



A Siemens Business

Tessent MemoryBIST User's Manual

For Use with Tessent® Shell

Software Version 2019.1

March 2019

Document Revision 12

© 2015-2019 Mentor Graphics Corporation
All rights reserved.

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

Note - Viewing PDF files within a web browser causes some links not to function (see [MG595892](#)).
Use HTML for full navigation.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT LICENSE RIGHTS: The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: mentor.com/trademarks.

The registered trademark Linux® is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

End-User License Agreement: You can print a copy of the End-User License Agreement from: mentor.com/eula.

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: mentor.com
Support Center: support.mentor.com

Send Feedback on Documentation: support.mentor.com/doc_feedback_form

Revision History

Revision	Changes	Status/ Date
12	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the <i>Tessent Release Notes</i> for this product are reflected in this document. Approved by Ron Press.	Released Mar 2019
11	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the <i>Tessent Release Notes</i> for this product are reflected in this document. Approved by Ron Press.	Released Dec 2018
10	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the <i>Tessent Release Notes</i> for this product are reflected in this document. Approved by Ron Press.	Released Aug 2018
9	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the <i>Tessent Release Notes</i> for this product are reflected in this document. Approved by Ron Press.	Released May 2018

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 Mentor Graphics Technical Publication's source. For specific topic authors, contact Mentor Graphics Technical Publication department.

Revision History: Released documents maintain a revision history of up to four revisions. For earlier revision history, refer to earlier releases of documentation which are available at the following URL:

<http://support.mentor.com>

Table of Contents

Revision History

Chapter 1

Tessent MemoryBIST Overview	21
Why Consider Tessent MemoryBIST for Your Chip?	21
Tessent MemoryBIST Capabilities	22
Tessent MemoryBIST Library Algorithms	22
User-Defined Algorithms	23
Tessent MemoryBIST Library Operation Sets	23
Custom Operation Sets	24

Chapter 2

Getting Started	25
DFT Flow Using Tessent Shell	27
Prerequisites	28
Design Flow Dofile Example	28
Design Loading	30
Set the Context	30
Read the Libraries	31
Read the Design	32
Elaborate the Design	33
Report the Design Data	33
Specify and Verify DFT Requirements	36
Set DFT Specification Requirements	36
Add Properties and Constraints	37
Define Clocks	39
Run DRC	40
Create DFT Specification	42
Invoke create_dft_specification	42
Edit/Configure the DFT Specification According to Your Requirements	43
Validate the DFT Specification	44
Process DFT Specification	45
Create DFT Hardware with the DFT Specification	45
Extract ICL	46
Preparation for Pattern Generation	46
Create Patterns Specification	48
Automatically Created Patterns Specification	48
Edit/Configure the Patterns Specification According to Your Requirements	49
Process Patterns Specification	51
Process Patterns According to the Patterns Specification	51
Run and Check Test Bench Simulations	52
Run Simulations	52

Check Results	53
Formal Verification.....	53
Test Logic Synthesis	54
RTL Design Flow Synthesis.....	55
Using Generated SDC for MemoryBIST.....	55
Synthesizing the RTL Design with Test Logic	56
Gate Level Design Flow Synthesis.....	57
Run Synthesis.....	57
Concatenate Netlist Generation	57
Run and Check Test Bench Simulations with a Gate Level Netlist	57
Chapter 3	
Planning and Inserting MemoryBIST	59
Design and Library Requirements	59
Specifying and Verifying MemoryBIST Requirements	61
Enabling Memory BIST & Repair	61
Loading Layout Placement Information.....	62
Loading Power Domain Information	62
Adding Clocks	63
Changing Default Settings	63
Adding Constraints Before Design Rule Checking	65
Creating, Modifying, and Validating a MemoryBIST DFT Specification	69
Adding Constraints Before Creating an Initial DftSpecification	69
Parameter Selection Impacts on Performance and Area.....	70
MemoryBIST Partitioning Rules	73
Creating the DftSpecification	74
Review and Basic Edits of the DftSpecification.....	75
Re-Creating the DftSpecification	79
Validating the DftSpecification	82
Additional Editing Options of the DftSpecification	84
Invoking the GUI and Editing the DftSpecification.....	84
Validating a DftSpecification in the GUI.....	88
Editing the DftSpecification Using Shell Commands	89
Processing the DftSpecification.....	90
Chapter 4	
Creating and Verifying Test Patterns.....	93
Certifying TCD Memory Library Files with memlibCertify in Tessent Shell	94
Certification Steps for Memories without Redundancy	95
Certification Steps for Memories with Redundancy.....	101
Limitations	108
memlibCertify Utility Usage in Tessent Shell	109
memlibCertify.....	110
memlibCertify Output	112
Pattern Generation for TS-MBIST Insertions	113
Performing Pattern Generation	114
Generated MBIST Verification Patterns	117

Table of Contents

Chapter 5	
Implementing and Verifying Memory Repair	121
Overview of Memory Repair Capabilities.....	123
Types of Repair.....	123
Repair Steps	123
Chip Level Repair Architecture	123
Built-in Repair Analysis (BIRA)	125
Limitations and Restrictions.....	125
Built-in Self Repair (BISR)	126
Implementing Soft Repair.....	127
Memory Library Preparation and Repair Registers Description	129
Repair Analysis for IO/Column Elements	130
IO/Column Repair	131
Implementing IO/Column Repair Analysis	133
Repair Analysis Registers.....	143
Repair Analysis Output Ports for IO/Column Repair.....	145
Repair Analysis for Row Elements.....	147
Row Repair.....	148
Implementing Row Repair Analysis	149
Repair Analysis Registers.....	154
Repair Analysis Output Ports for Row Repair.....	156
Repair Analysis for Row and IO/Column Elements.....	158
Implementing Row and IO/Column Repair Analysis	159
Repair Analysis Registers.....	167
Repair Analysis Output Ports for IO/Column and Row Repair	168
Built-In Self-Repair	169
Parallel BISR Interface.....	169
Serial BISR Interface	169
BISR-Specific Memory Library Syntax.....	171
Memory Library Examples with Parallel and Serial BISR Interface	176
Implementing BIRA and BISR Logic	180
Inserting BISR Chains in a Block.....	181
Assigning Memories to Power Domains	181
Controlling the BISR Chain Order.....	182
Disabling the Insertion of BISR registers.....	184
Excluding Child Block BISR Chains	184
Determining the Fuse Box Size	186
Single-Chain Case	186
Multi-Chain Case	187
Incremental Repair Case.....	188
Creating and Inserting the BISR Controller	190
Understanding the BISR Controller	191
Connecting the BISR Controller to an External Fuse Box	193
Connecting a BISR Controller to System Logic	194
Choosing a Functional Repair Clock	195
Top-Level Verification and Pattern Generation	197
Fuse Box Programming	198
Autonomous Self Fuse Box Program	198

FuseBox Access	198
Verify BISR at the Block Level	200
Executing Fault-Inserted Memory BIST	202
Performing BIRA-to-BISR Capture	202
Executing Post-Repair Memory BIST	202
Verifying Top-Level BISR	203
FuseBox Access Mode	203
Autonomous Modes	204
Autonomous Mode for Memory BIST-Only Verification	206
Functional Mode Verification	208
Creating Multi-Load Scan Patterns with Repairable Memories	216
Generating Your Manufacturing Test Patterns	218
Initialization Test Pattern	219
Pre-Repair Test Patterns	219
Repair Test Patterns	221
Post-Repair Test Patterns	224
Tester Settings Considerations	225
Manufacturing Flow Variations	226
Compression Algorithm and Fuse Box Organization	229
CompressBisrChain Script	231
Incremental Repair	235
Incremental Repair Overview	235
BIRA Initialization	237
BIRA Repair Status Bits Checking	238
Absence of Fuse Programming Step	245
Handling of Blocks Without Incremental Repair Capability	245
Considerations Specific to Hard Incremental Repair	246
Repair Sharing	248
Repair Sharing Overview	248
Repair Sharing Conditions	248
Implementing Repair Sharing	252
Creating and Modifying Repair Share Groups	252
BISR Segment Order File	256
BISR Instance Location	257
Fast BISR Loading	259
Fast BISR Loading Overview	259
Fast BISR Architecture	259
Implementing Fast BISR Loading	264
Fuse Box Interface	264
DFT Insertion	265
Pattern Generation	266
Chapter 6	
Implementing MemoryBIST with Memory Shared Bus Interface	269
Limitations	270
Applying Memory BIST to a MemoryCluster	270
Shared Bus Support Features	273
Shared Bus Interface MemoryBIST Implementation Flow	275

Table of Contents

Library Requirements	276
Memory Cluster Core Library File	277
Logical Memory Core Library File	278
Physical Memory Core Library File	279
Design Loading	279
Specify and Verify DFT Requirements	279
Create DFT Specification	280
Process DFT Specification	282
Extract ICL	284
Create Patterns Specification	284
Process Patterns Specification	286
Run and Check Test Bench Simulations	288
Test Logic Synthesis	288
Handling MemoryCluster Modules with Repairable Memories	289
BIRA and BISR Generation for a Memory Cluster Module	289
Design and Library File Prerequisites for BIRA and BISR	291
Functional Design Preparation for BIRA and BISR	291
Memory Cluster Library File Preparation for BIRA and BISR	291
Chapter 7 Using Tessent User-Defined Algorithms	293
Usage Context	293
Overview & Terminology	293
Defining a Custom Algorithm	295
Example Algorithm: March C- Algorithm	296
Coding the Algorithm	297
TestRegisterSetup Wrapper	297
MicroProgram Wrapper	298
Algorithm Optimization	303
Making a UDA Available to TS-MBIST	303
Selecting a UDA for MBIST Execution	304
Chapter 8 Tessent MemoryBIST Diagnosis	309
Memory BIST Diagnosis Approaches	310
Diagnosis Objectives	310
Diagnosis Levels	310
Memory BIST Diagnosis Capabilities	311
Achieving Specific Diagnosis Level	312
Diagnosing Failing Memories Only	312
Diagnosing Failing Addresses and Bit-Mapping in a Memory	313
Enhanced Stop-On-Nth-Error Approach	314
Performing Enhanced Stop-On-Nth-Error Diagnosis for Bitmap Applications	315
Using Diagnosis with Local Comparators	316
Chapter 9 Common Implementation Flows	317
Top-Down Flow	317

Bottom-Up Flow	319
Appendix A	
Tessent Core Description	323
Core	325
Memory	327
Port	341
AddressCounter	352
PhysicalAddressMap	355
PhysicalDataMap	359
GroupWriteEnableMap	361
RedundancyAnalysis	363
RedundancyAnalysis/RowSegment	366
RedundancyAnalysis/ColumnSegment	369
PinMap	373
IclPorts	377
MemoryCluster	380
MemoryBistInterface	381
MemoryBistInterface/LogicalMemoryToInterfaceMapping	382
MemoryGroupAddressDecoding	384
PhysicalToLogicalMapping	386
PinMappings	387
FuseBoxInterface	390
Interface	393
Appendix B	
Configuration-Based Specification	397
MemoryOperationsSpecification	400
Algorithm	401
AddressGenerator	406
AddressRegisterA AddressRegisterB	412
DataGenerator	420
MicroProgram	423
Instruction/AdvancedOptions	425
Instruction/DataCommands	429
Instruction/AddressCommands	437
Instruction/CounterCommands	445
Instruction/NextConditions	447
OperationSet	466
SignalPipelineStages	468
Operation	470
AddressOverrides	472
Cycle	474
Cycle/AdvancedSignals	479
Cycle/UserSignals	486
Cycle/DramSignals	488
Cycle/ConcurrentPortSignals	491

Table of Contents

Appendix C	
MemoryBIST Algorithms	499
Notation Describing MemoryBIST Algorithms	500
MemoryBIST Algorithm Detected Faults	500
MemoryBIST Algorithm Test Times	502
Available Library Algorithms	504
SMarch Algorithm	504
ReadOnly Algorithm.....	506
SMarchCHKB Algorithm.....	507
SMarchCHKBci Algorithm	510
SMarchCHKBcil Algorithm	513
SMarchCHKBvcd Algorithm	517
LVMarchX Algorithm	529
LVMarchY Algorithm	534
LVMarchCMinus Algorithm	539
LVMarchLA Algorithm	544
<i>LVRowBar Algorithm</i>	550
LVColumnBar Algorithm.....	554
LVGalPat Algorithm.....	558
LVGalColumn Algorithm.....	565
<i>LVGalRow Algorithm</i>	573
LVCheckerboard1X1 Algorithm	580
LVCheckerboard4X4 Algorithm	585
LVWalkingPat Algorithm.....	590
LVBitSurroundDisturb Algorithm	596
LVAddressInterconnect Algorithm	612
LVDataInterconnect Algorithm	617
Appendix D	
Memory Fault Types	629
Address Decoder Faults	630
Bit/Group/Global Write Enable Faults	630
Access Transistor Current Leakage Faults	630
Data Retention Faults.....	631
DataPath Shorts	631
Destructive Read Faults	632
Dynamic Coupling Faults	633
Idempotent Coupling Faults.....	634
Inversion Coupling Faults	635
Memory Select Faults	635
Multi-Port Interference Faults	636
Multiport Synchronous Bitline Coupling Faults	637
Neighborhood Pattern Sensitive Faults	637
Parametric Faults	637
Read Disturb Faults	638
Read Enable Faults	638
Single Port Bitline Coupling Faults	639
Stuck-At Faults	639

Stuck-Open Faults	640
Transition Faults	640
Write Recovery Faults	641
Appendix E	
Parallel Static Retention Testing	643
Parallel Static Retention Testing Limitations	643
Handling Multiple Controllers	645
Sequence for Test Sub-Phases	645
Sample PatternsSpecification Syntax for Test Sub-Phases	646
Handling Memories Tested Sequentially	647
Testing Controllers in Groups	648
Sequence for Test Sub-Phases	648
Sample PatternsSpecification Syntax for Test Sub-Phases	649
Appendix F	
Memory BIST Physical Mapping Examples	651
Memory Cores	652
Example 1	652
Logical and Physical Mapping	655
Example 2	657
Example 3	658
Example 4	660
Example 5	662
Appendix G	
Complete Examples for Multi-Segment Memory Repair	667
Parallel BISR Interface Example	667
Serial BISR Interface Example	673
Appendix H	
Functional Debug Memory Access	679
Feature Description	680
Algorithm Description	680
Verification of Read and Write Access Functions	684
Requirements, Assumptions and Limitations	685
Implementation	686
Appendix I	
BAP Parallel Memory Access	691
Feature Description	692
BAP Architecture	692
BAP Direct Access Interface Pins	694
System Logic Interaction and Timing	698
Timing Diagram	698
Clocking Schemes	698
Sampling Pass/Done Signals	699

Table of Contents

Requirements, Assumptions and Limitations	699
Implementation	701
Inserting the BAP Direct Access Interface	702
BAP and Controller DftSpecification Options	702
Clock Connections	704
System Connections	705
Verifying the BAP Direct Access Interface	707
PatternsSpecification Options	707
Selection Codes	710
Timing Closure Considerations	712
Appendix J	
Getting Help	713
The Tesson Documentation System	713
Mentor Support Services	714
Third-Party Information	
End-User License Agreement	

Table of Contents

List of Figures

Figure 2-1. Design Flow for Tessent Shell MemoryBIST.....	27
Figure 2-2. Design Loading.....	30
Figure 2-3. Specify and Verify DFT Requirements	36
Figure 2-4. Create DFT Specification	42
Figure 2-5. GUI to Edit the DFT Specification.....	44
Figure 2-6. Process DFT Specification	45
Figure 2-7. Extract ICL	46
Figure 2-8. Create Patterns Specification	48
Figure 2-9. GUI to Edit the Patterns Specification	50
Figure 2-10. Process Patterns Specification	51
Figure 2-11. Run and Check Test Bench Simulations.....	52
Figure 2-12. Test Logic Synthesis	54
Figure 3-1. DFTVisualizer GUI Invocation	85
Figure 3-2. DftSpecification Hierarchy and Values	86
Figure 3-3. Creating a New Test Step for Controller(c1)	87
Figure 3-4. MemoryInterface(m3) Drag-and-Drop Result	88
Figure 3-5. DftSpecification Validation Error in DFTVisualizer	89
Figure 4-1. Verilog Simulation Log Files for Single Memory Certification.....	105
Figure 4-2. Partial Output of a Verilog Simulation Log	106
Figure 4-3. Example RedundancyAnalysis Wrapper (Partial)	107
Figure 4-4. Pattern Generation.....	114
Figure 5-1. Chip-Level Configuration with BISR Controller	124
Figure 5-2. Memories With Serial and Parallel Self-Repair Interfaces.....	127
Figure 5-3. Example Memory with Redundant IO	131
Figure 5-4. Example Memory with Redundant Column	132
Figure 5-5. Repair Analysis Support in Memory Library File	134
Figure 5-6. Example of Memory with IO Redundancy	136
Figure 5-7. RedundancyAnalysis Wrapper Syntax for Example 1	137
Figure 5-8. FuseSet Registers for Example 1	138
Figure 5-9. Example of Memory with Column/IO Redundancy	139
Figure 5-10. RedundancyAnalysis Wrapper Syntax for Example 2	141
Figure 5-11. FuseSet Registers for Example 2	142
Figure 5-12. Example Memory with Redundant Rows.....	148
Figure 5-13. Repair Analysis/Row Support in Memory Library File	150
Figure 5-14. Memory Library File Sample Syntax	152
Figure 5-15. Repair Analysis Support in Memory Library File	160
Figure 5-16. Example Memory with Row and IO/Column Redundancy	163
Figure 5-17. Memory Library File Sample Syntax	164
Figure 5-18. Repair Support in Memory Library File	172
Figure 5-19. Example Memory with Two Spare Rows Using Parallel BISR interface	176

Figure 5-20. Memory Library File Fragment for Two Spare Rows Using Parallel BISR Interface	177
Figure 5-21. Example Memory With One Spare Column and Serial BISR Interface	178
Figure 5-22. A Fragment of the Memory Library File Sample Syntax	179
Figure 5-23. Example MBISR TCD File Reporting BISR Statistics	187
Figure 5-24. Example PatternsSpecification to Verify BIRA and BISR	200
Figure 5-25. Example PatternsSpecification Configuration for FuseBox Access Mode . . .	204
Figure 5-26. PatternsSpecification Configuration for Autonomous Modes	205
Figure 5-27. PatternsSpecification Configuration for Autonomous Modes for MemoryBIST- Only Verification	207
Figure 5-28. BISR Controller Protocol in Functional Mode (Single Power Domain Group)	208
Figure 5-29. BISR Controller Protocol in Functional Mode (Multiple Power Domain Groups)	210
Figure 5-30. BISR Controlled from Functional Inputs	214
Figure 5-31. Post-Repair MemoryBIST with BISR Controlled from Functional Inputs . . .	215
Figure 5-32. iProc Usage for Physical Block Level with No Fuse Box Controller	216
Figure 5-33. iProc Usage for Chip Level with Fuse Box Controller Present	217
Figure 5-34. BISR Manufacturing Flow Example	218
Figure 5-35. Initialization Test Pattern Configuration	219
Figure 5-36. Pre-Repair Test Patterns Example	221
Figure 5-37. Example Configuration for TP2_1 Repair Test Patterns	222
Figure 5-38. Example Configuration for TP2_2 Repair Test Patterns	223
Figure 5-39. Example Configuration for Post-Repair Test Patterns	225
Figure 5-40. Flow Variation: Testing Repair Solution Before Fuse Programming	227
Figure 5-41. Flow Variation: Performing Fuse Programming During Any Test Insertion .	228
Figure 5-42. Fuse Box Organization	230
Figure 5-43. Fuse Box Organization (<code>max_fuse_box_programming_sessions > 1</code>)	231
Figure 5-44. Encoding of Repair Information Using CompressBisrChain Script	232
Figure 5-45. Manufacturing Flow Using CompressBisrChain Script (Incremental Repair) .	234
Figure 5-46. Soft Incremental Repair Flow	236
Figure 5-47. BISR-to-BIRA Transfer Connection	237
Figure 5-48. Detection of Repair Needed Condition	239
Figure 5-49. Hard Incremental Repair Manufacturing Flow Example	247
Figure 5-50. Incompatible Address Mapping Example	250
Figure 5-51. Incompatible Segment Size Example	251
Figure 5-52. DefaultsSpecification Repair Sharing Properties	253
Figure 5-53. Repair Sharing on All Memory Instances Example	253
Figure 5-54. Example DftSpecification Properties for Repair Sharing	255
Figure 5-55. BISR Chain Ordering with Repair Sharing	256
Figure 5-56. BISR Register Location with Local Comparators	257
Figure 5-57. BISR Register Location Example with Two Repair Groups	258
Figure 5-58. BISR Register Location Example with One Repair Group	258
Figure 5-59. Overall Fast BISR Loading Architecture	260
Figure 5-60. Parallel Load Multiplexers	261
Figure 6-1. Example of CHIP with Memory Cluster Module	271

