverage of the repair functions defined in the RedundancyAnalysis wrapper.

memlibCertify Limitations

The memlibCertify utility has the following limitations:

- The legacy LogicVision memory library format is supported natively and is automatically translated into the memory TCD format when read. Note that Tessent Shell requires the LogicVision MemoryTemplate name to match the specified CellName as shown below:

```
MemoryTemplate(mydram) {
      MemoryType:    SRAM;
      CellName:      mydram;
      .
      .
      .
}
```

- The memlibCertify utility does not support validating memory cluster libraries.

- The utility does not automate the process of validating memories with redundancies. This process requires additional steps that are explained in the "Certification Steps for Memories with Redundancy" section.

- The utility can validate physical address and data mapping that are described in the PhysicalAddressMap and PhysicalDataMap wrappers of the memory library file. The process requires the vendor bitmap file for the corresponding memory module. For further information, refer to the memlibCertify utility description.

- The tool does not validate the MilliWattsPerMegaHertz property in the memory library file.

- Disabling the generation of BISR logic for memories that support self repair requires additional manual steps. For further information, refer to the memlibCertify utility description.

- By default, Tessent Shell utilizes Questa SIM from Siemens Digital Industries Software. For information on how to specify other simulators that are supported by Tessent Shell and how to pass simulator options, refer to the memlibCertify utility description.

memlibCertify Utility Usage in Tessent Shell

Note - Viewing PDF files within a web browser causes some links not to function. Use HTML for full navigation.

memlibCertify

A tool that automates the process of validating memory library files.

Usage

memlibc -**memLib** *filename*]...
      -**simModelDir** *dir_path* ...
      -**simModelFile** *file_path*...
      [-extension *ext1*[:*ext2*] ...]
      [-quiet]
      [-bitMapDir *dir_path*] ...
      [-bitMapFile *file_path*] ...
      [-reportBitMapFileMatchingOnly]
      [-verilogOptionFile *file_name*] ...
      [-romFileExtension *ext1*[:*ext2*]...]
      [-romFileDir *dir_path*[:*dir_path2*]...]
      [-help]

Description

memlibCertify is a tool that automates the process of validating memory library files. The tool can be used in either the LV flow or Tessent Shell flow. By default, memlibCertify generates the necessary ".do" files as well as a Makefile that can be utilized to complete the certification process for Tessent Shell flow. memlibCertify also provides physical mapping data validation between memory vendor bitmap files and matching memory TCD files in the Tessent Shell flow. For information on memlibCertify use in the LV flow, refer to the *Certifying Memory Library Files with memlibCertify* manual. For use in the Tessent Shell flow, refer to the other topics within "Certifying TCD Memory Library Files With memlibCertify in Tessent Shell" and the argument descriptions below.

You can pass simulator arguments and switches to the run_testbench_simulations command invoked within the *simulation.do* file created by memlibCertify. You can use this method to change the default Questa SIM simulator, tailor waveform and debug data, and specify simulator-specific options and other valid arguments for the run_testbench_simulations command as needed.

To utilize this feature, you specify and assign the *simOptions* variable when running "make sim_rtl" or "make all". For example, if you are using the default simulator Questa SIM, the method shown below will pass in the -debugDB simulator option to create the *vsim.dbg* file that the schematic viewer accesses when debugging your waveforms. You must specify this simulator option in conjunction with the -store_simulation_waveform switch for the

run_testbench_simulations command. Either of the examples shown below will pass these simulator options to the run_testbench_simulations command:

```
make sim_rtl simOptions="-store_simulation_waveforms on \
         -simulator_options -debugDB"

make all simOptions="-store_simulation_waveforms on \
         -simulator_options -debugDB"
```

Note that simulator specific options must be consistent with the simulator that the run_testbench_simulations command utilizes. The memlibCertify utility creates (or overwrites) a file named *SimOptions* in the current working directory that contains the contents of the latest *simOptions* Makefile variable assignment. The *simulations.do* dofile always checks if the *SimOptions* file exists, and if so, passes the contents to the run_testbench_simulations command.