List of Figures

Figure 6-2. Composition of Logical Memory LM_1	272
Figure 6-3. Memory Cluster Module After Embedded Test Insertion	273
Figure 6-4. Memory BIST Shared Bus Hardware Overview	274
Figure 6-5. Memory Cluster Core Library File Syntax	277
Figure 6-6. Logical Memory Core Library File Syntax	278
Figure 6-7. Design Overview of a Memory Cluster Module with BISR	290
Figure 8-1. Scan-based Diagnosis Approach Topology Using Enhanced Stop-On-Nth-Error Serial Scan	311
Figure 9-1. Hierarchical Design Example with Many Memories	318
Figure 9-2. Hierarchical Design Example with MemoryBIST Controllers	319
Figure 9-3. Steps to Implement MemoryBIST for Example Design in One Tool Invocation	321
Figure A-1. Memory with 2 Stages of Built-In Pipelining on the Output Data	340
Figure C-1. LVMarchX Example Algorithm Wrapper	532
Figure C-2. LVMarchY Example Algorithm Wrapper	537
Figure C-3. LVMarchCMinus Example Algorithm Wrapper	542
Figure C-4. LVMarchLA Example Algorithm Wrapper	548
Figure C-5. LVRowBar Example Algorithm Wrapper	553
Figure C-6. LVColumnBar Example Algorithm Wrapper	557
Figure C-7. LVGalPat Example Algorithm Wrapper	563
Figure C-8. LVGalColumn Example Algorithm Wrapper	570
Figure C-9. LVGalRow Example Algorithm Wrapper	578
Figure C-10. LVCheckerboard1X1 Example Algorithm Wrapper	583
Figure C-11. LVCheckerboard4X4 Example Algorithm Wrapper	588
Figure C-12. LVWalkingPat Example Algorithm Wrapper	594
Figure C-13. LVBitSurroundDisturb Example Algorithm Wrapper	609
Figure C-14. LVAddressInterconnect Example Algorithm Wrapper	615
Figure C-15. LVDataInterconnect Example Algorithm Wrapper	622
Figure E-1. Parallel Static Retention Test Sequence	646
Figure E-2. Parallel Static Retention Test with Controller Groups	649
Figure F-1. Simple Architecture for the Memory in Example 1	653
Figure F-2. Logical Memory Cell Arrangement	655
Figure F-3. Row Address Mapping Example	657
Figure F-4. Column Address Mapping Example	659
Figure F-5. Data Mapping Example	661
Figure F-6. Data Mapping Example that Incorporates AND	664
Figure G-1. Example Memory with Parallel BISR Interface	668
Figure G-2. Memory Library File Sample Syntax for Parallel BISR Interface	669
Figure G-3. Example Memory with Serial BISR Interface	674
Figure G-4. Memory Library File Sample Syntax for Serial BISR Interface	674
Figure H-1. Example Read Algorithm for a Single Location	681
Figure H-2. Example Write Algorithm for a Block of Locations	683
Figure H-3. Test Steps Used for Verification of Debug Capability	685
Figure H-4. Functional Debug Mode Write Example	688
Figure H-5. Functional Debug Mode Read Example	689
Figure I-1. BAP Parallel Memory Access Diagram	693

Figure I-2. BAP Parallel Memory Access Timing Protocol 698

List of Tables

Table 3-1. Parameter Impact on Performance and Area	71
Table 5-1. BIRA Module Location	125
Table 5-2. Fuse Box for Mapping Defective IO/Columns	139
Table 5-3. <MEMORY_INSTANCE_NAME>_REPAIR_STATUS Decodes	143
Table 5-4. <MEMORY_INSTANCE_NAME>_REPAIR_STATUS Decodes	154
Table 5-5. <MemoryInstance>_STATUS_REG Decodes	167
Table 5-6. REPAIR_STATUS Register Decodes	239
Table 5-7. GO Output Interpretation by Repair Stage	240
Table 5-8. Action Table for GO_ID	240
Table 5-9. Action Descriptions	241
Table 5-10. GO_ID Behavior in BIRA Mode	243
Table 5-11. GO_ID Behavior in Stop-On-Error Mode	243
Table 5-12. GO_ID Behavior in Go/NoGo Mode	243
Table 5-13. Legend	243
Table 5-14. Fast BISR Loading Support	262
Table 5-15. BISR Loading Terminology	262
Table A-1. Conventions for Command Line Syntax	323
Table A-2. Syntax Conventions for Configuration Files	323
Table A-3. Valid Function Values	343
Table A-4. Function Value Details and Guidelines	347
Table A-5. Operator Precedence	356
Table B-1. Configuration Data Editing and Introspection Commands	397
Table B-2. Syntax Conventions for Configuration Files	398
Table B-3. Un-Nested address_sequence Property Modification	454
Table B-4. Nested address_sequence Property Modification	454
Table B-5. Un-Nested expect_data_sequence Property Modification	456
Table B-6. Nested expect_data_sequence Property Modification	457
Table B-7. Un-Nested write_data_sequence Property Modification	461
Table B-8. Nested write_data_sequence Property Modification	461
Table B-9. Required Settings for Increment/Decrement by 1	480
Table B-10. Required Settings for Increment/Decrement by > 1	481
Table B-11. Required Settings for Increment/Decrement by 1	483
Table B-12. Required Settings for Increment/Decrement by > 1	484
Table B-13. write_enable Write Access Operation When Explicitly Specified	493
Table B-14. write_enable Write Access Operation Default Settings	494
Table C-1. MemoryBIST Algorithm Fault Detection	501
Table C-2. MemoryBist Algorithm Test Times	502
Table C-3. MemoryBist Algorithm PSRT Test Times	503
Table C-4. Description of SMarch Test Algorithm per Test Port	505
Table C-5. Description of ReadOnly Test Algorithm per Test Port	506

Table C-6. Description of SMarchCHKB Test Algorithm per Test Port	507
Table C-7. Description of SMarchCHKBci Test Algorithm per Test Port	510
Table C-8. Description of SMarchCHKBcil Test Algorithm per Test Port	513
Table C-9. Description of SMarchCHKBvcd Test Algorithm per Test Port	518
Table C-10. Mapping of Operation Code to Operation Name	527
Table C-11. Description of LVMarchX Algorithm	530
Table C-12. Description of LVMarchY Algorithm	535
Table C-13. Description of LVMarchCMinus Algorithm	540
Table C-14. Description of LVMarchLA Algorithm	545
Table C-15. Description of LVRowBar Algorithm	551
Table C-16. Description of LVColumnBar Algorithm	556
Table C-17. Description of LVGalPat Algorithm	560
Table C-18. Description of LVGalColumn Algorithm	567
Table C-19. Description of LVGalRow Algorithm	574
Table C-20. Description of LVCheckerboard1X1 Algorithm	582
Table C-21. Description of LVCheckerboard4X4 Algorithm	586
Table C-22. Description of LVWalkingPat Algorithm	591
Table C-23. Description of LVBitSurroundDisturb Algorithm	597
Table C-24. Description of LVAddressInterconnect Algorithm	613
Table C-25. LVAddressInterconnect Algorithm Fault Coverage	616
Table C-26. Description of LVDataInterconnect Algorithm	618
Table C-27. LVDataInterconnect Algorithm Fault Coverage	627
Table I-1. BAP Direct Access Interface Ports	694
Table I-2. Pattern Generation Options	708

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?	21
Tessent MemoryBIST Capabilities	22
Tessent MemoryBIST Library Algorithms	22
User-Defined Algorithms	23
Tessent MemoryBIST Library Operation Sets	23
Custom Operation Sets	24

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.
- Define one or more custom test memory algorithms that will be hard coded into the memory BIST controller.
- Choose memory test algorithms from Mentor Graphics' library of 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
- LVAddressInterconnect
- LVDataInterconnect

For details on each algorithm, refer to the “[MemoryBIST Algorithms](#)” section.

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 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.

- Sync
- SyncWR
- SyncWRvcld

- Async
- AsyncWR
- ROM

For details on each operation set, refer to the [OperationSet](#) property, which is set within the [Core/Memory](#) wrapper in the Tessent Core Description (TCD) file.

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.

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, which 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	27
Prerequisites	28
Design Flow Dofile Example	28
Design Loading.....	30
Set the Context	30
Read the Libraries.....	31
Read the Design	32
Elaborate the Design.....	33
Report the Design Data.....	33
Specify and Verify DFT Requirements	36
Set DFT Specification Requirements	36
Add Properties and Constraints	37
Define Clocks	39
Run DRC.....	40
Create DFT Specification	42
Invoke create_dft_specification	42
Edit/Configure the DFT Specification According to Your Requirements	43
Validate the DFT Specification	44
Process DFT Specification.....	45
Create DFT Hardware with the DFT Specification	45
Extract ICL	46
Preparation for Pattern Generation	46
Create Patterns Specification	48
Automatically Created Patterns Specification	48
Edit/Configure the Patterns Specification According to Your Requirements	49
Process Patterns Specification	51

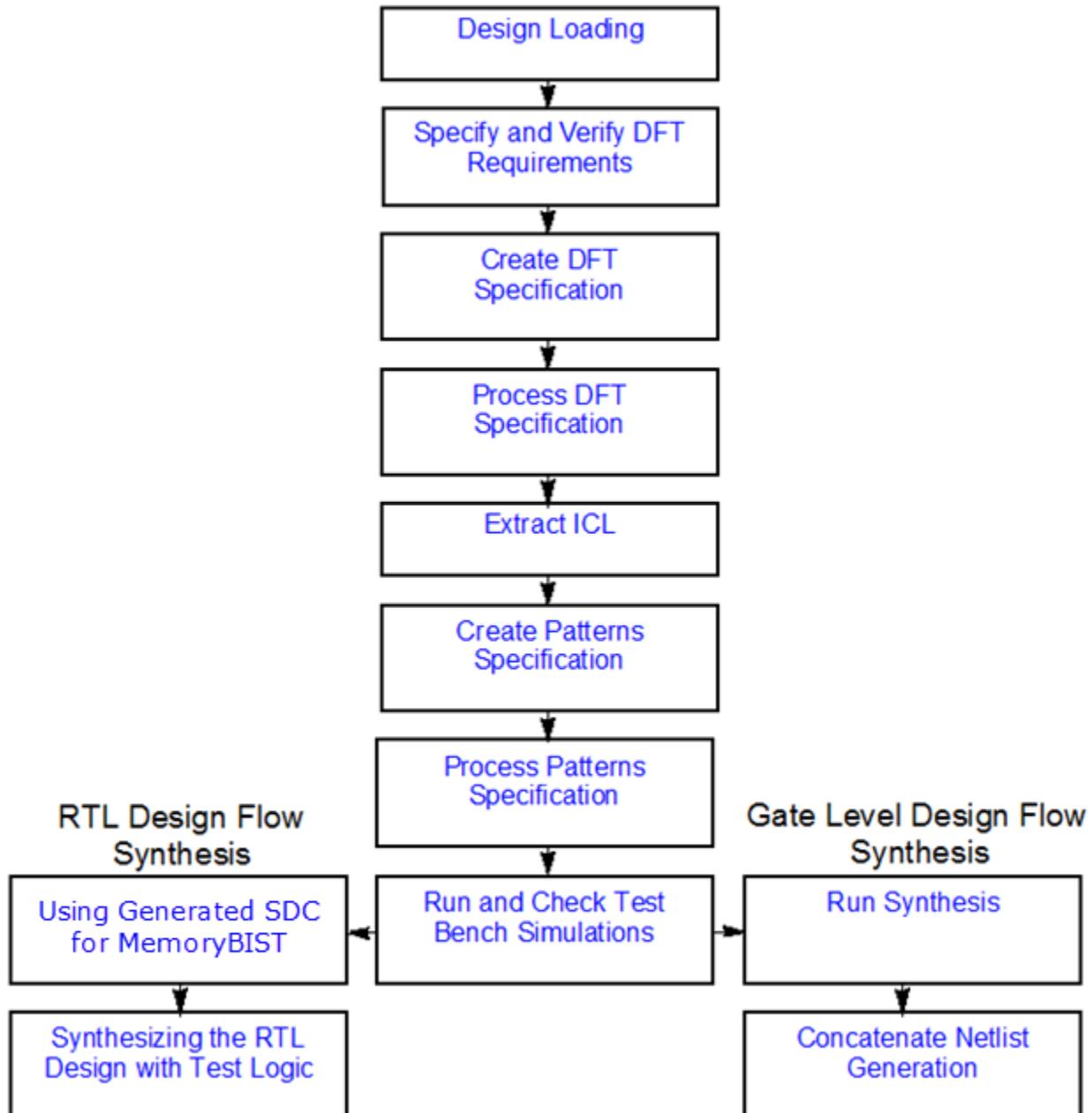
Process Patterns According to the Patterns Specification	51
Run and Check Test Bench Simulations	52
Run Simulations	52
Check Results	53
Formal Verification.....	53
Test Logic Synthesis.....	54
RTL Design Flow Synthesis.....	55
Gate Level Design Flow Synthesis.....	57
Run and Check Test Bench Simulations with a Gate Level Netlist	57

DFT Flow Using Tesson Shell

Tesson MemoryBIST in Tesson 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 Tesson Shell MemoryBIST



Prerequisites..... 28

Design Flow Dofile Example	28
----------------------------------	----

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 Test Bench Simulations

`run_testbench_simulations`
`check_testbench_simulations`

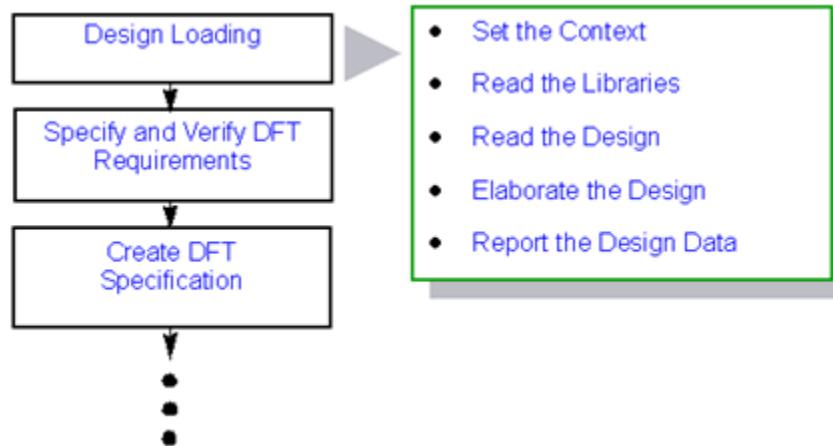
Test Logic Synthesis

`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, and elaborating the design.

Figure 2-2. Design Loading



Set the Context	30
Read the Libraries	31
Read the Design	32
Elaborate the Design	33
Report the Design Data	33

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 will be 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. When inserting memoryBIST into an RTL-only design, reading the memory BIST library is sufficient but is not typical because some library cells are instantiated in an RTL design for pad cells and clock control. 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 to be 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.

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.

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

Example 1

The following example shows how to read in one netlist, which 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 will be used. The -v and -y options accept a list of paths, if needed.
-

Example 3

The following example shows how to read in DEF (Design Exchange Format) file containing the placement information of the memories and a UPF (Unified Power Format) file describing the power domain associated with the memories.

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

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 will act 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 fanin 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
```

Report the Design Data

You can use several commands to check the loaded design and associated libraries.

Examples of the more commonly used reporting commands are as follows.

Example 1

The following example executes set_design_sources and then reports the designs sources that will be 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 is displayed,

```
> 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

 If core descriptions other than memories are loaded, they will also be displayed in this report.

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_pipelineing : 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_pipelineing : off
// physical_cluster_override :
// power_domain_island :
// test_clock_override :
// use_in_memory_bist_dft_specification : auto
// use_in_memory_bisr_dft_specification : auto
```

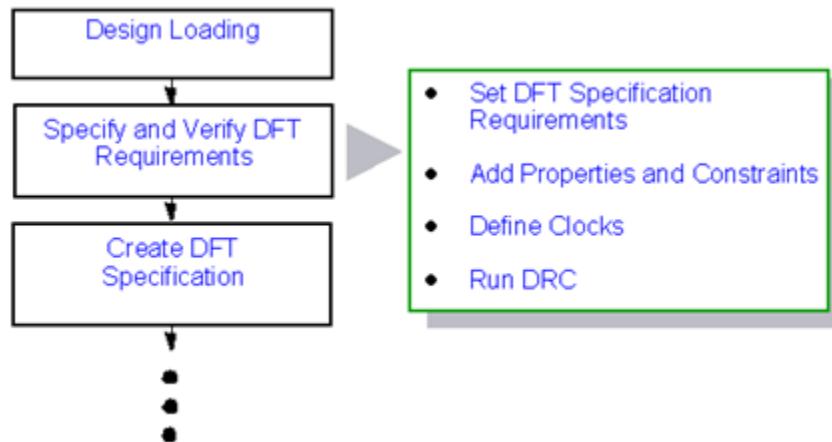
Note

 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 UPF file to specify power domain islands or can be changed explicitly by using the [set_memory_instance_options](#) command.

Specify and Verify DFT Requirements

The next step to insert memoryBIST in Tessent Shell is to specify the DFT requirements, add constraints, and verify whether the DFT requirements specified are correct by running DRC (Design Rule Checking).

Figure 2-3. Specify and Verify DFT Requirements



Set DFT Specification Requirements	36
Add Properties and Constraints	37
Define Clocks	39
Run DRC	40

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 will affect DFT specification creation.

Example 1

To insert memoryBIST 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 .tesson_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*.

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 will be 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 will drive 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, Tesson 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 will 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`, which 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 Tesson Shell invocation or a subsequent invocation to reuse the modified `<design_name>.bisr_segment_order` file, you must specify the `set_memory_instance_options -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 executed.

```
SETUP> set_memory_instance_options -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 executed. 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

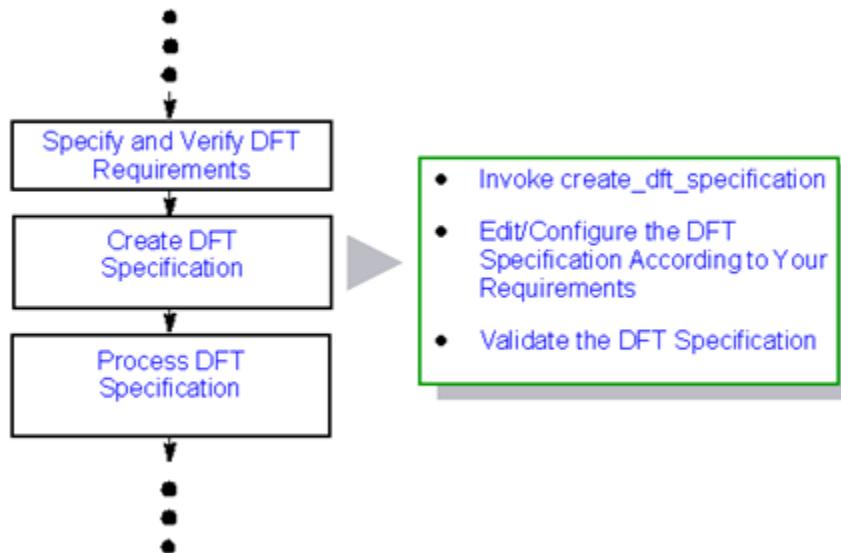
-  The file name of the `bisr_segment_order_file` is arbitrary and does not need to be the default name.
-

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	42
Edit/Configure the DFT Specification According to Your Requirements	43
Validate the DFT Specification.	44

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 re-generate 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%.

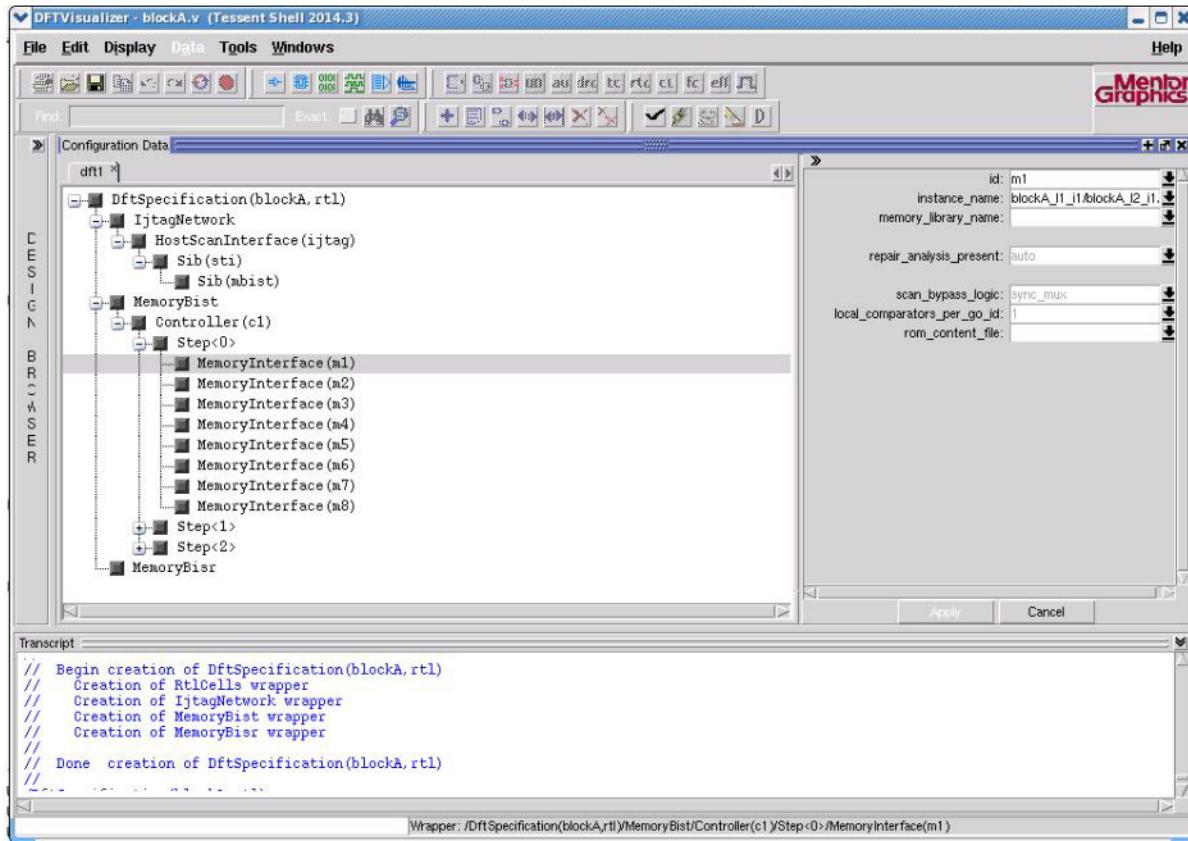
```
set_system_mode setup
set_memory_instance_options -physical_cluster_size_ratio 40
check_design_rules
create_dft_specification -replace
```

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

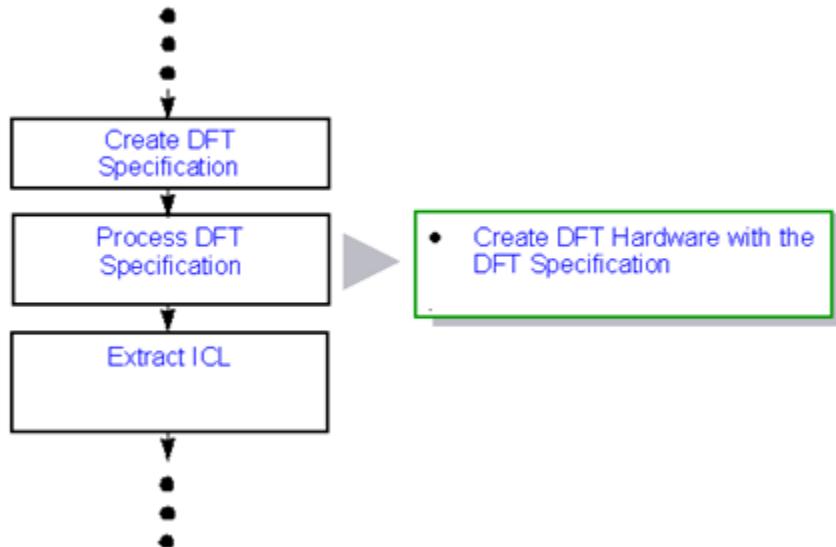
The following example shows how to validate your DFT specification to check for issues or mistakes before processing it.

```
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 45

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

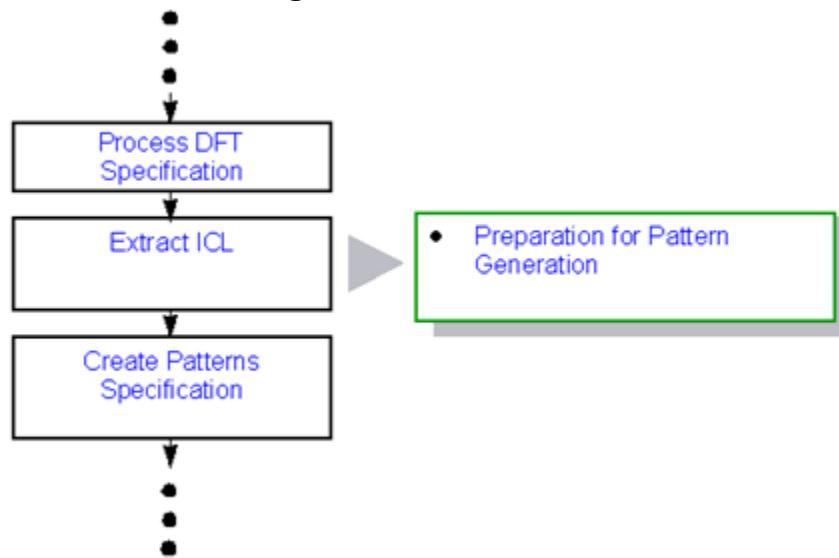
The Extract ICL step verifies the proper connectivity of the ICL modules that were inserted with the process_dft_specification command and, when no DRC violations are detected, extracts the top-level ICL description.

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 46

Preparation for Pattern Generation

The [extract_icl](#) command prepares the current design for pattern generation by finding all modules (both Tessent instruments and non-Mentor Graphics 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 “Using DFTVisualizer” section in the Tessent IJTAG User’s Manual.

Example

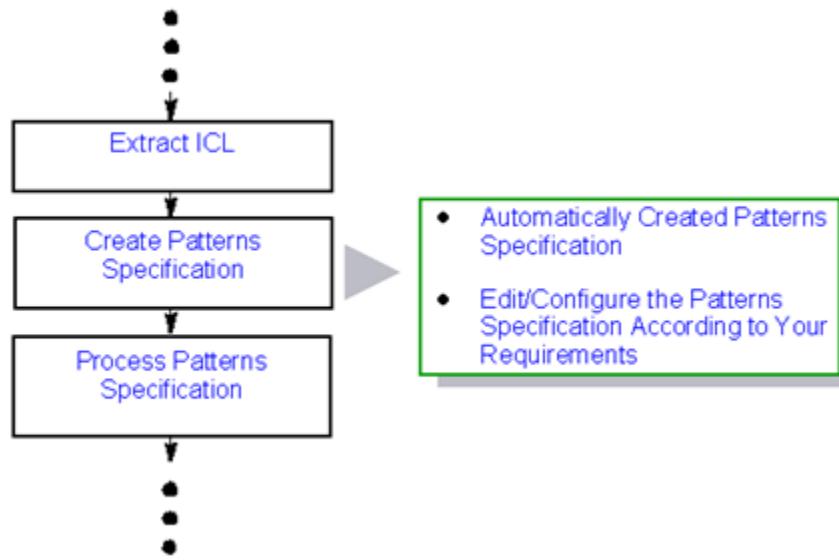
The following shows how to extract all ICL modules to the root of the design. The example shows the transcript when performing ICL extraction at the current_design level.

```
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 will be created using process_patterns_specification. You can edit or configure the default patterns specification to generate the desired patterns specification.

Figure 2-8. Create Patterns Specification



Automatically Created Patterns Specification	48
Edit/Configure the Patterns Specification According to Your Requirements.....	49

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 master Tcl script or dofile runs, the patterns specification that is written out will overwrite 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 desired edits 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.

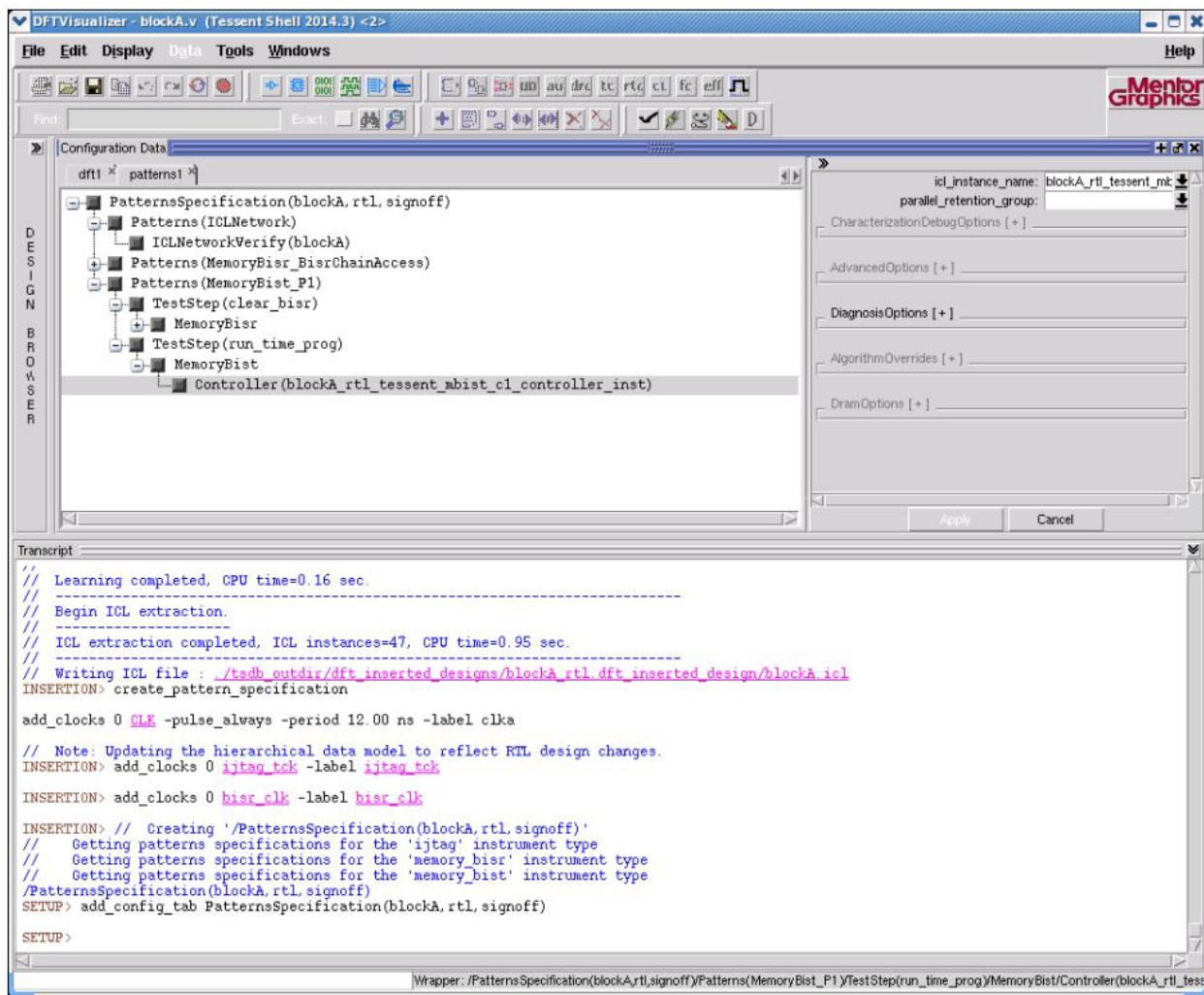
Example

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.

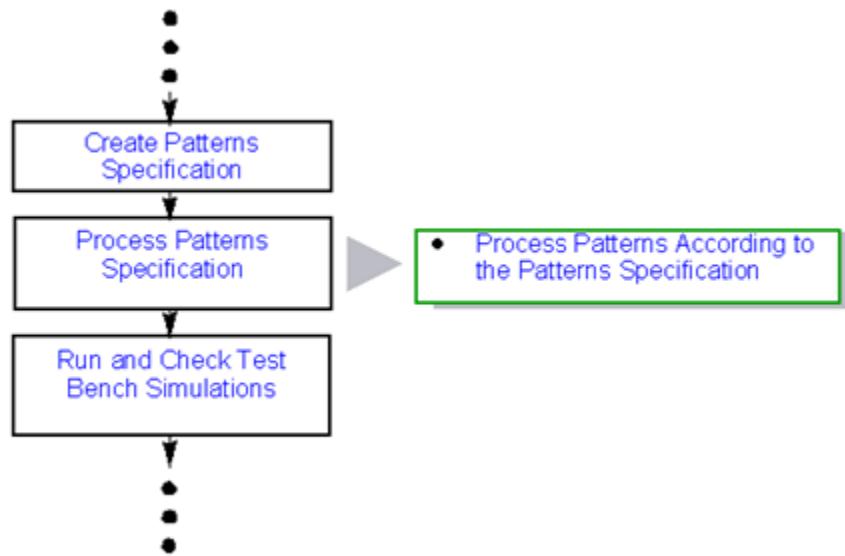
Figure 2-9. GUI to Edit the Patterns Specification



Process Patterns Specification

The Process Patterns Specification step creates the patterns and test benches.

Figure 2-10. Process Patterns Specification



Process Patterns According to the Patterns Specification 51

Process Patterns According to the Patterns Specification

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

Example

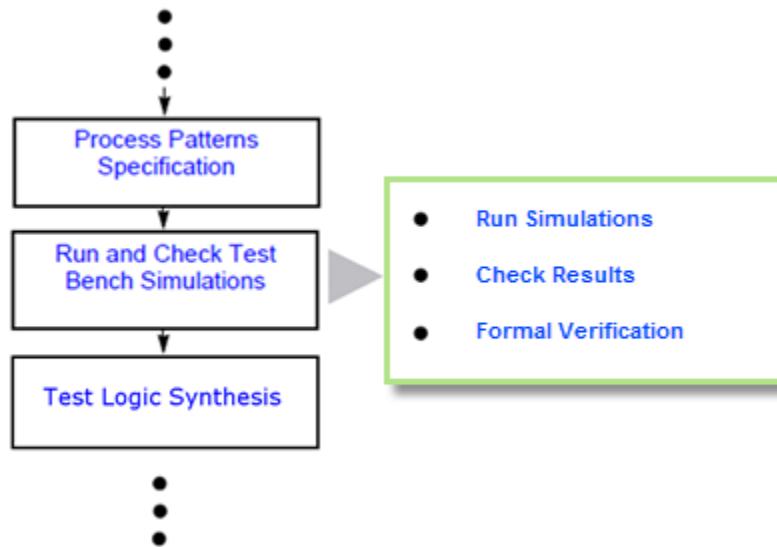
The following example generates the test benches.

```
process_patterns_specification
```

Run and Check Test Bench Simulations

The Run and Check Test Bench Simulations step is the last step in memoryBIST insertion using Tesson Shell. Here you run simulations of the memory BIST verification and then check the results.

Figure 2-11. Run and Check Test Bench Simulations



Run Simulations.....	52
Check Results.....	53
Formal Verification	53

Run Simulations

Use the `run_testbench_simulations` command to invoke a simulation manager to run a set of simulation test benches.

The `run_testbench_simulations` command compiles and simulates the test benches 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 *Tesson 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

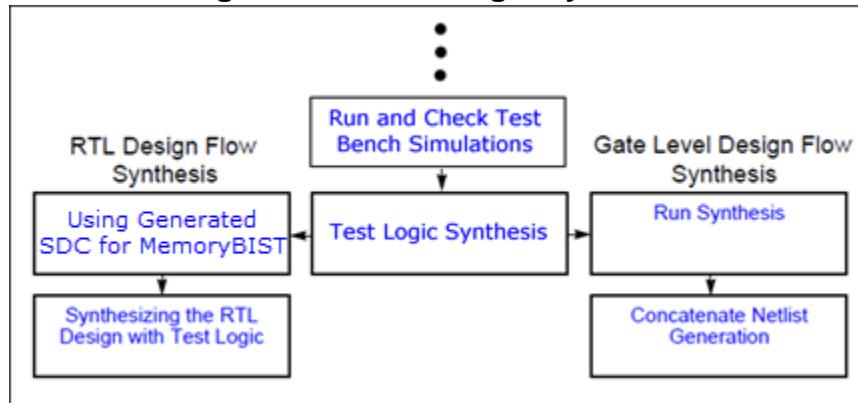
Tessent Shell based products currently do not generate scripts for use with Synopsys Formality.

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 different when handling a RTL or gate level designs. The following sections outline the different options.

Figure 2-12. Test Logic Synthesis



RTL Design Flow Synthesis	55
Gate Level Design Flow Synthesis	57

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	55
Synthesizing the RTL Design with Test Logic	56

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). Since 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. The names and function of the procedures are outlined below:

- `tesson_set_default_variables`

This proc defines the default value of the variables used in instrument timing constraints. It is provided as a template. The user should override the values of these variables to correspond to their setup. For example, the values of the array "tesson_clock_mapping" should be overwritten with the user's chosen names for the functional clocks.

- `tesson_create_functional_clocks`

This proc defines the functional clocks used by the instrument constraints. This proc is not typically called since most users will define their functional clocks in their own timing scripts.

- `tesson_constraint_<design_name>_non_modal`

This proc defines the constraints for instruments other than memory BIST and can be merged with other instrument constraints as well as design level constraints. For memory BIST, this proc contains constraints for BISR.

- `tesson_constraint_<design_name>_mentor_memory_bist_non_modal`

These procs define the constraints for memory BIST and can be merged with other instrument constraints as well as design level constraints.

- `tesson_constraint_<design_name>_mentor_memory_bist_modal`

This proc defines the constraints that enable the embedded test mode of the memory BIST inserted logic for static timing analysis (STA) of the design. For example, it

sensitizes the BIST clock path, blocks all clocks other than the test clock to memories that have asynchronous ports and blocks TCK from propagating on the functional path. The controllers and interfaces are also placed in test mode.

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.

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	57
Concatenate Netlist Generation	57

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 which contains the synthesized test logic and modified design modules will automatically be created and placed in the `dft_inserted_designs` directory of the TSDB.

Run and Check Test Bench 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 test benches

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, i.e. 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.....	59
Specifying and Verifying MemoryBIST Requirements.....	61
Enabling Memory BIST & Repair	61
Loading Layout Placement Information.....	62
Loading Power Domain Information	62
Adding Clocks	63
Changing Default Settings	63
Adding Constraints Before Design Rule Checking	65
Creating, Modifying, and Validating a MemoryBIST DFT Specification	69
Adding Constraints Before Creating an Initial DftSpecification	69
Parameter Selection Impacts on Performance and Area.....	70
MemoryBIST Partitioning Rules	73
Creating the DftSpecification	74
Review and Basic Edits of the DftSpecification.....	75
Re-Creating the DftSpecification	79
Validating the DftSpecification	82
Additional Editing Options of the DftSpecification	84
Processing the DftSpecification	90

Design and Library Requirements

You must have loaded the design into which the memory BIST and repair solution shall be inserted. 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 will be 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;  
    .  
    .  
    .  
}
```

The Tessent Cell library is needed for a design flow where memory BIST and repair logic must be inserted into a gate-level design. Additionally, for a top level insertion the library must include pad cells, or a pad cell library must be loaded. For an RTL level flow, no library is needed until the test bench simulation step at the end of the flow, as described in “[Run and Check Test Bench 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 will 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 rule checks, which will enable you to create the memory BIST DftSpecification in the next section of the design flow. Tessent MemoryBIST will then utilize this DftSpecification to implement the desired memory BIST solution in your design.

Enabling Memory BIST & Repair	61
Loading Layout Placement Information	62
Loading Power Domain Information	62
Adding Clocks	63
Changing Default Settings	63
Adding Constraints Before Design Rule Checking	65

Enabling Memory BIST & Repair

After design loading and setting the current level, you first must tell Tessent Shell that you want to execute the 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 which Tessent MemoryBIST will set automatically. For example, Tessent MemoryBIST will automatically enable 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 will issue a warning or an error, depending on your particular flow case.

Tip

i 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 Tesson 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

Tesson MemoryBIST will take 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 Tesson MemoryBIST assigns the current memory to a new controller or to an existing one.

After design rule checks 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

Tesson MemoryBIST will take 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, and test time estimates to mention only a few. You provide the power domain information through a standard UPF or CPF file.

After design rule checks and during the creation of the initial DftSpecification, you have the option to observe the power domain assignment Tesson MemoryBIST has derived from the

UPF or CPF files, and override the assignment if desired. The “[Adding Constraints Before Creating an Initial DftSpecification](#)” section provides details and examples on how to perform these actions. The example below shows how to load a UPF file for your design.

```
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.
```

Adding Clocks

Using the `add_clocks` command, you tell the Tessent MemoryBIST tool which 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 will create one for you. You will find this label 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 you will find 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.

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 will be 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 will not be 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, you will learn how to make changes that impact only one or a collection of memories. Key for this is the `set_memory_instance_options` command.

Most changes using the `set_memory_instance_options` command will 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 Tesson MemoryBIST. Also done during design rule checking is the evaluating of the memory properties, like repair status.

Nonetheless, you may want to add certain constraints already in setup mode, before design rule checking. One important one 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. Tesson MemoryBIST will issue 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: 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 :

```

```
// 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_pipeline               : 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. Tesson 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 these analysis results before you create your initial DftSpecification.

At this part 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. Tesson MemoryBIST will then utilize this DftSpecification to implement the desired memory BIST solution 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, allowing 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	69
Parameter Selection Impacts on Performance and Area.....	70
MemoryBIST Partitioning Rules	73
Creating the DftSpecification	74
Review and Basic Edits of the DftSpecification	75
Re-Creating the DftSpecification	79
Validating the DftSpecification.	82
Additional Editing Options of the DftSpecification	84

Adding Constraints Before Creating an Initial DftSpecification

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

You can use the [set_memory_instance_options](#) command to change those constraints. As an example, this section shows you the steps you need to take to change the loaded power domain information.

The [check_design_rules](#) command analyzed the design and attached the power domain information to each memory instance. You may want to report or introspect this information in case your test setup uses other power domains than the functional mode, which you have loaded through the UPF or CPF file [earlier](#) in the design flow. You will observe that the loaded power information is not evaluated until after changing to analysis mode through the [check_design_rules](#) command.

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 will issue 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.

Parameter Selection Impacts on Performance and Area

Some parameters specified in the DftSpecification and DefaultsSpecification will 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	+		
	gate_bist_clock_in_functional_mode			
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
MemoryCluster	pipeline_cluster_inputs	+	-	8
	pipeline_cluster_outputs	+	-	8
	memory_access_level			
	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
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, which 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” will increase area and degrade the performance of the address counters.
8. Use this parameter if the memory cluster does not already have registered inputs and/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 due to a large number of memories and/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.

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 “CCR_x” and Step Compatibility Rules are denoted as “SCR_x” 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 will be testing memories of the same type exclusively.
- **CCR2:** All DRAMs on the same controller must have identical row, column and bank dimensions.
- **CCR3:** Memories are separated in groups such that the physical region, clock domain, and memory cluster are identical for all memories in a common group. When a multi-port memory is driven by multiple clock domains, the memory is associated to one of the clock domains with the fastest frequency. Refer to “[Shared Bus Interface MemoryBIST Implementation Flow](#)” for information on creating a memory cluster on a shared bus interface.

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.

- **SCR6:** All memories with more than one test port must have the same number of ports. However, single-port memories can be placed with any multi-port memory. Single test port memories have no restrictions.
- **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 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 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 will never be tested by the same controller. Likewise, non-identical DRAMs will never be 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

 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 will happen 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.

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 {
          tessonnt_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_11_i1/blockA_12_i1/mem1;
        }
        MemoryInterface(m2) {
          instance_name : blockA_11_i1/blockA_12_i1/mem2;
        }
        MemoryInterface(m3) {
          instance_name : blockA_11_i1/blockA_12_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_11_i1/blockA_12_i1/mem1;
        }
        MemoryInterface(m2) {
            instance_name : blockA_11_i1/blockA_12_i1/mem2;
        }
        MemoryInterface(m3) {
            instance_name : blockA_11_i1/blockA_12_i1/mem3;
        }
    }
    Step {
        MemoryInterface(m4) {
            instance_name : blockA_11_i1/blockA_12_i1/mem4;
        }
        MemoryInterface(m5) {
            instance_name : blockA_11_i1/blockA_12_i1/mem5;
        }
    }
    Step {
        MemoryInterface(m6) {
            instance_name : blockA_11_i1/blockA_12_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 : "";
        functional_debug_mode : off;
    [...]
```