The memlibCertify utility automatically generates BISR logic for memories that have the following wrappers in their memory library file:

- RedundancyAnalysis/ColumnSegment/PinMap

- RedundancyAnalysis/RowSegment/PinMap

If needed, you can prevent generation of Built-In Self-Repair (BISR) logic for memories that support self repair by editing the dft_spec.do file created by memlibCertify and adding the -memory_bisr_chains argument and setting as follows:

```
set_dft_specification_requirements -memory_test on
        -memory_bisr_chains off
```

Arguments

- -memLib *filename*

  Use this repeatable option to specify the name of the TCD memory library file. The memory library file might define one or more Memory, OperationSet, or Algorithm wrappers. All Memory wrappers in a given file are processed. This argument is mandatory and repeatable.

- -simModelDir *dir_path* ...

  When you have multiple simulation models in different files, you can put them all in one directory and then use this option to have memlibCertify search this directory. This argument is repeatable. It is mandatory to specify one of the -simModelDir or -simModelFile options. Both can be specified if needed.

- -simModelFile *file_path* ...

  When you have only one simulation model file, use this option to specify the file that contains the simulation models for memories or library cells. This argument is repeatable. It is mandatory to specify one of the -simModelFile or -simModelDir options. Both can be specified if needed.

- -extension *ext1*[:*ext2*]...

  Use this option to control the file extension naming (without the dot) used in conjunction with the -simModelDir option. These extensions define the order that files are searched within a directory to resolve a Verilog module name. Use colons to separate multiple extensions. This argument is optional and defaults to *vb*.

- -quiet

  Use this option to reduce the verbosity of memlibCertify when processing memory library files.

- -bitMapDir *dir_path*

  When you have multiple memory vendor bitmap files, you can put them all in one directory and use this option to have memlibCertify search this directory. The search is limited to files with a *.bitmap* extension and memlibCertify supports the ad hoc industry ASCII format, as described in "Adding Physical X-Y Coordinate Data to Bitmap Reports" in the *Tessent SiliconInsight User's Manual for Tessent Shell*. This argument is optional and repeatable, but you must specify either the -bitMapDir or -bitMapFile option to enable physical mapping data validation. The sequence in which directories are specified implies the search order.

- -bitMapFile *file_path*

  Use this option to explicitly specify the memory vendor bitmap file. The memlibCertify utility supports the ad hoc industry ASCII format, as described in "Adding Physical X-Y Coordinate Data to Bitmap Reports" in the *Tessent SiliconInsight User's Manual for Tessent Shell*. This argument is optional and repeatable, but you must specify either the -bitMapFile or -bitMapDir option to enable physical mapping data validation.

- -reportBitMapFileMatchingOnly

  Use this option to only produce a matching report between the TCD memory library names found by the -memLib specification and the specified memory vendor bitmap file names. The validation of the physical mapping data is not performed. Either the -bitMapDir or -bitMapFile option must be specified for the report to be generated, otherwise the argument is ignored.

- -verilogOptionFile *file_name*

  Use this option to specify a Verilog option file that will be read as part of the simulation step. This is useful for forwarding debug options into the testbench or to modify the simulation library sources. This argument is optional and repeatable, however if it is repeated, only the last occurrence will take effect. The memlibCertify utility adds the option "-f *file_name*" to reference the specified file for the set_simulation_library_sources command within the *simulations.do* file created for the certification simulations. If you do not specify a Verilog option file, or you specify a file that does not exist, memlibCertify creates an empty file called *VerilogOptions* in the current working directory and adds "-f VerilogOptions" to the set_simulation_library_sources command.

- -romFileExtension *ext1*[:*ext2*] ...

  Use this option to specify the list of file extensions (without the dot) used to identify the ROM data files. Use colons to separate multiple extensions. By default, the search is limited to the directories containing the memory library files, as specified with the -memLib option, unless additional locations are specified with the -romFileDir option. The memlibCertify utility searches for the ROM data file belonging to the MemoryTemplate defining a ROM memory type. The ROM data filename must match the ROM module name with the specified extension or extensions. The file search results will be populated into the dofile named *dft_spec.do*. This argument is optional and when it is specified, at least one extension must be specified to enable the automatic file search.

- -romFileDir *dir_path*[:*dir_path2*]...