Example 4: Introspecting and Changing DftSpecification Values

This example shows you the steps to edit a specific entry in the DftSpecification. Since you edit the information in-memory, a subsequent execution of the modified DftSpecification with the [process_dft_specification](#) command will consider 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;
    RepairOptions {
        fuse_set_extraction_sequence : address_before_fuse_map;
        row_bira_location : controller;
    }
    StopOnErrorOptions {
        failure_limit : 4096;
        failure_limit_auto_incrementation : on;
    }
}
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;
    RepairOptions {
        fuse_set_extraction_sequence : address_before_fuse_map;
        row_bira_location : controller;
    }
    StopOnErrorOptions {
        failure_limit : 2048;
        failure_limit_auto_incrementation : on;
    }
}
```

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_11_i1/blockA_12_i1/mem1, to a separate controller. The easiest way is to place this instance into its own physical cluster.

Example 1: 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, since 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_11_i1/blockA_12_i1/mem1;
        }
        MemoryInterface(m2) {
            instance_name : blockA_11_i1/blockA_12_i1/mem2;
        }
        MemoryInterface(m3) {
            instance_name : blockA_11_i1/blockA_12_i1/mem3;
        }
    }
    Step {
        MemoryInterface(m4) {
            instance_name : blockA_11_i1/blockA_12_i1/mem4;
        }
        MemoryInterface(m5) {
            instance_name : blockA_11_i1/blockA_12_i1/mem5;
        }
    }
    Step {
        MemoryInterface(m6) {
            instance_name : blockA_11_i1/blockA_12_i1/mem6;
        }
    }
}
ANALYSIS> report_memory_instances -limit 2
// -----
// Memory Instance: 'blockA_11_i1/blockA_12_i1/mem1'
// -----
// bist_data_in_pipelineing          : 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_11_i1/blockA_12_i1/mem2'
// -----
// bist_data_in_pipelineing          : 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_11_i1/blockA_12_i1/mem1 -
physical_cluster_override MyCluster
ANALYSIS> report_memory_instances -limit 2
// -----
// Memory Instance: 'blockA_11_i1/blockA_12_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_11_i1/blockA_12_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_11_i1/blockA_12_i1/mem1;
        }
        MemoryInterface(m2) {
            instance_name : blockA_11_i1/blockA_12_i1/mem2;
        }
        MemoryInterface(m3) {
            instance_name : blockA_11_i1/blockA_12_i1/mem3;
        }
    }
    Step {
        MemoryInterface(m4) {
            instance_name : blockA_11_i1/blockA_12_i1/mem4;
        }
        MemoryInterface(m5) {
            instance_name : blockA_11_i1/blockA_12_i1/mem5;
        }
    }
    Step {
        MemoryInterface(m6) {
            instance_name : blockA_11_i1/blockA_12_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_11_i1/blockA_12_i1/mem1;  
        }  
    }  
}  
ANALYSIS> report_config_data $spec/MemoryBist/Controller(c2)  
  
Controller(c2) {  
    clock_domain_label : clka;  
    Step {  
        MemoryInterface(m1) {  
            instance_name : blockA_11_i1/blockA_12_i1/mem2;  
        }  
        MemoryInterface(m2) {  
            instance_name : blockA_11_i1/blockA_12_i1/mem3;  
        }  
    }  
    Step {  
        MemoryInterface(m3) {  
            instance_name : blockA_11_i1/blockA_12_i1/mem4;  
        }  
        MemoryInterface(m4) {  
            instance_name : blockA_11_i1/blockA_12_i1/mem5;  
        }  
    }  
    Step {  
        MemoryInterface(m5) {  
            instance_name : blockA_11_i1/blockA_12_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. Since 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 will 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 Tesson MemoryBIST.

Invoking the GUI and Editing the DftSpecification	84
Validating a DftSpecification in the GUI.....	88
Editing the DftSpecification Using Shell Commands.....	89

Invoking the GUI and Editing the DftSpecification

The GUI 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 GUI, expand the hierarchy of the DftSpecification, add a new specification wrapper (a test step in this case) and use the drag-and-drop feature of the GUI to move testing a memory instance from its original step to the newly created step.

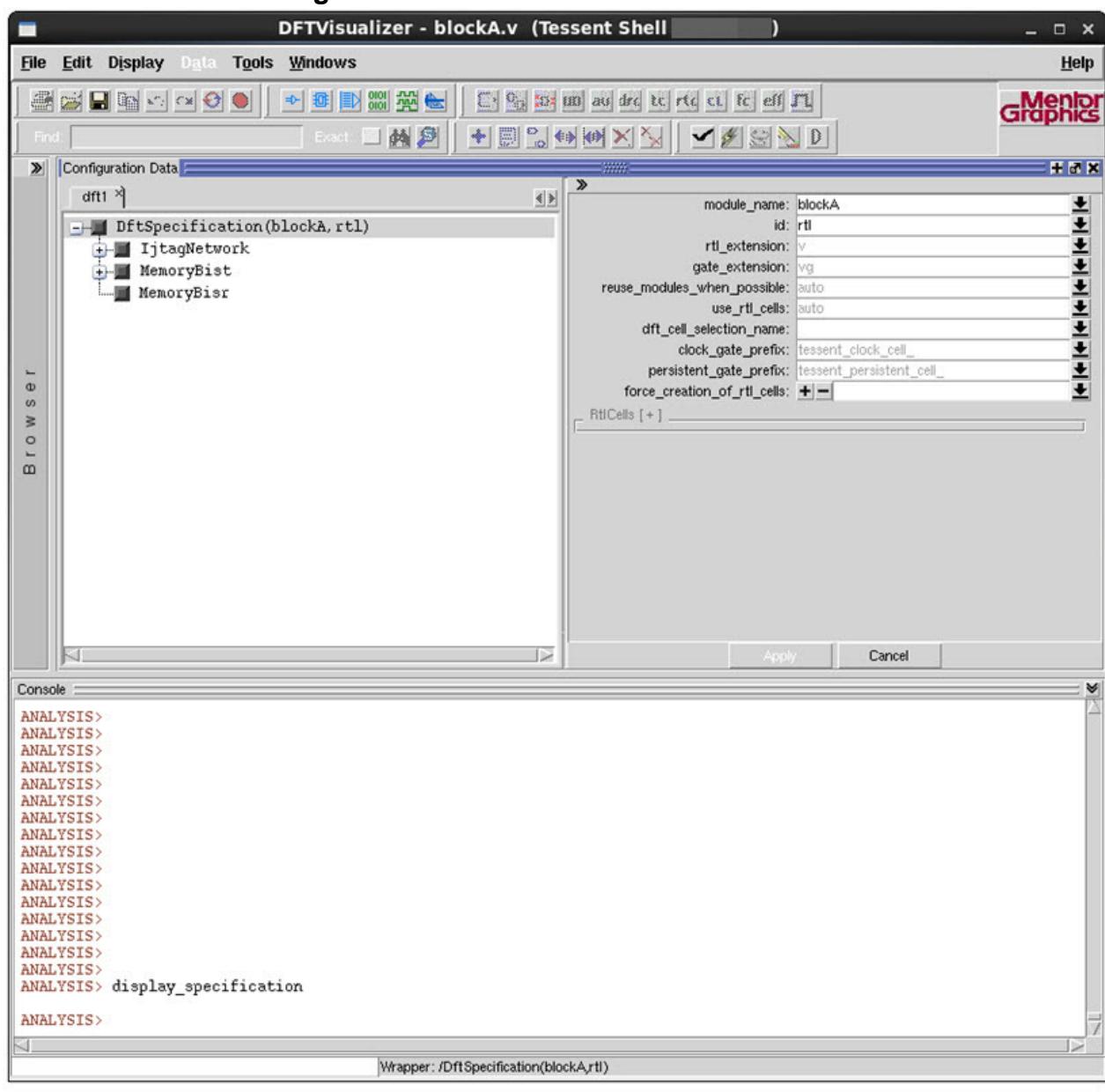
These examples show you only some of the capabilities of the GUI of the DftSpecification. There is much more to explore.

Example 1: Invoking the GUI for a DftSpecification

After you executed the [display_specification](#) command, the GUI starts up. It remembers the windows and DftSpecification expansions from your last session. The entries displayed in the GUI for your design might differ from the ones shown in [Figure 3-1](#), but the general objects should be there.

On the left, you see the (initially) collapsed hierarchy of the DftSpecification. On the right, you see data entry options, some collapsed, like the RtlCells wrapper. The fields are auto-populated based on the DftSpecification and its respective default value setting. The bottom part of the GUI windows displays the current Tesson Shell command history and prompt. You can type in commands here as well.

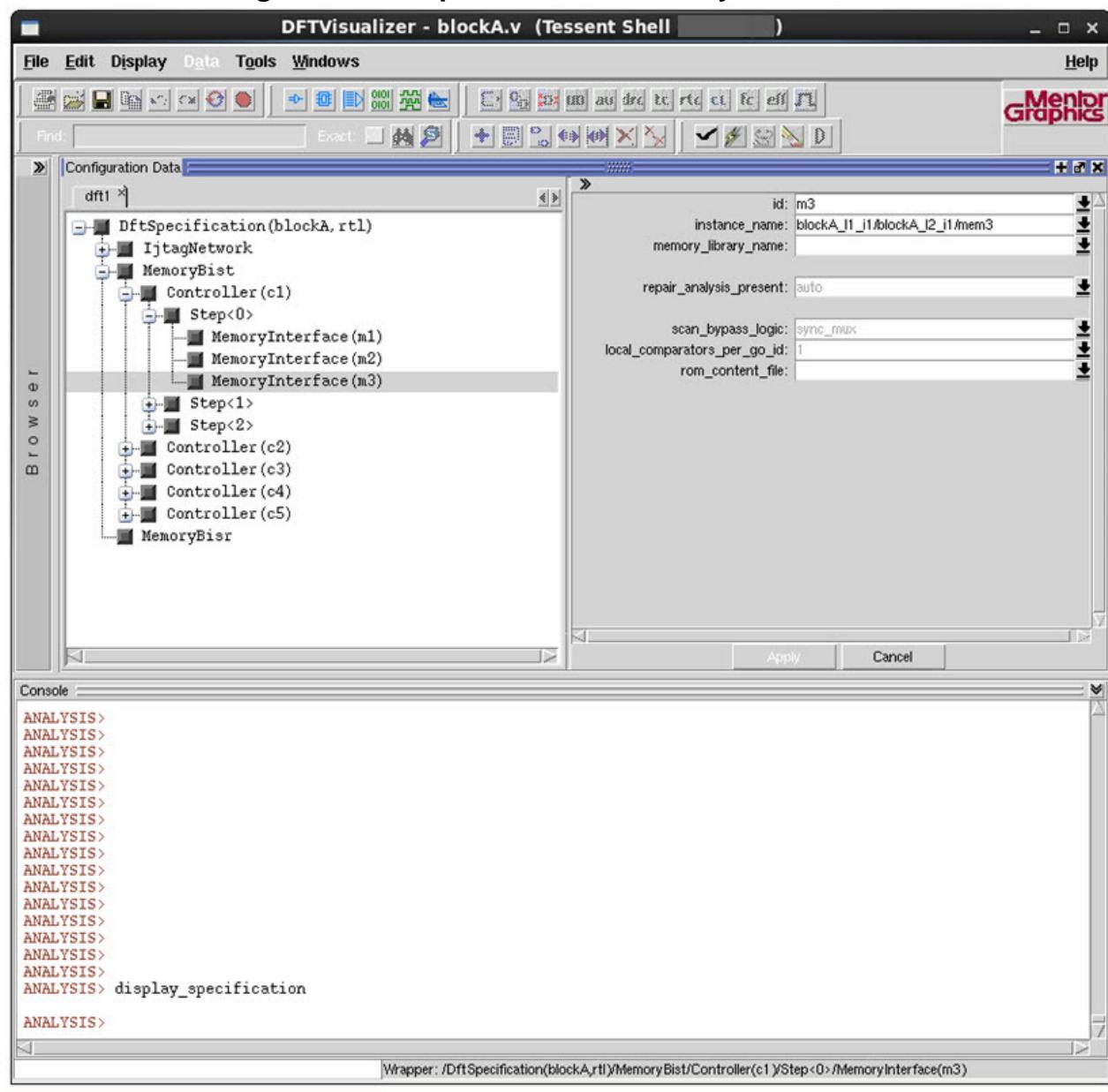
Figure 3-1. DFTVisualizer GUI Invocation



Example 2: Navigating through the Hierarchy of the DftSpecification

You navigate through the hierarchy of the DftSpecification by clicking on the small + symbols next to the specification wrappers. This will expand the wrapper, showing you additional wrappers until you finally come to a leaf entry. [Figure 3-2](#) shows you the expansion down to the memory interface m3 of step 0 of memory bist controller (c1).

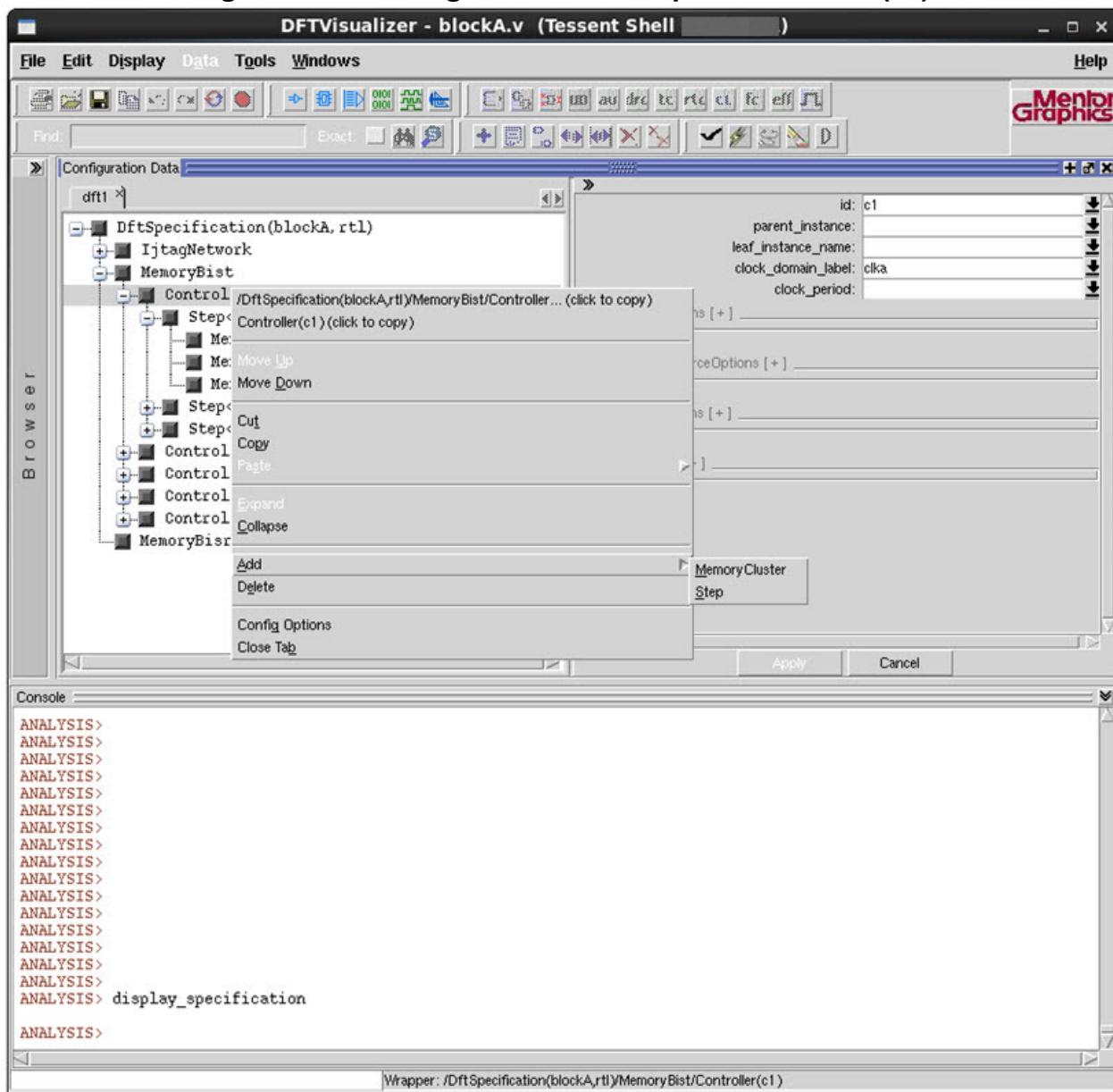
Figure 3-2. DftSpecification Hierarchy and Values



Example 3: Adding a new Specification Wrapper

For controller c1, you want to add another test step. As shown in Figure 3-3, you right-click on the Controller(c1) wrapper, a menu appears. Select there ‘Add’ and then ‘Step’. You will 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 on the right-hand window.

Figure 3-3. Creating a New Test Step for Controller(c1)

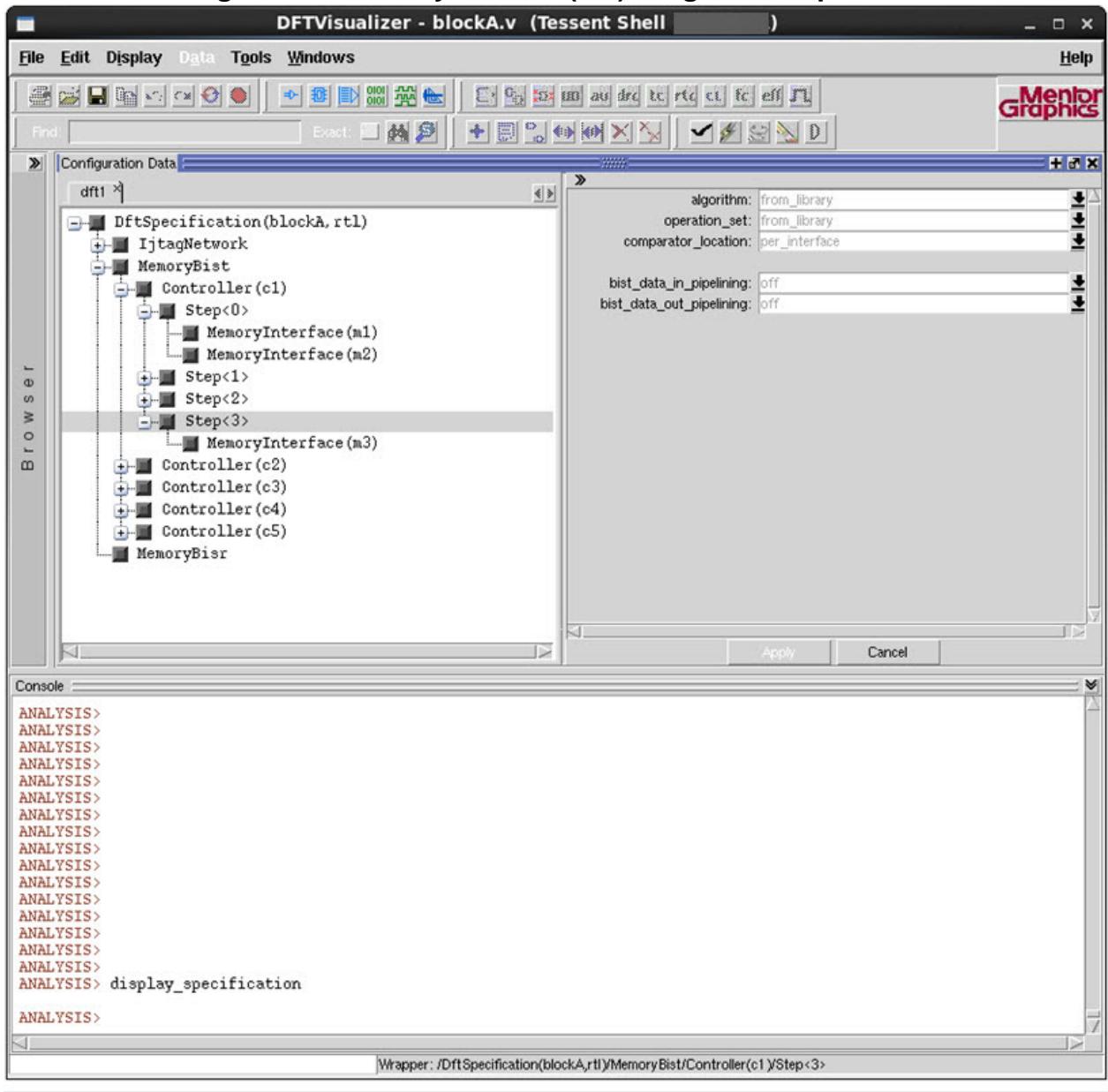


Example 4: Using Drag-and-Drop

Next you need to populate the new step wrapper by a MemoryInterface. In this example, you use the Drag-and-Drop feature of the GUI to move the MemoryInterface(3) from Step<0> to the new Step<3>. For this, you press the left mouse button on MemoryInterface(3), grab it, move the mouse cursor over Step<3> and let go of the left mouse button. You have just dropped the MemoryInterface(m3) to its new step and position in the memory BIST test execution.

Figure 3-4 shows you the final picture of your DftSpecification editing.

Figure 3-4. MemoryInterface(m3) Drag-and-Drop Result

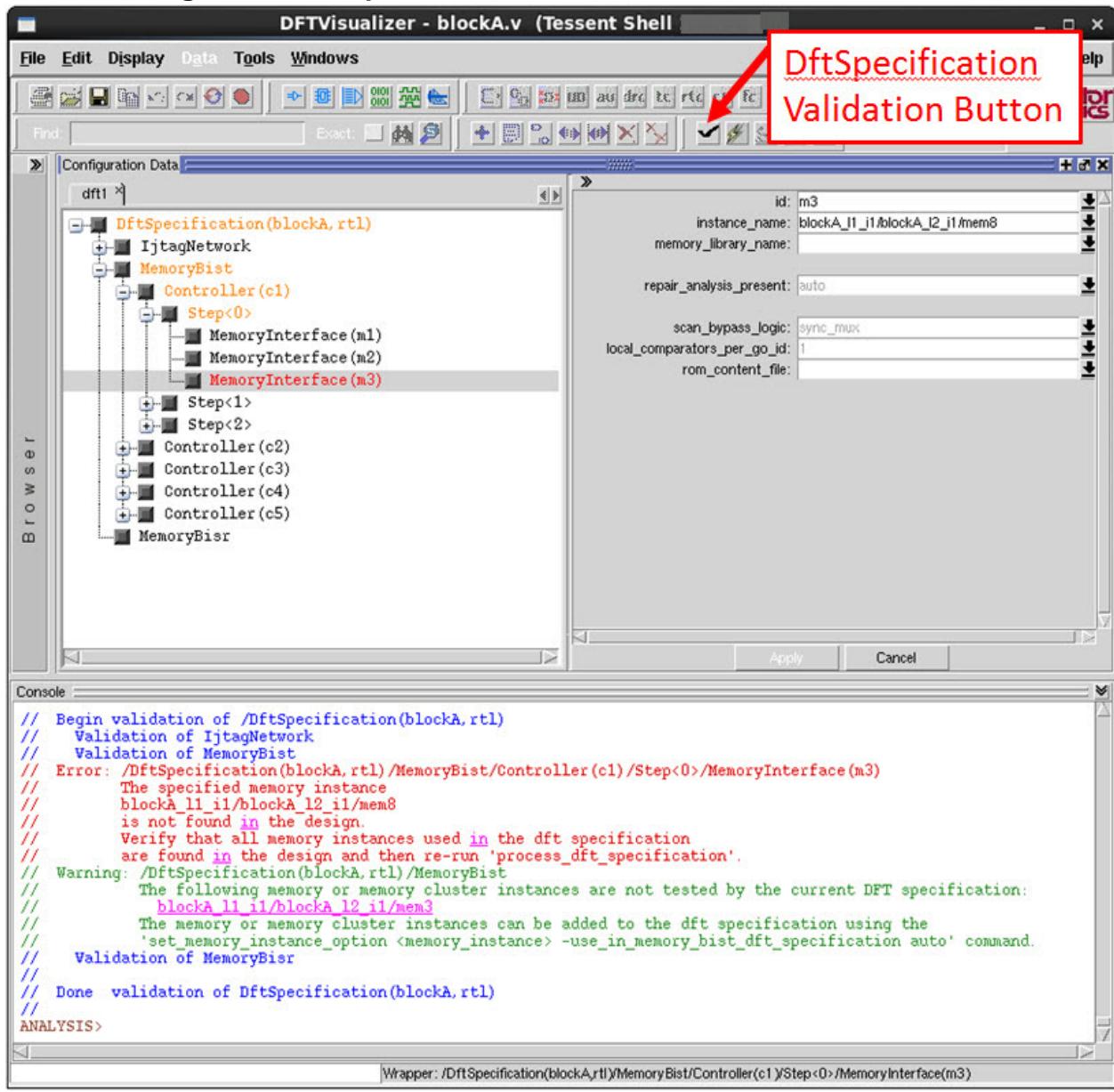


Validating a DftSpecification in the GUI

You should execute a validation step for the DftSpecification you edited either in the GUI, through commands, or manually. The advantage of running the validation step in the GUI 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 push the 'Validate' button as often as you like.

Figure 3-5 shows the color-coding of the path through the DftSpecification to the error point. In this example, there is a spelling error in the design instance pathname to the memory used in the MemoryInterface(m3).

Figure 3-5. DftSpecification Validation Error in DFTVisualizer



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

This section continues after the successful creation of the DftSpecification.

This section describes the steps you have to take to generate the memory BIST and repair logic and insert them into your design.

Example 1: Validating the DftSpecification

If you have not done so, it is recommended to execute 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. Since 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 will generate 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 will then generate all of its RTL as usual, but will 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 will be 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.

Certifying TCD Memory Library Files with memlibCertify in Tesson Shell.....	94
Certification Steps for Memories without Redundancy	95
Certification Steps for Memories with Redundancy.....	101
Limitations	108
memlibCertify Utility Usage in Tesson Shell	109
Pattern Generation for TS-MBIST Insertions	113
Performing Pattern Generation	114
Generated MBIST Verification Patterns.....	117

Certifying TCD Memory Library Files with [memlibCertify](#) in Tessent Shell

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.....	95
Certification Steps for Memories with Redundancy	101
Limitations	108
memlibCertify Utility Usage in Tessent Shell	109

memlibCertify.....	110
memlibCertify Output.....	112

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.

Prerequisites

- If a memory TCD is being certified for a ROM, you will 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.

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 allows 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 will 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      -tshell
                -memLib memlib/singlePort/spram.tcd_memory
                -memLib memlib/dualPort/dpram.lvlib
                -memLib memlib/roms/rom.tcd_memory
                -simModelDir verilog/libs/90nm/rams
                -simModelDir verilog/libs/90nm/roms
                -extension v:vg
```

This example instructs memlibCertify to process three memory library files (*spram.tcd_memory*, *dpram.lvlib* and *rom.tcd_memory*). Note that 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. Refer to the “[memlibCertify Output](#)” section for further information on the generated dofiles and makefile.

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 will be 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 will 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 ROMs**

When a ROM memory library file is included in the certification, the validation step prior to RTL generation will produce an error indicating that the path to the ROM content file is not found, and stop at the Tessent Shell analysis mode prompt.

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. For information on the proper formatting of a ROM contents file, refer to the RomContentsFile property description.

The DftSpecification MemoryBist/Controller/[Step/MemoryInterface\(id\)](#)/rom_content_file property is mandatory for MemoryType “rom” and is initially assigned as “<path_to_rom_content_file>” in the *dft_spec.do* file by memlibCertify. This initial value will create an error since it does not represent a valid file path, as shown in the example below.

```
// command: report_config_data $dft_spec
DftSpecification(memlibc_memory_bist_assembly,rtl) {
    IjtagNetwork {
        HostScanInterface(ijtag) {
            Sib(sti) {
                Attributes {
                    tesson_dft_function : scan_tested_instrument_host;
                }
                Sib(mbist) {
                }
            }
        }
    }
    MemoryBist {
        ijtag_host_interface : Sib(mbist);
        Controller(c1) {
            clock_domain_label : BIST_CLK;
            Step {
                MemoryInterface(m1) {
                    instance_name : mem_container_inst/m1_mem_inst;
                    memory_library_name : ROM_1R_64X23;
                    rom_content_file : <path_to_rom_content_file>;
                }
            }
        }
    }
}
// command: process_dft_specification
//
// Begin processing of
DftSpecification(memlibc_memory_bist_assembly,rtl)
// --- IP generation phase --
// Validation of IjtagNetwork
// Validation of MemoryBist
// Error: /DftSpecification(memlibc_memory_bist_assembly,rtl)/
MemoryBist/Controller(c1)/Step/MemoryInterface(m1)/
rom_content_file
//          The specified file does not exist:
'<path_to_rom_content_file>'.
//
// Done processing of
DftSpecification(memlibc_memory_bist_assembly,rtl)
//
// 'DOFile dft_spec.do' aborted at line 63
ANALYSIS>
```

To assign the correct rom_content_file path, the [set_config_value](#) command is used as shown below for this example:

```
ANALYSIS>set_config_value -in $dft_spec MemoryBist/
Controller(c1)/Step/MemoryInterface(m1)/rom_content_file memlib/
roms/ROM_1R_64X23.data
```

The `report_config_data` command can be used prior to issuing this command to obtain the relevant wrapper ids, and after issuing the command to verify the proper setting for the `rom_content_file` property.

Next, manually complete the remaining steps in the `dft_spec.do` file as shown below:

```
ANALYSIS>process_dft_specification
      .....
INSERTION>extract_icl
      .....
SETUP>exit
```

Alternately one can edit the `dft_spec.do` file and add the `set_config_value` command with the `rom_content_file` property, prior to the `process_dft_specification` step, to automate the process.

Upon completion of Steps 3.a or 3.b, and 3.c for ROMs, the generated RTL will contain 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 Tesson 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 Tesson Shell log file `tshell.log_dft_spec` is created in the working directory during this step and will report 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 Tesson MemoryBIST. You must examine all warnings and investigate their sources before proceeding to the next step.

The generated hardware will be placed in the [Tesson Shell Data Base \(TSDB\)](#) structured sub-directory named `tsdb_outdir` within the working directory.

4. **Generate Testbenches and Patterns** — Generate the testbench and test patterns that will be used to verify that the Tesson 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 will be 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 will report any errors or warnings that were encountered while processing the patterns specification. This file must be checked to make sure the creation of the test bench and patterns were successful. The patterns folder will also be 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 will report 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 will show a pass status with no miscompares reported for each pattern, as shown in the example below. This report will be 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, which 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 will exercise Mentor Graphics’ 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 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 test bench 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 allows 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

- 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 execute 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.

- 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 registers. The commands shown below will add the `extract_repair_fuse_map` property which will allow sampling of the repair analysis fuse registers.

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 will interact with the memory simulation model and perform 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 will modify the task name to what is present in the memory simulation model and adjust 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 *simulation.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 will fail due to the fault injection causing 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 Miscompares  Missing Miscompares  Date
//                               Miscompares
// -----
// ICLNetwork             pass      0                 0
// MemoryBisr_BisrChainAccess  pass      0                 0
// MemoryBist_P1          fail      9                 0
// MemoryBist_ParallelRetentionTest_P1  fail      6                 0
//
SETUP> exit
%

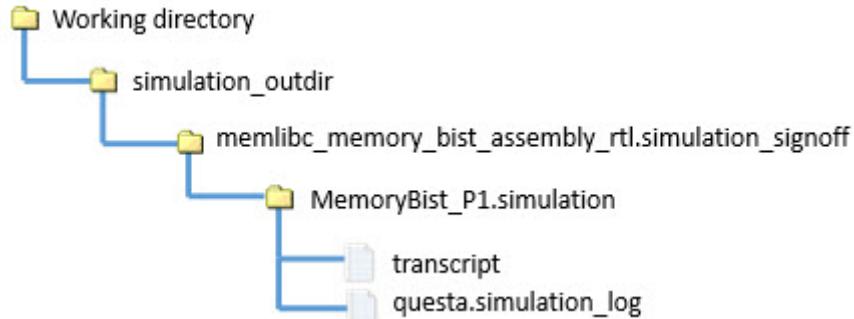
```

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 4-1](#) for a certification performed on a single memory library file with a default tool setup. For the Questa simulator, either the *transcript* or *questa.simulation_log* files can be referenced.

Figure 4-1. Verilog Simulation Log Files for Single Memory Certification



A portion of the Verilog simulation log for this example is shown in [Figure 4-2](#). The repair analysis engines are implemented in the memory interface. Each memory instance has a repair status register in the BIRA hardware named `<memory_instance_name>_interface_inst.RA_INTERFACE_STATUS_REG[x]`. 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

Bit1 Bit0	Repair Status
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 will 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 4-2. 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 4-2](#) the **.RA_INTERFACE_rs_SPARE0_ALLOC_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 4-3](#). 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 together identifies 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 will cover 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 4-3. 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.

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 Tesson 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, which are explained in the “[Certification Steps for Memories with Redundancy](#)” section.
- The utility does not validate physical address and data mapping that are described in the [PhysicalAddressMap](#) and [PhysicalDataMap](#) wrappers of the memory library file.
- The tool does not validate the [MilliWattsPerMegaHertz](#) property in the memory library file.
- For ROMs, the generated *dft_spec.do* file must be modified to specify the location of the ROM contents file. The additional steps are explained in the “[Generation from Tesson Shell Invocation](#)” section.
- Disabling the generation of BISR logic for memories that support self repair requires additional manual steps. For further information, refer to the LV flow [-disableSelfRepair](#) argument description for the [memlibCertify](#) utility.
- By default, the *simulation.do* file created by the utility does not pass simulation options and utilizes the Questa simulator from Mentor Graphics. For information on how to specify other simulators that are supported by Tesson Shell, refer to the LV flow [-simulator](#) argument description for the [memlibCertify](#) utility. For information on how to pass simulator options, refer to the LV flow [-verilogOptionFile](#) argument description.

memlibCertify Utility Usage in Tesson Shell

memlibCertify	110
memlibCertify Output.....	112

memlibCertify

A tool that automates the process of validating memory library files.

Usage

```
memlibc [-extension ext1 [:ext2] ...]
         [-memLib filename]...
         [-quiet]
         {[[-simModelDir dir_path] | [-simModelFile file_path]...}
         {[[-tshell] | {
             [-simulator ModelSim | Verilog-XL | NC-Verilog | VCS]
             [-disableSelfRepair]
             [-verilogOptionFile filename]}}}
         [-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. For information on its 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.

Arguments

- **-extension *ext1* [:*ext2*]...**
Use this repeatable option to control the file extension naming (without the dot) used in conjunction with the -simModelDir option. These extensions define the order in which 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*.
- **-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.
- **-quiet**
Use this option to reduce the verbosity of memlibCertify when processing memory library files.
- **-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 optional and repeatable, but you must specify either the -simModelDir option or the -simModelFile option.

- 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 optional and repeatable, but you must specify either the -simModelFile option or the -simModelDir option.

- -tshell

This option is used to specify memory library file certification for the Tessent Shell flow. When specified, memlibCertify generates the necessary Tessent Shell “.do” files as well as a Makefile that can be utilized to complete the certification process. This argument is mandatory for Tessent Shell flow usage. For further information on the overall process, refer to “[Certifying TCD Memory Library Files with memlibCertify in Tessent Shell](#)”.

- -simulator ModelSim | Verilog-XL | NC-Verilog | VCS

This optional argument is ignored when the -tshell option is specified for Tessent Shell certification flow. It is only valid for the LV flow.

For Tessent Shell flow, the default simulator is Questa. To specify a simulator other than Questa, you will need to edit the simulations.do file created by memlibCertify, and add the -simulator switch and value pair to the [run_testbench_simulations](#) command.

- -disableSelfRepair

This optional argument is ignored when the -tshell option is specified for Tessent Shell certification flow. It is only valid for the LV flow.

For Tessent Shell flow, memlibCertify automatically generates BISR logic for memories that have the following wrappers in their memory library file:

- [RedundancyAnalysis/ColumnSegment/PinMap](#)
- [RedundancyAnalysis/RowSegment/PinMap](#)

If it is desired to prevent generation of Built-In Self-Repair (BISR) logic for memories that support self repair, edit the dft_spec.do file created by memlibCertify and add the -memory_bisr_chains argument and setting as follows:

```
set_dft_specification_requirements -memory_test on
                                  -memory_bisr_chains off
```

- -verilogOptionFile *filename*

This optional argument is ignored when the -tshell option is specified for Tessent Shell certification flow. It is only valid for the LV flow.

For Tessent Shell flow, if it is desired to forward simulator-specific options to be used during simulation, one must edit the simulations.do file created by memlibCertify and add the desired simulation options to the [run_testbench_simulations](#) command. These are simulator specific options and must be consistent with the simulator being utilized by the [run_testbench_simulations](#) command.

For example, if you are using the default Questa simulator, the following method can be used to pass in the -debugDB simulator option so the vsim.dbg file is created and can then be accessed using the schematic viewer when debugging your waveforms. You must specify this simulator option in conjunction with the -store_simulation_waveform switch for the run_testbench_simulations command. To accomplish this, the simulation.do file would be edited to add the following arguments to the run_testbench_simulations command in addition to the arguments already present in the generated dofile:

```
run_testbench_simulations -store_simulation_waveforms on \
-simulator_options "-debugDB"
```

- **-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 execute 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.

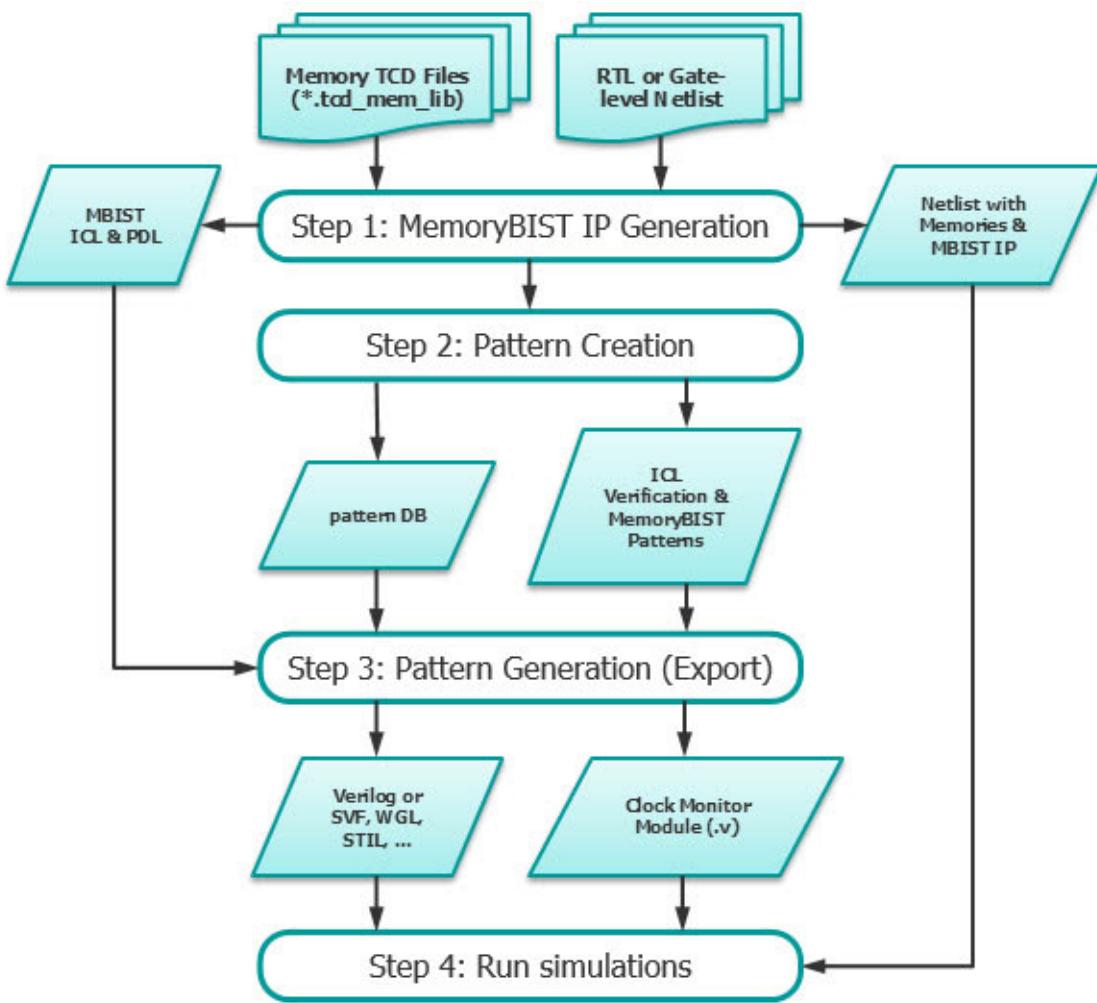
Target	Description
<hr/>	
make gen	Generate the memoryBist hardware for memory templates.
make testbench	Generate the Verilog testbench for the memoryBist controllers.
make sim_rtl	Executes the Verilog simulation for the memoryBist controllers.
make clean	Deletes the TSDB and simulation output directories.
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 and is not used in the Tessent Shell memory library certification flow. Refer to the -verilogOptionFile argument description for [memlibCertify](#) for information on forwarding simulator-specific options that are to be used during simulation.

Pattern Generation for TS-MBIST Insertions

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

Figure 4-4. Pattern Generation



You generate the following:

- For the bottom-up flow, core-level Verilog testbenches
- For the top-down flow, top-level patterns (including Verilog testbenches)

The tool supports all test pattern formats currently supported for ATPG.

Performing Pattern Generation

Use this procedure to generate simulation testbenches or test patterns that can be applied from a tester. Tessent MemoryBIST uses a similar flow for generating patterns at the block, core or top level.

You typically generate only Verilog testbenches at the block or core level, since that portion of the circuitry will normally get instantiated in a larger (i.e. top) design. There is therefore no need to generate manufacturing test patterns at those levels.

At the top level, Verilog testbenches are generated before signing-off the design. Tester (ATE) test patterns are then created, to be applied 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 allows completing MBIST insertion in smaller design portions, even though the rest of the design may not be ready yet. Since 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 (i.e. when the chip is essentially completed and just about to tape out), so unexpected delays may impact the design schedule.

Prerequisites

The required inputs for this step of the flow that you specify in the pattern generating dofile are as follows:

- The TSDB (i.e. 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.