  Use this option to specify a location the tool searches for ROM content files with the file name extensions specified by the -romFileExtension property. The -romFileDir argument is optional and *dir_path* is repeatable. Use colons to separate multiple *dir_path* values. The tool processes the first ROM content file found if the file is present in multiple directories specified by additional -romFileDir arguments. If -romFileDir is not specified, the search for ROM content files defaults to the directories containing the memory library files, as specified with the -memLib option.

- -help

  Use this option to display the tool help contents and runtime options.

memlibCertify Output

The memlibCertify utility creates the following files in the working directory:

- Tessent Shell dofiles needed to perform the certification process:

  - *dft_spec.do*

    Tessent Shell dofile that performs the DFT insertion steps for the specified memory libraries.

  - *patterns_spec.do*

    Tessent Shell dofile that creates the patterns and testbenches necessary for certification.

  - *simulations.do*

    Tessent Shell dofile that performs the simulations and completes the certification process.

- Makefile

  The memlibCertify utility generates a makefile that contains several make targets. These make targets can be used to automate the certification of the memory library file.

_____ **Note** _____

To see all make targets, type *make* in the current working directory.

```
%make
Target                      Description
---------------------------------------------------------------
make gen            Generate the memoryBist hardware for memory
                    templates.
make testbench      Generate the Verilog testbench for the
                    memoryBist controllers.
make sim_rtl        Runs the Verilog simulation for the
                    memoryBist controllers.
make clean          Deletes the TSDB and simulation output
                    directories.
make all            gen testbench sim_rtl
```

- *memlibc.log*

  Log file capturing the screen output from the memlibCertify utility execution.

- *VerilogOptions* file

  An empty *VerilogOptions* file is created by memlibCertify if the -verilogOptionFile argument is not specified, or specifies a nonexistent file. Refer to the memlibCertify -verilogOptionFile argument description for information on forwarding simulator-specific options that are to be used during simulation.

- *SimOptions* file

  The *SimOptions* file is created when the Makefile variable *simOptions* is specified and assigned when invoking "make all" or "make sim_rtl". The *SimOptions* file is created in the current working directory and contains the values assigned to the *simOptions* variable. Refer to the memlibCertify command description for details on how this file is used.

There are several ways to get help when setting up and using Tessent software tools. Depending on your need, help is available from documentation, online command help, and your Siemens representative.

The Tessent Documentation System

At the center of the documentation system is the InfoHub that supports both PDF and HTML content. From the InfoHub, you can access all locally installed product documentation, system administration documentation, videos, and tutorials. For users who want to use PDF, you have a PDF bookcase file that provides access to all the installed PDF files.

For information on defining default HTML browsers, setting up browser options, and setting the default PDF viewer, refer to the *Siemens® Software and Mentor® Documentation System* manual.

You can access the documentation in the following ways:

- **Shell Command —** On Linux platforms, enter **mgcdocs** at the shell prompt or invoke a Tessent tool with the -manual invocation switch.

- **File System —** Access the Tessent InfoHub or PDF bookcase directly from your file system, without invoking a Tessent tool. For example:

  HTML:

  ```
  firefox <software_release_tree>/doc/infohubs/index.html
  ```

  PDF:

  ```
  acroread <software_release_tree>/doc/pdfdocs/_tessent_pdf_qref.pdf
  ```

- **Application Online Help —** You can get contextual online help within most Tessent tools by using the "help -manual" tool command. For example:

  > help dofile -manual

  This command opens the appropriate reference manual at the "dofile" command description.

Global Customer Support and Success

A support contract with Siemens Digital Industries Software is a valuable investment in your organization's success. With a support contract, you have 24/7 access to the comprehensive and personalized Support Center portal.

Support Center features an extensive knowledge base to quickly troubleshoot issues by product and version. You can also download the latest releases, access the most up-to-date documentation, and submit a support case through a streamlined process.

https://support.sw.siemens.com

If your site is under a current support contract, but you do not have a Support Center login, register here:

https://support.sw.siemens.com/register

Third-Party Information

Details on open source and third-party software that may be included with this product are available in the *<your_software_installation_location>/legal* directory.