Procedure

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

```
% tessent -shell
```

3. Set the tool context to IJTAG patterns mode using the [set_context](#) command as follows:

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

4. 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
```

5. Unless it is already in memory, read the current design's extracted ICL with the [read_icl](#) command. For example:

```
SETUP> read_icl \
./tsdb_outdir/dft_inserted_designs/blockA_rtl.dft_inserted_design/
blockA.icl
```

6. 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.

7. Create a pattern specification using the [create_patterns_specification](#) command and then process it with the [process_patterns_specification](#) command:

```
SETUP> create_patterns_specification
SETUP> process_patterns_specification
```

Memory BIST patterns are generated at this point and can also be exported as Verilog, PDL or any of the supported formats (WGL, STIL or SVF) if needed for ATE tests.

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

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

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

```
SETUP> run_testbench_simulations
```

10. 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_icl \
./tsdb_outdir/dft_inserted_designs/blockA_rtl.dft_inserted_design/
blockA.icl
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 \[Tessent Shell Reference Manual\]](#)

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. This clock will be created as a clock generator in Verilog testbenches (i.e. for simulations) but would typically be provided by the tester when creating manufacturing patterns for silicon.

A single memory BIST pattern file contains one or several [TestStep](#) wrapper(s). Test steps are executed sequentially and execute specific memory BIST and/or BISR controllers actions.

Memory BIST controllers can execute algorithms across the entire address space being tested. This verification step may be considered overkill 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 will 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 will be 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) which resulted in the expected vs. observed data comparison error.

Tip

 **Note:** 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 Tesson Shell MemoryBIST flow.

Overview of Memory Repair Capabilities	123
Types of Repair	123
Repair Steps	123
Chip Level Repair Architecture	123
Built-in Repair Analysis (BIRA)	125
Built-in Self Repair (BISR)	126
Implementing Soft Repair	127
Memory Library Preparation and Repair Registers Description	129
Repair Analysis for IO/Column Elements	130
Repair Analysis for Row Elements	147
Repair Analysis for Row and IO/Column Elements	158
Built-In Self-Repair	169
Implementing BIRA and BISR Logic	180
Inserting BISR Chains in a Block	181
Determining the Fuse Box Size	186
Creating and Inserting the BISR Controller	190
Top-Level Verification and Pattern Generation	197
Fuse Box Programming	198
Verify BISR at the Block Level	200
Verifying Top-Level BISR	203
Creating Multi-Load Scan Patterns with Repairable Memories	216
Generating Your Manufacturing Test Patterns	218
Compression Algorithm and Fuse Box Organization	229
CompressBisrChain Script	231
Incremental Repair	235
Incremental Repair Overview	235
BIRA Initialization	237
BIRA Repair Status Bits Checking	238
Absence of Fuse Programming Step	245
Handling of Blocks Without Incremental Repair Capability	245
Considerations Specific to Hard Incremental Repair	246
Repair Sharing	248
Repair Sharing Overview	248
Repair Sharing Conditions	248
Implementing Repair Sharing	252

Fast BISR Loading.....	259
Fast BISR Loading Overview.....	259
Fast BISR Architecture.....	259
Implementing Fast BISR Loading	264

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	123
Repair Steps	123
Chip Level Repair Architecture.....	123
Built-in Repair Analysis (BIRA)	125
Built-in Self Repair (BISR).....	126
Implementing Soft Repair.....	127

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

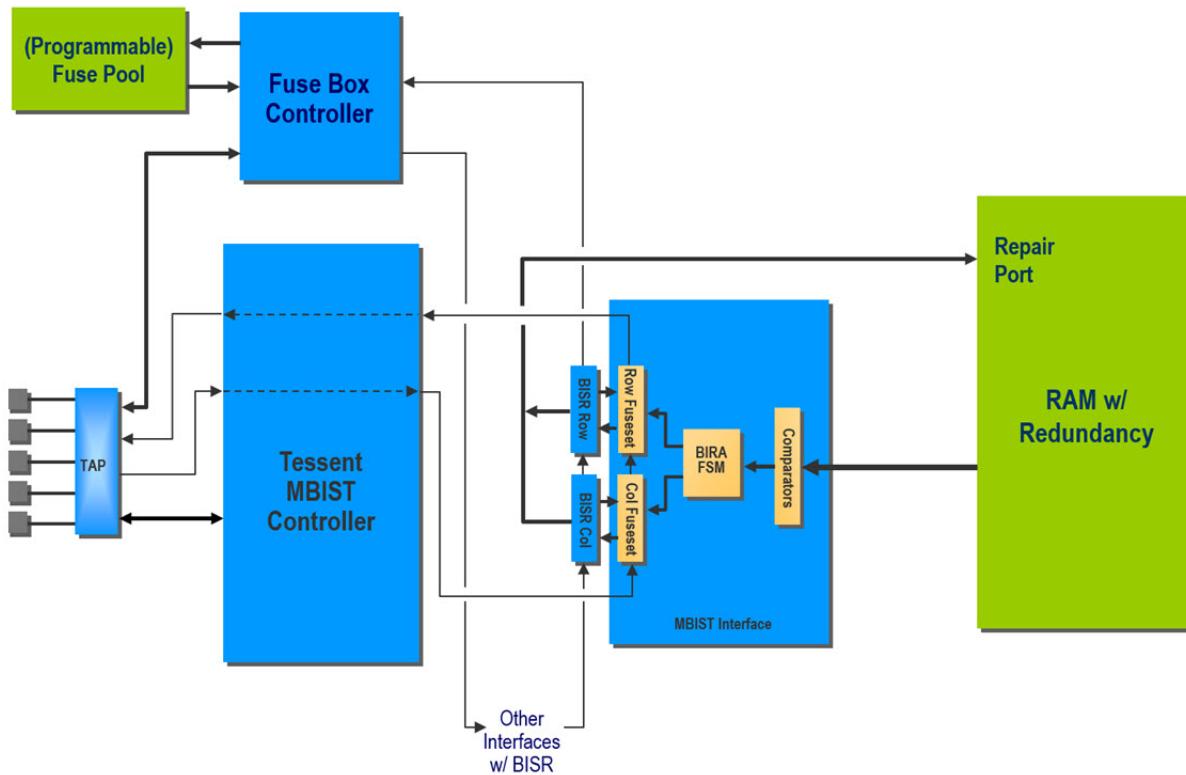
Memory repair is performed in two steps. The first step consists of analyzing failures reported by the memory BIST controller during test execution to determine if the memory is repairable and, if repairable, determining the values to be applied to the repair inputs of the memory.

Chip Level Repair Architecture

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

[Figure 5-1](#) 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 and/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 allows 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 will not always find a repair solution even if one exists. The reason is that the engine does not perform an exhaustive search of the repair solutions in order to minimize area and test time. By default, the engine assigns all spare columns first unless a multi-bit error at a same address is encountered. In that case, the engine assigns a spare row even if spare columns are available. This repair strategy will not find a solution for some combinations of errors with a common row or column address appearing in a certain sequence. However, the low number of these combinations will have negligible impact on yield.
- When several ShiftedIO properties are specified in the Memory BIST library with the same `bitString` value, the IO/Column BIRA will not always find a repair solution even if one exists. The memory will be 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.

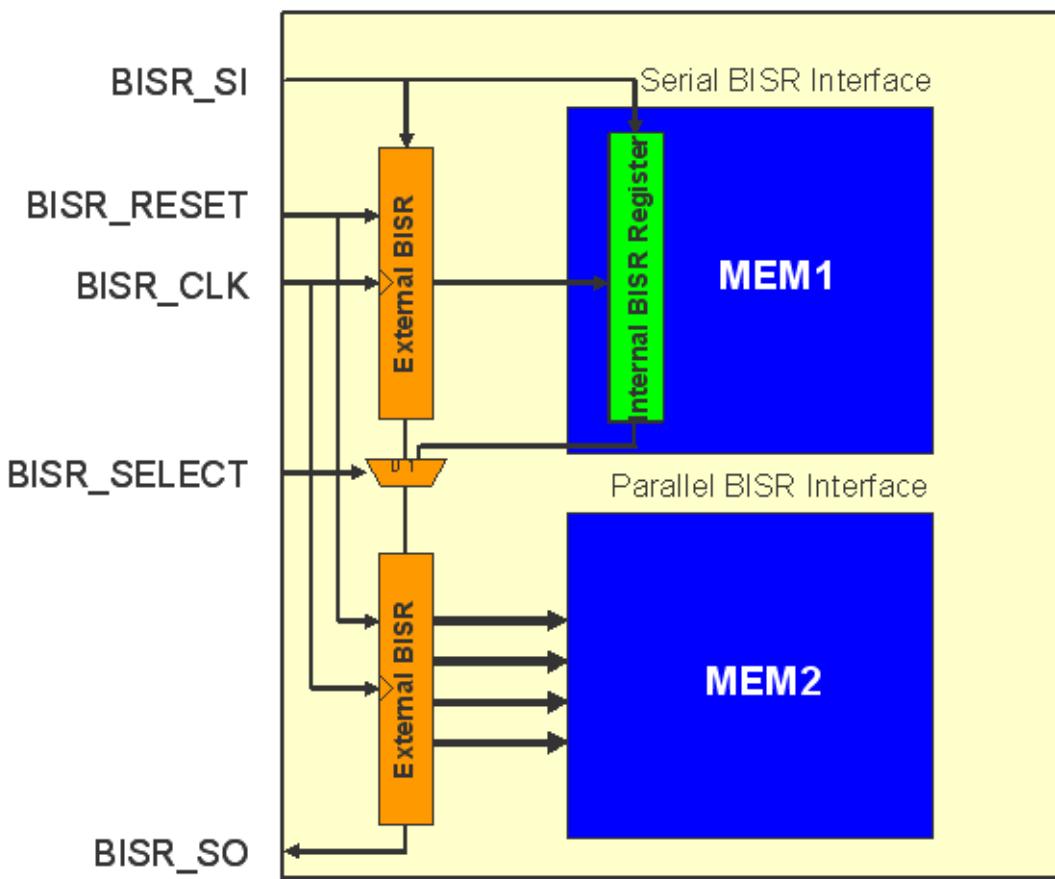
Built-in Self Repair (BISR)

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 which allows 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-execute 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 since 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 allows significant area savings since it does not require the presence of an eFuse in the design. Also, this method allows 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 in order to avoid resetting the BISR chains. For the same reason, the IJTAG network and/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 will not be 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.....	130
Repair Analysis for Row Elements.....	147
Repair Analysis for Row and IO/Column Elements.....	158
Built-In Self-Repair	169

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](#)” section for further information. Refer to the “[Verify BISR at the Block Level](#)” section for information on validating the repairable memories and BIRA/BISR logic at the block level.

IO/Column Repair	131
Implementing IO/Column Repair Analysis.....	133
Repair Analysis Registers	143
Repair Analysis Output Ports for IO/Column Repair	145

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.

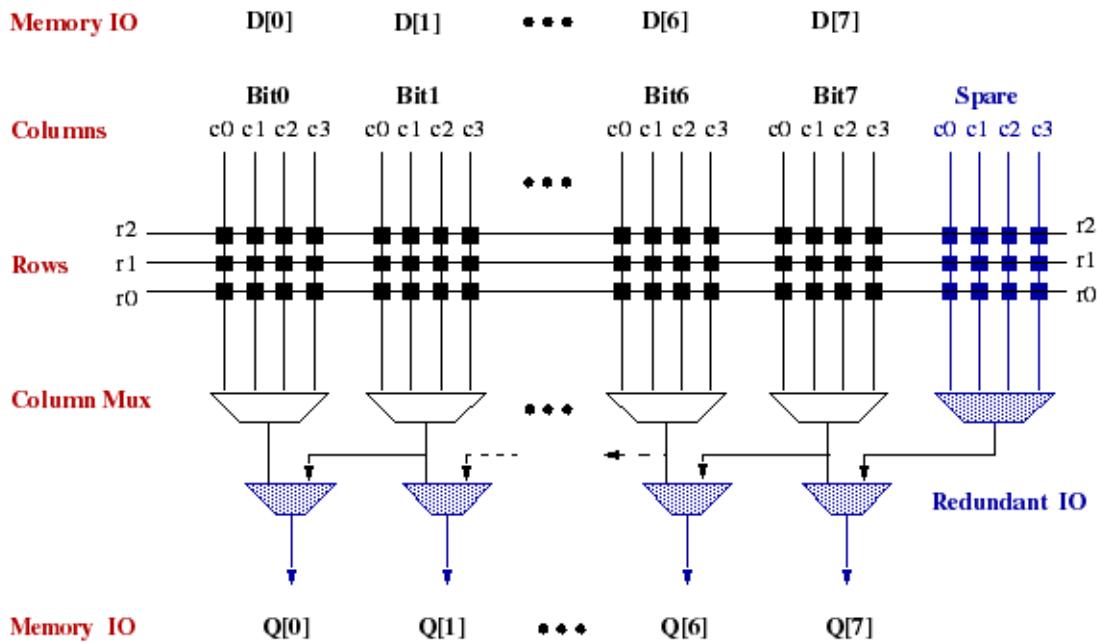
IO Replacement	131
Column Replacement	132

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

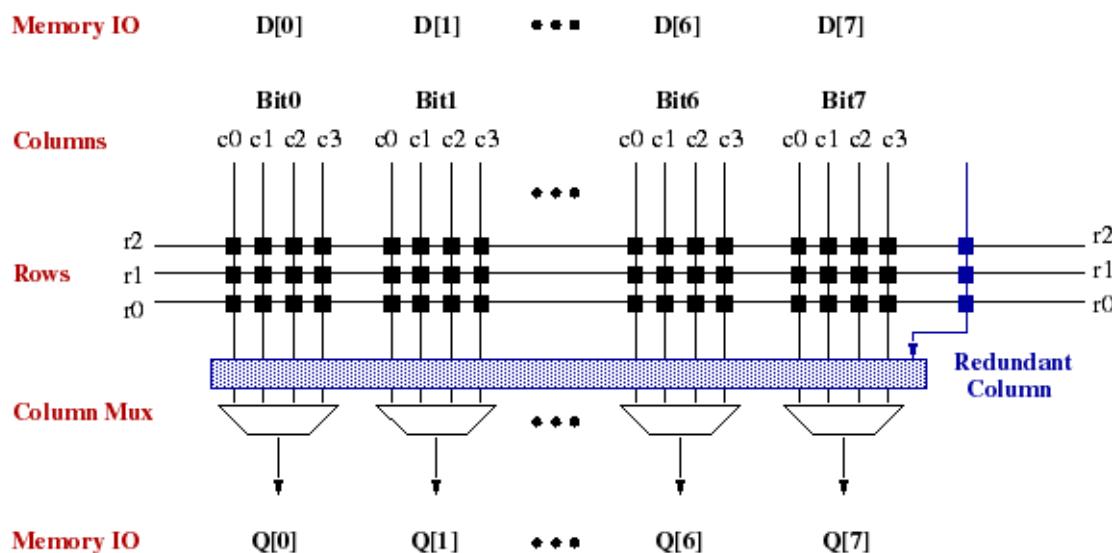
To identify the failing IO, the built-in repair analysis circuit must log the defective IO value.

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	133
ColumnSegmentRange Wrapper	134
ColumnSegment Wrapper	135
Memory Library File Sample Syntax	136

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 be specified if only one [RedundancyAnalysis/ColumnSegment](#) wrapper is defined. In this configuration, the column segment will encompass 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 bit mapping 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.

Note

 The *FuseMap* wrapper is independent of the **Fuse** property.

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 will identify each defective IO.

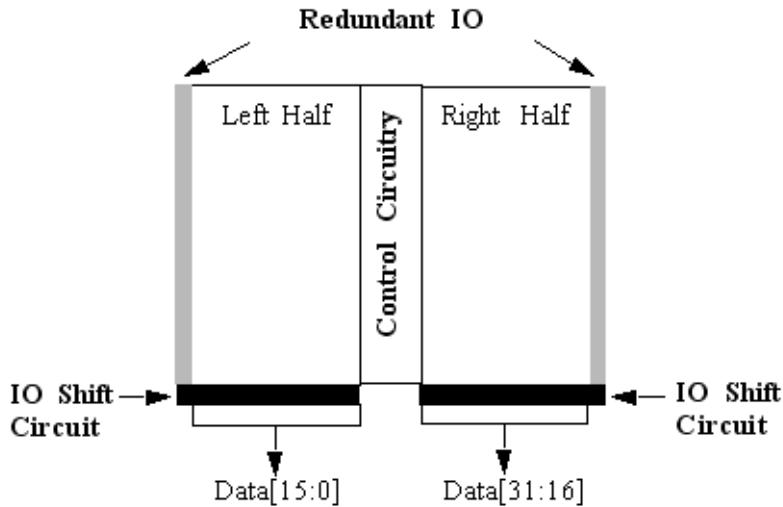
Memory Library File Sample Syntax

This section provides examples of the memory library sample syntax.

Example 1

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 will be 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

```

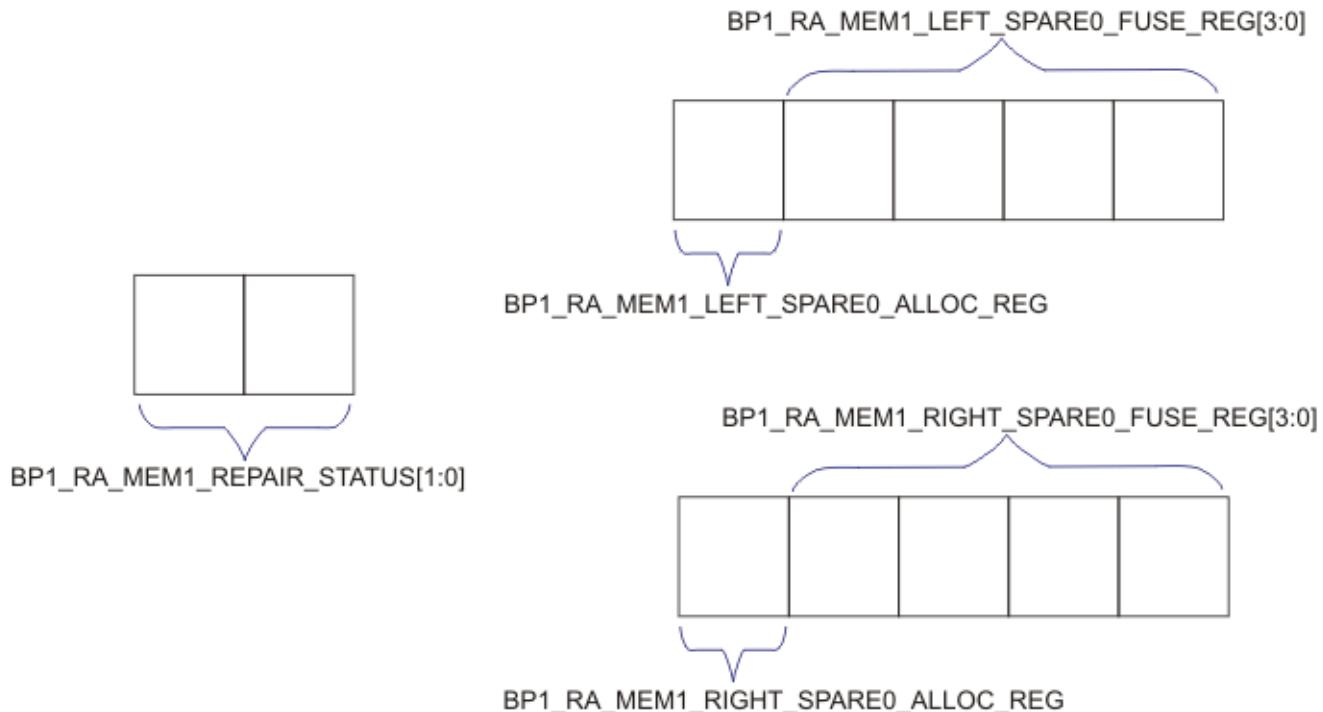
RedundancyAnalysis {
    ColumnSegment (LEFT) {
        ShiftedIORange: Data[15:0];
        FuseSet {
            FuseMap [3:0] {
                ShiftedIO(Data[0]): 4'b0000;
                ShiftedIO(Data[1]): 4'b0001;
                ShiftedIO(Data[2]): 4'b0010;
                ShiftedIO(Data[3]): 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

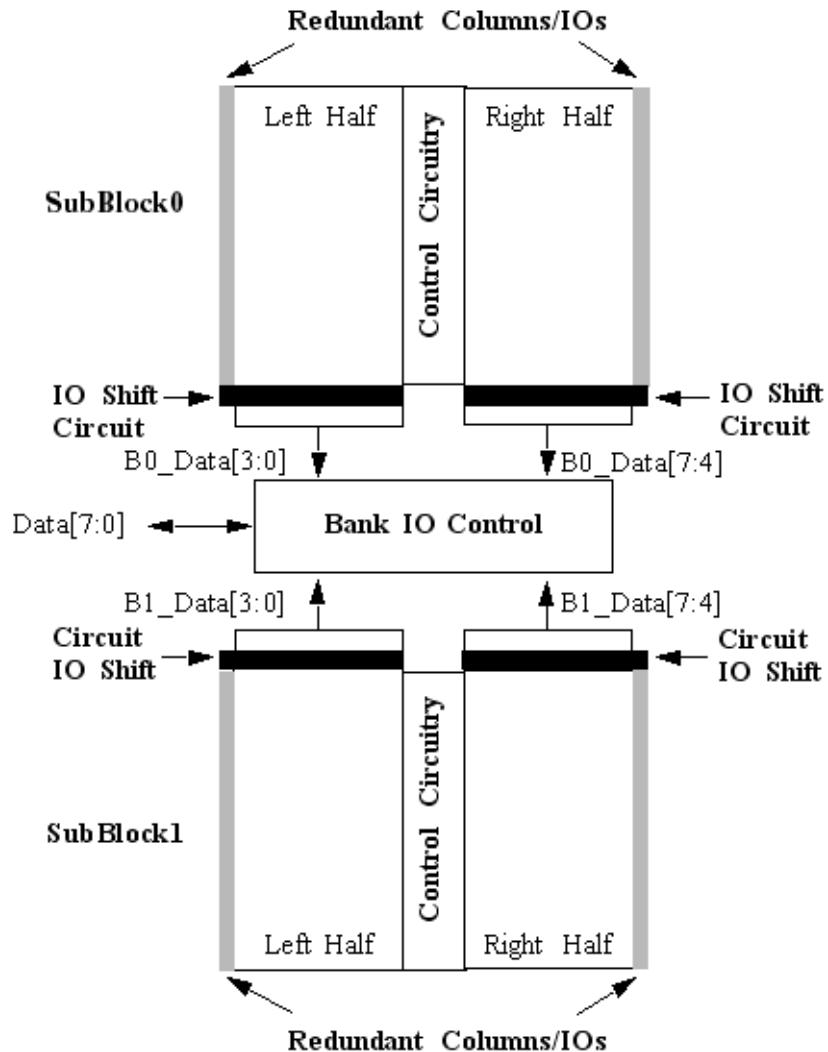


For information on the built-in repair analysis registers, refer to the “[Repair Analysis Registers](#)” section.

Example 2

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 will be programmed by a 5-bit fuse. The fuse box bit mapping 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;
                }
            }
        }
    }
}

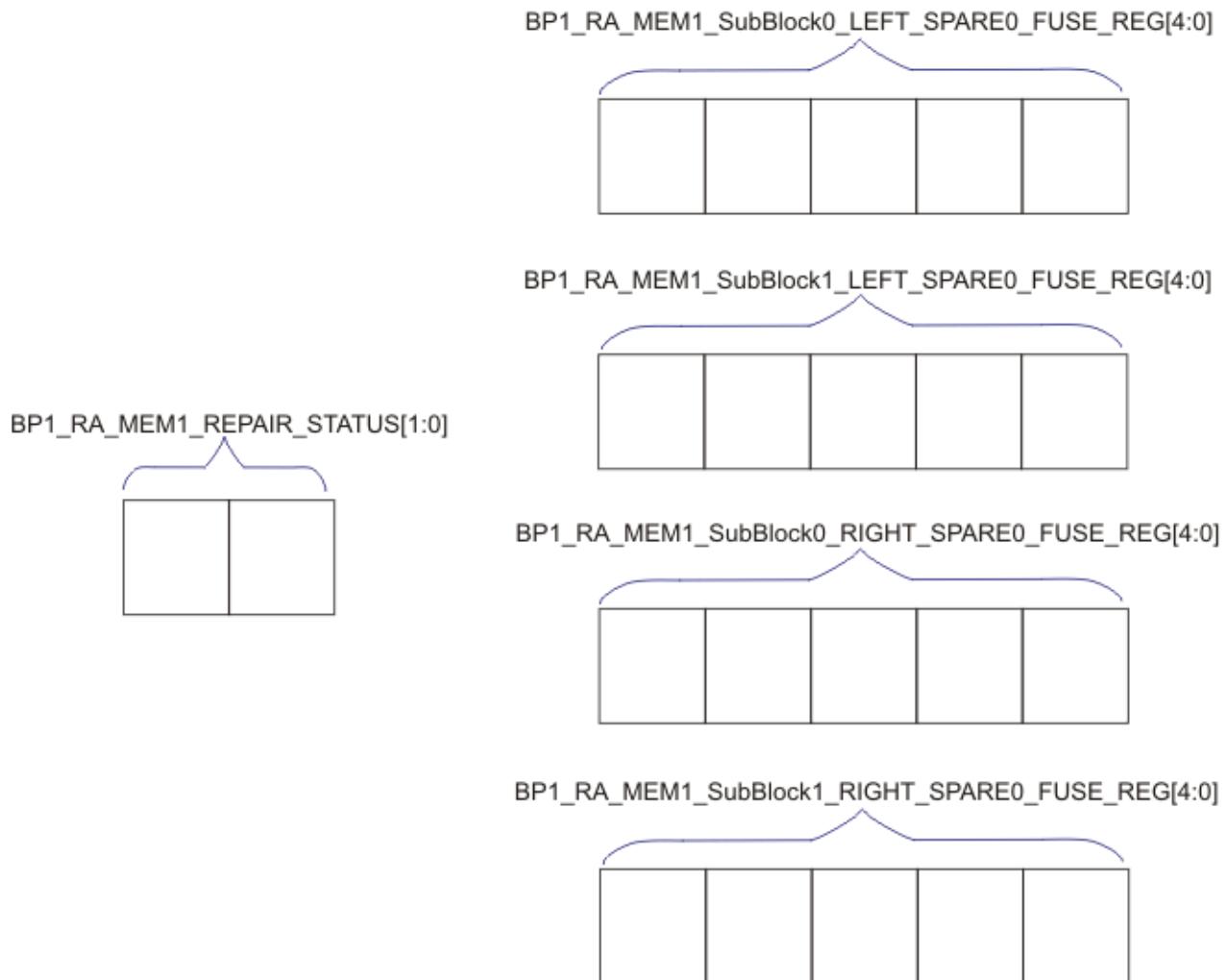
```

```
    }
```

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 will not include the allocation bit because the NotAllocated property is being specified in the [FuseMap](#) definition.

Figure 5-11. FuseSet Registers for Example 2



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 Tesson 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 IO/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 [143](#)

Fuse Registers [144](#)

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 IO 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..... 145

BIRA Port Names on the Interface When Using Local Comparators 145

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:
 - “[Repair Status Register](#)”
 - “[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](#)” section for further information. Refer to the “[Verify BISR at the Block Level](#)” section for information on validating the repairable memories and BIRA/BISR logic at the block level.

Row Repair	148
Implementing Row Repair Analysis	149
Repair Analysis Registers	154
Repair Analysis Output Ports for Row Repair	156

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.

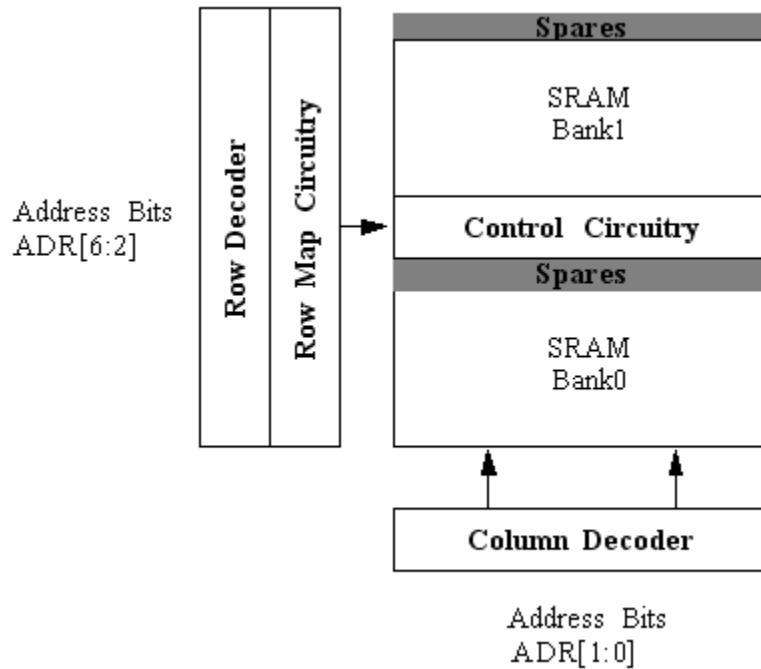
Row Replacement 148

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 will always replace 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	149
RowSegmentRange Wrapper	150
RowSegment Wrapper	151
Memory Library File Sample Syntax	152

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. which address ports are used to define the range of each memory segment.

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:

- A minimum of one Fuse must be defined per FuseSet.
- name must be either a scalar port or a single bit of a bused port.
- 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

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 Tesson 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 [154](#)

Fuse Registers [155](#)

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 Bit 0	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..... 156

BIRA Port Names on the Interface When Using Local Comparators 156

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](#).

- $<\text{prefix}>_{_}\text{FUSE_REG}$

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

- $<\text{prefix}>_{_}\text{FUSE_ADD_REG}$

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

- $<\text{prefix}>_{_}\text{ALLOC_REG}$

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:
 - “[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](#)” section for further information. Refer to the “[Verify 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	159
Repair Analysis Registers	167
Repair Analysis Output Ports for IO/Column and Row Repair.....	168

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..	159
RowSegmentRange and ColumnSegmentRange Wrappers	161
RowSegment and ColumnSegment Wrappers	162
Sample Memory Library File Syntax	163

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
            .

        } //End of FuseSet
        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 in which 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:

- A minimum of one Fuse must be defined per FuseSet.
- *name* must either be a scalar port or a single bit of a vectored port.

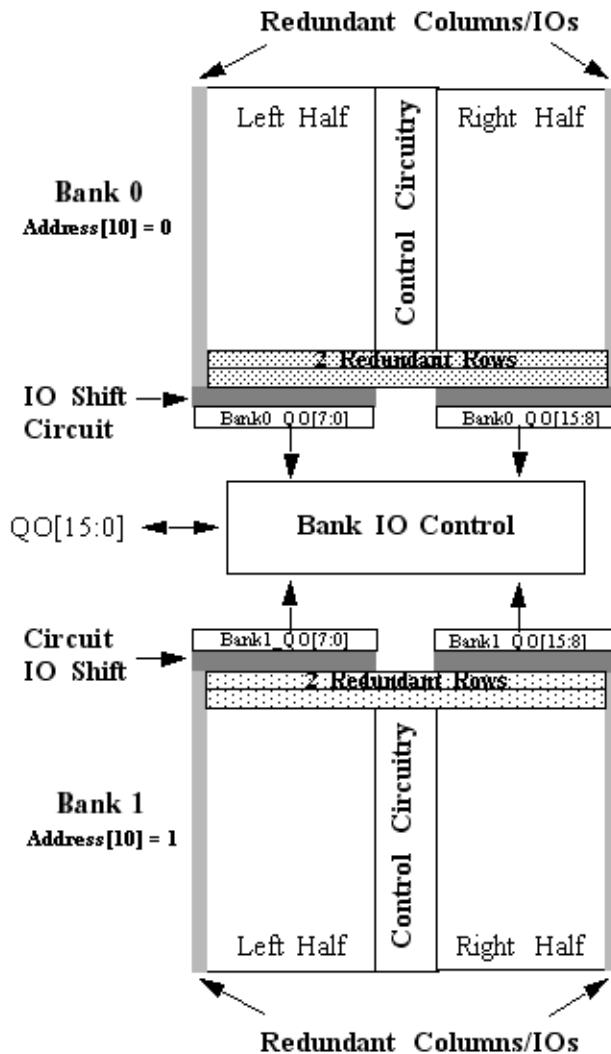
- name must be a port defined a **Port** wrapper with Function: Address in the memory library file. Constant LogicHigh and LogicLow values can be used instead of address bits.

Sample Memory Library File Syntax

An example memory library implementation will be shown 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(DO[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 columns 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 column 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 Tesson 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 the memory controller and/or interface.

Repair Status Register	167
Fuse Registers	167

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

See the “[Repair Analysis Output Ports for IO/Column Repair](#)” and “[Repair Analysis Output Ports for Row Repair](#)” sections.

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	169
Serial BISR Interface.....	169
BISR-Specific Memory Library Syntax	171
Memory Library Examples with Parallel and Serial BISR Interface	176

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 need to match the internal memory BISR register order. The internal BISR register ordering is provided in the memory data sheet. Any mismatch between the order of the internal BISR chains and the order specified by the data sheet will result in incorrect repair data scanned inside the memory. We 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 will quickly identify 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 Tesson 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.....	173
PinMap SpareElement Wrapper	173

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 will be specified with the following port functions, as specified in the Function property of the [Port](#) wrapper:

- For *Parallel* BISR interface:
 - BisrParallelData
- For *Serial* BISR interface:
 - BisrSerialData (with [Port](#)/Direction : input | output;)
 - BisrClock (with [Port](#)/Polarity : (activeHigh) | activeLow;)
 - BisrReset (with [Port](#)/Polarity : (activeHigh) | activeLow;)
 - 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 allows 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 will generate and instantiate 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 will 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 [RowSegment](#) and/or [ColumnSegment](#) wrappers.

- When specified inside the RowSegment wrapper, 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.
- When specified inside the ColumnSegment wrapper, only a single SpareElement wrapper may be specified inside the [PinMap](#) wrapper.

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:
 - The `repairPortName` must be either a scalar port or a single bit of a vectored port.
 - The `repairPortName` must be a port defined in the memory library file corresponding to a [Port/Function](#): `BisrParallelData` setting.
- For the Serial Repair Interface:
 - The `RepairRegister[x]` value must be specified where x is the index of the Repair Enable register inside the BISR chain.
 - 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 to which this BISR is connected 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](#)”

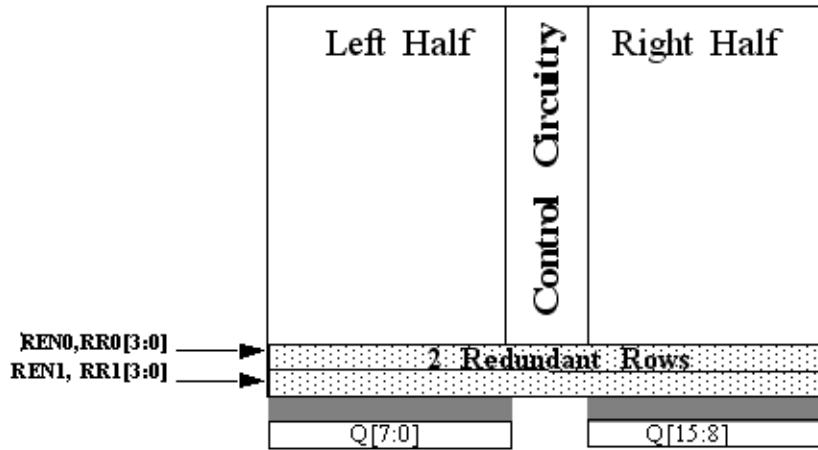
This example shows a memory with one redundant column using a *serial* BISR interface. The redundant column allocation is accessed using the serial BISR ports on the memory module.

Row Repair and Parallel BISR Interface	176
Column Repair and Serial BISR Interface	177

Row Repair and Parallel BISR Interface

The example in [Figure 5-19](#) shows 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(RENO) {
    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: RENO;
                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 [Figure 5-21](#) 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

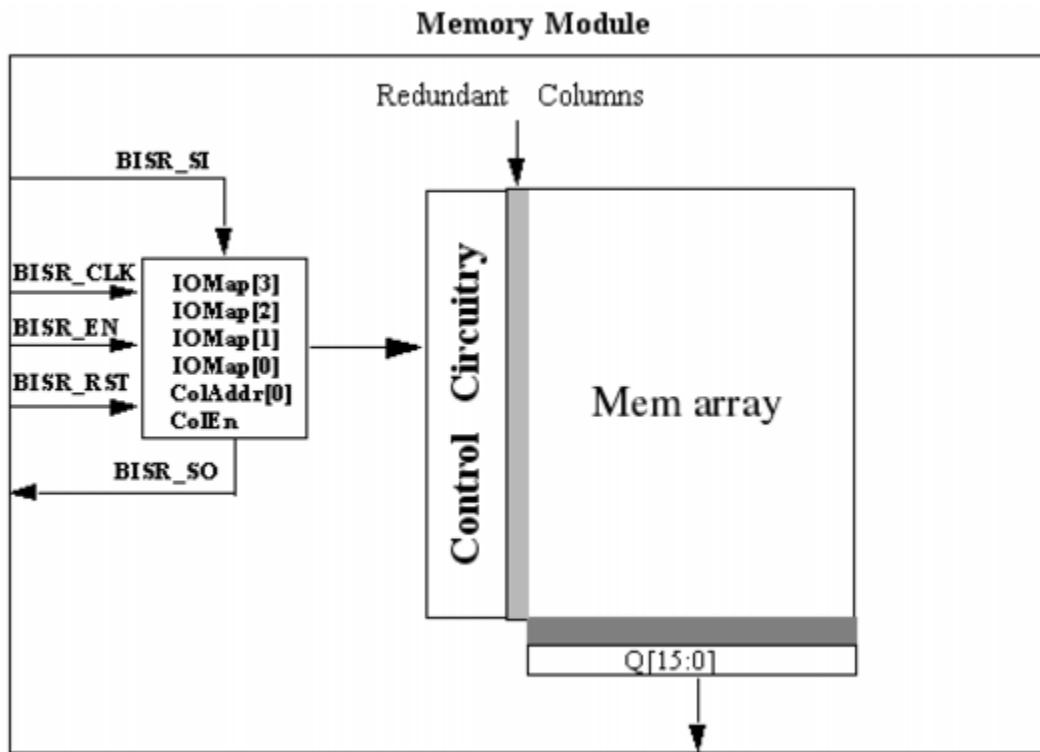


Figure 5-22 illustrates the memory library file fragment associated with the memory illustrated in Figure 5-21.

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	181
Determining the Fuse Box Size.....	186
Creating and Inserting the BISR Controller.....	190

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	181
Controlling the BISR Chain Order	182
Disabling the Insertion of BISR registers	184
Excluding Child Block BISR Chains	184

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 which 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 Tesson 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`.

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 will 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 the user wishes 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
```

will 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 will be 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 which is automatically generated in the current directory when entering the analysis mode, or when the [create_bisr_segment_order_file](#) command is executed. Analysis mode is automatically entered when [check_design_rules](#) is executed without any DRC errors occurring.

In the example file below, there are three power domains, the default power domain (-) and two other power domains (pd_A and pd_B). The default power domain is assigned to all repairable memories which are not explicitly assigned to a domain. It is often associated to the portion of the design whose power is never switched off.

```

//-----
// File created by: Tesson 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;
        }
    }
}

```

If the default BISR chain order is not satisfactory, there are two ways to affect BISR chain ordering. The preferred method is to read in a DEF file corresponding to the design using the `read_def` command, and create BISR segment ordering that is automatically optimized based on memory placement and physical block or sub-block BISR_SI pin coordinates. The physical blocks and sub-blocks will have memoryBIST insertion and BISR chain ordering done separately. The file should be loaded in Tesson Shell during the setup mode at the same time as the other design files are read in. For example, the command

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

reads the DEF file corresponding to design modA. The BISR chain ordering will be determined 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 have the coordinates specified, but the parent block does, those coordinates will be used for ordering optimization.

The second way consists of manually modifying the order of memory instance names of the `<design_name>.bisr_segment_order` file before executing the [process_dft_specification](#) command. If `check_design_rules` is re-run either during the same Tesson Shell invocation or a subsequent one, and you wish to reuse the modified `<design_name>.bisr_segment_order` file, you need to specify the [set_memory_instance_options](#) when still in setup mode. For example, the command

```
SETUP> set_memory_instance_options -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 executed. Note that the file name of the bisr_segment_order_file is arbitrary and does not need to always be the default name.

Disabling the Insertion of BISR registers

It is possible to disable the generation of BISR registers for specific memory instances by issuing 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 will disable the generation of the BISR register for memory instance mem4 even if the memory has spare resources. This option is, however, rarely used.

BISR insertion can also be disabled 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-executing 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 executed. If check_design_rules is subsequently re-run and you wish to reuse the modified *<design_name>.bisr_segment_order* file, you need to specify the [set_memory_instance_options](#) when still in setup mode. For example, the command:

```
SETUP> set_memory_instance_option -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 executed. 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 will result and point 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.

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 will help 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	186
Multi-Chain Case.....	187
Incremental Repair Case.....	188

Single-Chain Case

The single-chain case formula is as follows:

$$\begin{aligned} \text{NumberOffuses} = & (\text{NumberOfRepairs} * \text{buffer_size}) \\ & + ((\text{NumberOfRepairs} + 1) * (\text{zero_counter_bits} + 1)) \end{aligned}$$

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 (e.g., scan tests), making the chips non-repairable and/or posing a reliability problem.
- *buffer_size* is the size of the largest BISR register in the BISR chain. The size typically varies between 6 and 12. Remember that a BISR register is associated with a single spare resource (that is, spare row or spare column). A memory might have several spare resources.

- *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 *log2* 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 executed. An example of this file is shown in [Figure 5-23](#). 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 5-23. 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:

$$\text{NumberOfFuses} = (\text{NumberOfRepairs} * \text{buffer_size}) + ((\text{NumberOfRepairs} + 1) * (\text{zero_counter_bits} + 1)) + (\text{NumberOfGroups} - 1) * (\text{fuse_box_address_bits} + (\text{zero_counter_bits} + 1))$$

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 simply will be the log2 of the number of fuses in the fuse box, and the value is reported in the MBISR TCD file as shown in [Figure 5-23](#) and as described in the “[Single-Chain Case](#)” section. However, if the fuse box is shared for other purposes, the number of address bits will be 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 5-23](#) 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 allows 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 simply will be the log2 of the number of fuses in the fuse box, and the value is reported in the MBISR TCD file as shown in [Figure 5-23](#) and as described in the “[Single-Chain Case](#)” section. However, if the fuse box is shared for other purposes, the number of address bits will be larger.

Creating and Inserting the BISR Controller

This sections covers BISR controller function, implementation and operation.

Understanding the BISR Controller	191
Connecting the BISR Controller to an External Fuse Box	193
Connecting a BISR Controller to System Logic	194
Choosing a Functional Repair Clock	195

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 which 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 the *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 192

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. Six autonomous operations can be performed: *self_fuse_box_program*, *power_up_emulation*, *verify_fuse_box*, *rotate_bisr_chain*, *calculate_bisr_chain_length* 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. Due to 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, Mentor Graphics 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 fuse box controller.

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 will be 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 will be 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. 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 will first be disconnected and then connected to the BISR controller module.

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 fuse box controller and the fuse box interface will be 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 will be 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 under the MemoryBisr:Controller. The system logic must provide a minimum of two inputs 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.

If the design contains multiple power domain groups, 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 several power domains are present, 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 if power domain will be 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 10 MHz or less in that functional mode.

Note that all other modes used for manufacturing are using the TAP clock (TCK), which is also 10 MHz by default. Using such a low frequency simplifies timing closure. At 10 MHz, self-repair will take approximately 1 ms to execute for a BISR chain with 5000 bits, which are 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 due to 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. Note that

the routing of the BISR chain is not optimized based on placement unless you use the [BisrSegmentOrderSpecification](#) wrapper and provide a DEF file that specifies memory coordinates. If a DEF file is not provided, the BISR chain order can still be specified to minimize the length of BISR connections and group by power domain.

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.....	198
Verify BISR at the Block Level	200
Verifying Top-Level BISR	203
Creating Multi-Load Scan Patterns with Repairable Memories	216
Generating Your Manufacturing Test Patterns	218

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 “[Verify BISR at the Block Level](#)” section describes these two methods in detail.

Autonomous Self Fuse Box Program	198
FuseBox Access	198

Autonomous Self Fuse Box Program

In the *Autonomous SelfFuseBoxProgram* 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 test bench. 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.

Verify 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 following verification tasks at the design block level in a bottom-up flow:

- Executing Fault-Inserted Memory BIST
- Performing BIRA-to-BISR Capture
 - 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 5-24](#). 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 5-24. 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..... 202

Performing BIRA-to-BISR Capture	202
Executing Post-Repair Memory BIST	202

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 5-24](#). 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 5-24](#). As the name indicates, a full rotation of the BISR chain also is performed after the transfer from the BIRA register to the external BISR register. The rotation allows 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.

Executing Post-Repair Memory BIST

This task re-executes 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 5-25](#) shows a portion of the default PatternsSpecification configuration file generated for the TAP Access mode, while [Figure 5-26](#) 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, will be 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 test benches generated at the top level will 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	203
Autonomous Modes	204
Autonomous Mode for Memory BIST-Only Verification	206
Functional Mode Verification.	208

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 5-25](#), 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 5-25. Example Patterns Specification 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 5-26](#) initializes the BISR chain and calculates its length. The BISR chain length is used in subsequent test steps.

- The second TestStep executes 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 (i.e. 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 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 5-26. 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 5-27. PatternsSpecification Configuration for Autonomous Modes for MemoryBIST-Only Verification

```

Patterns(MemoryBist_P1) {
    ClockPeriods {
        clk_b : 12.0ns;
        clk_a : 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_clk_a_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.

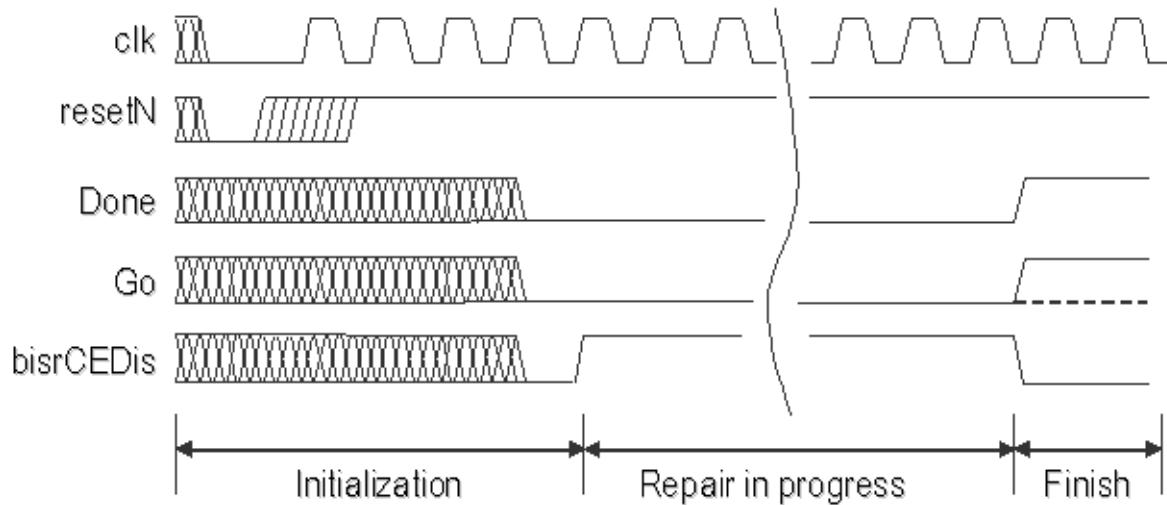
This section describes the protocol between the system logic and the BISR controller, as well as the equations to determine approximate repair time. This section also shows how you can use a [ProcedureStep](#) to partially verify the functional mode of operation in combination with other BISR manufacturing tests.

BISR Protocol (Single Power Domain Group)	208
BISR Protocol (Multiple Power Domain Groups)	209
Repair Time Calculation.....	211
Combining Functional Mode with Manufacturing Tests.....	212

BISR Protocol (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 whether 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 5-28 shows the protocol for a single power domain.

Figure 5-28. 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 protocol is slightly different when multiple power domain groups are present. [Figure 5-29](#) illustrates this protocol for a circuit with 3 groups. Signals `clk` (repair clock), `resetN` (repair enable), `Done`, and `Go` are the same as in [Figure 5-28](#). New signals in [Figure 5-29](#) 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 3 groups. Signal names associated with a group have the same suffix, which 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)`

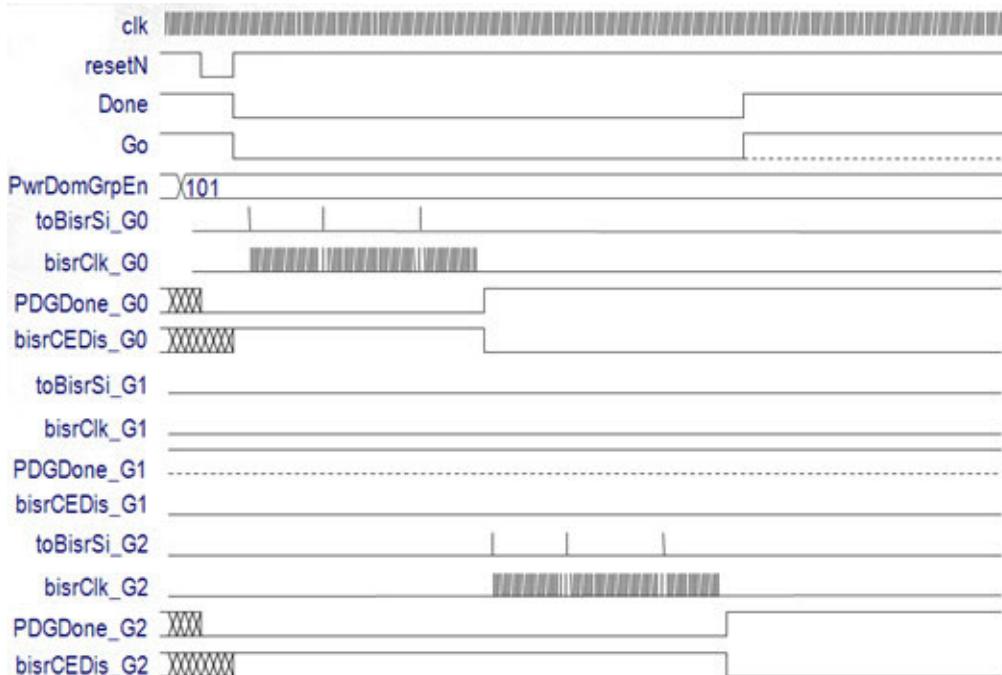
- $G2$ ($_G2$ suffix)

For each group, [Figure 5-29](#) shows four signals:

- $toBisrSi_{_Gx}$ is the serial input of the BISR chain.
- $bisrClk_{_Gx}$ is the clock of the BISR chain.
- $PDGDone_{_Gx}$ ([AdvancedOptions/bisr_done_connection](#) property) is a signal indicating that the memories of group x have been repaired and are ready to use.
- $bisrCEDis_{_Gx}$ ([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 5-29. BISR Controller Protocol in Functional Mode (Multiple Power Domain Groups)



The $PwrDomGrpEn$ input bus determines which memory groups will be 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 will be repaired, whereas the state of group G1 will be 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), which 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:

$$T_{repair_min} = T_{clk} * \text{bisrChainLength}$$

$$T_{repair_max} = T_{repair_min} + (\text{readDuration} * T_{clk} * \text{numberFuses})$$

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 due to 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 reads 32 fuses at a time, a fuse box access is only performed for about 3% of the BISR controller requests. If a fuse box access takes 10 clock cycles, this means that the effective readDuration factor is 2.25 instead of 2. 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 will go high. However, the *Go* output will be 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 allows 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 will call 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 5-39](#):

```
MemoryBisr {
    run_mode : autonomous;
    Controller(chip_rtl_tessent_mbisr_controller_inst) {
        AutonomousOptions {
            operation : power_up_emulation;
        }
    }
}
```

An iProc named init_bisr_chains is automatically created by Tessent Shell MemoryBIST, which can be edited to remove the power-up emulation initialization sequence invoked from the IJTAG network and replaced with a user-defined system-clocked functional sequence. This iProc can then be sourced from a ProcedureStep to initialize the BISR chains.

The file containing the init_bisr_chains iProc is located in the TSDB memory_bisr instrument container, which by default is located in the following folder:

```
tsdb_outdir/instruments/<design_name>_<design_id>_mbisr.instrument
```

The filename is:

```
<design_name>_<design_id>_tessent_mbisr.pdl
```

An example of this file is shown in [Figure 5-30](#). The code highlighted in red is the power-up emulation sequence that is to be removed. The code highlighted in green represents a user-defined functional power-up sequence that is added. The sequence shown sets the resetN port on the BISR controller to 0 and transitions it to 1, which starts the BISR controller and serially loads the BISR chain with repair data contained in the fuse box.

Figure 5-30. BISR Controlled from Functional Inputs

```
iTopProc init_bisr_chains {{bisr_controller_instance {}}} } {
    set bisr_chain_length [get_bisr_chain_length]
    set current_design [get_single_name [get_current_design]]
    if { [sizeof_collection $bisr_controller_instance] } {
        set bisr_ctl_inst_c $bisr_controller_instance
    } else {
        set bisr_ctl_inst_c [get_icl_instances -filter
            {tessent_instrument_type==mentor::memory_bisr &&
            tessent_instrument_subtype==controller} -silent]
    }
    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 master_module_name $ctl_inst]
        iClock ${ctl_inst_name}.clk
        set memory_repair_info_dict
            [write_memory_repair_dictionary -fuse_box_controller_icl_instance
            $ctl_inst_name -return_dict]
        set controller_period_ns
            [get_iclock_option [get_icl_pins clk -of_inst $ctl_inst]
            -period -in ns]
        dict with memory_repair_info_dict {
            set fuse_box_init_cycles [expr
                {ceil(1.0*$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 fuse box controller $ctl_inst_name"
    set iProc ${ctl_mod_name}_Autonomous
    set iProcArgs [dict create]
    foreach argn [get_iproc_argument_list $iProc -of_module
        $current_design]{
        dict set iProcArgs $argn \
        [get_iproc_argument_default $iProc $argn -of_module $current_design]
    }
    dict set iProcArgs controller_instance $ctl_inst_name
    dict set iProcArgs run_length [expr {int(ceil($run_length))}]
    dict set iProcArgs operation power_up_emulation
    iCall ${ctl_mod_name}_Autonomous {*} [dict values $iProcArgs]
```

```

#
# Insert the sequence to start the fuse box controller here.
# This sequence must set the BISR controller resetN port to 0,
# then to 1.
#
iWrite resetN 0b0;
iApply;
iWrite resetN 0b1;
iApply;
#
# Wait for the controller to complete execution
#
iRunLoop $run_length -tck;
}
iApply
}
}

```

An example of a post-repair test pattern that uses a ProcedureStep and iCall to the modified iProc given in [Figure 5-30](#) is provided in [Figure 5-31](#). Note that the iProc requires the ICL instance name of the fuse box controller as its argument. After the BISR chain is initialized from the resetN functional input, the post-repair memory BIST can then be executed on the repaired memory.

Figure 5-31. Post-Repair MemoryBIST with BISR Controlled from Functional Inputs

```

Patterns(MemoryBist_postrepair_P1) {
    ProcedureStep(functional_bisr_load) {
        iCall(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 Multi-Load Scan Patterns with Repairable Memories

Multi-load scan patterns can be used to generate scan patterns through ROM and RAM memories. Before generating multi-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.

The needed 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 multi-load patterns. The examples shown in [Figure 5-32](#) and [Figure 5-33](#) below show how this iProc is utilized to initialize the BISR chains, then create and write out the multi-load patterns.

Figure 5-32. iProc Usage for Physical Block Level with No Fuse Box 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 ram_sequential -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 ram_sequential_serial.v -verilog -serial -replace
write_patterns ram_sequential_parallel.v -verilog -parallel -replace
```

Figure 5-33. iProc Usage for Chip Level with Fuse Box Controller Present

```

set_context patterns -scan_retargeting

# Read design
set_current_design chip

add_core_instances -instance core_inst1/wrapper_i1/blka_i1 \
                  -mode ram_sequential
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_ram_sequential/blka_ram_sequential_stuck.patdb
set_external_capture_options -pll_cycles 5 [lindex [get_timeplate_list] 0]
write_pattern chip_ram_seq_repair_W_blka_i1_parallel.v -parallel -v \
               -replace -param_list {SIM_TOP_NAME TB}
write_pattern chip_ram_seq_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, the “-front” command line option must be specified to ensure that the memory repair procedure is executed at the beginning of the pattern. Not doing so, will result in an error during pattern generation, because in this condition, access to the repair logic will be blocked due to the scan test signals.

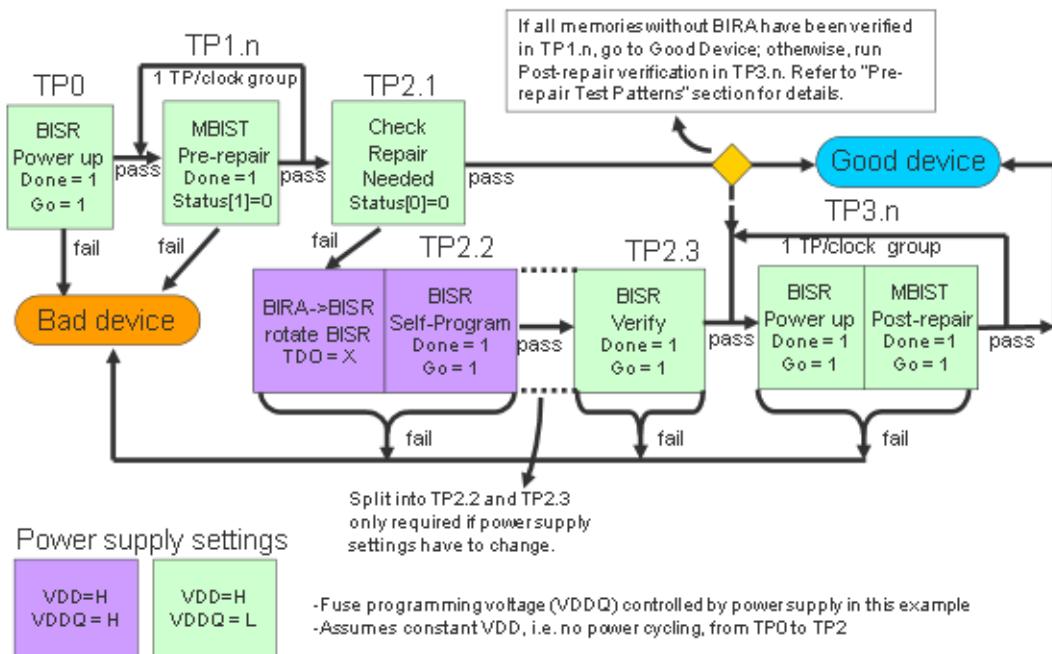
Generating Your Manufacturing Test Patterns

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 5-34](#) 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 5-35](#) to [Figure 5-39](#) 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 5-34. BISR Manufacturing Flow Example



Initialization Test Pattern	219
Pre-Repair Test Patterns	219
Repair Test Patterns	221
Post-Repair Test Patterns	224
Tester Settings Considerations	225
Manufacturing Flow Variations	226

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 5-35. 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 5-8](#). 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 allows 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 execute 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 execute 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, Tesson 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 appears, you must execute 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 executed.

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 5-36. 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_CaptureBiraRotate*) of the Repair group transfers the contents of the built-in repair analysis (BIRA) registers to the BISR registers (*enable_bira_capture: on*), and the contents of the BISR chain is rotated (operation: *rotate_bisr_chain*).
- 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 execute 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 will take approximately 1.3ms to execute 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 executed. 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 5-34](#). Refer to the “[Tester Settings Considerations](#)” section for more details.

The figures below show examples of Repair test patterns.

Figure 5-37. 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 5-38. Example Configuration for TP2_2 Repair Test Patterns

```

TestStep(TP2_2_CaptureBiraRotate) {
    AdvancedOptions {
        split_patterns_file : on;
    }
    MemoryBisr {
        run_mode : autonomous;
        Controller(chip_rtl_tessent_mbisr_controller_inst) {
            AutonomousOptions {
                operation : rotate_bisr_chain;
                enable_bira_capture : on;
            }
        }
    }
}

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 in order 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 will be 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 [224](#)

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 5-39. 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

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 will cause splitting of the test patterns as shown in [Figure 5-34](#). The TP2.2 test pattern including test steps *TP2_2_CaptureBiraRotate* and *TP2_2_SelfFuseBoxProgram* will be executed with the dedicated power supply set at the voltage required to program the fuse box, whereas

TP2_1_CheckRepairNeeded and *TP2_3_VerifyFuseBox* are executed with the voltage set to allow reading the fuse box. The test pattern file (for example, WGL) will contain 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_CaptureBiraRotate*).

Manufacturing Flow Variations

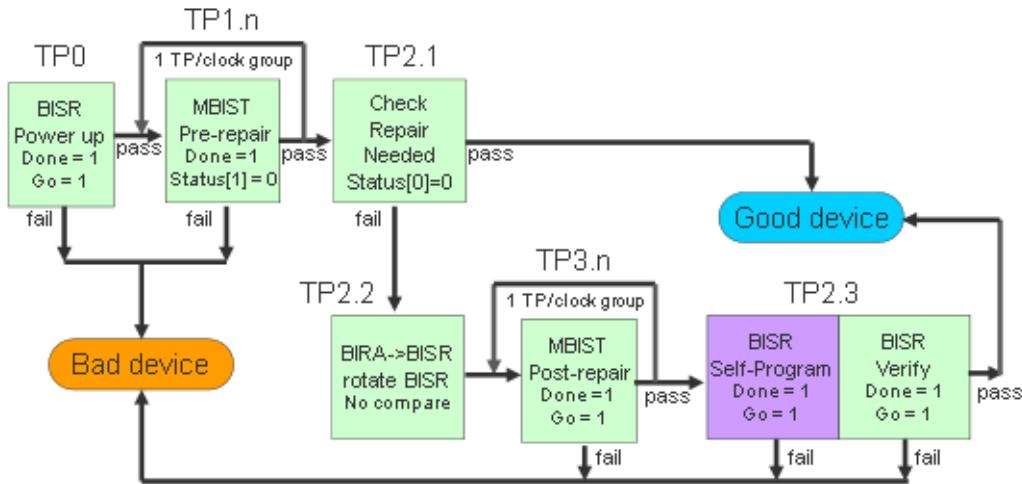
Several other variations of the flow illustrated in [Figure 5-34](#) are possible to optimize yield and/or test time. The section “Testing Repair Solution Before Fuse Programming” describes one variation that allows testing the repair solution prior to fuse programming. The section “Performing Fuse Programming During Any Test Insertion” describes another variation that allows 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 5-40](#), 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.

Figure 5-40. Flow Variation: Testing Repair Solution Before Fuse Programming



Performing Fuse Programming During Any Test Insertion

The test flows of [Figure 5-34](#) and [Figure 5-40](#) 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.

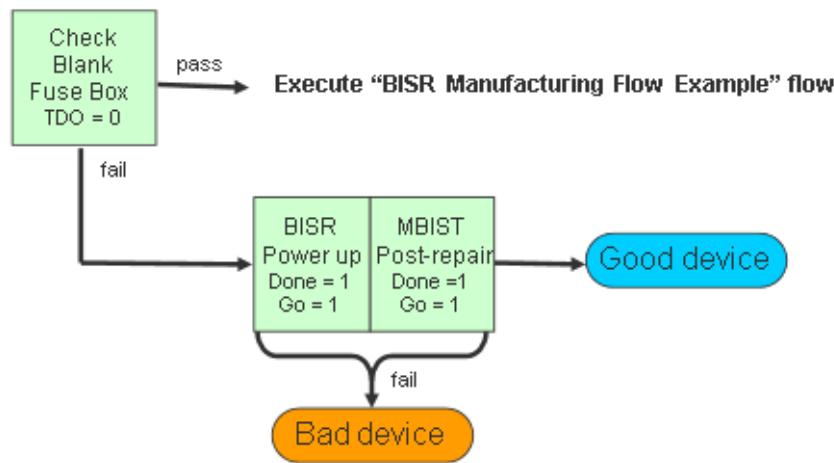
This is done by reading a number of fuses starting at address 0. The number is determined by the largest of the MemoryBISR/Controller/[AdvancedOptions](#)/repair_word_size and zero_counter_bits, and is usually less than 32. The following shows an example of a TestStep wrapper reading the first 32 locations 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 5-34](#) can be executed as shown by the “pass” path in [Figure 5-41](#) below. Otherwise, the Post-Repair test pattern is executed.

Figure 5-41. Flow Variation: Performing Fuse Programming During Any Test Insertion



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 $1 + \text{Log}_2(\text{MaxChainLength})$ 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 5-23](#) and described in the “Single-Chain Case” section.

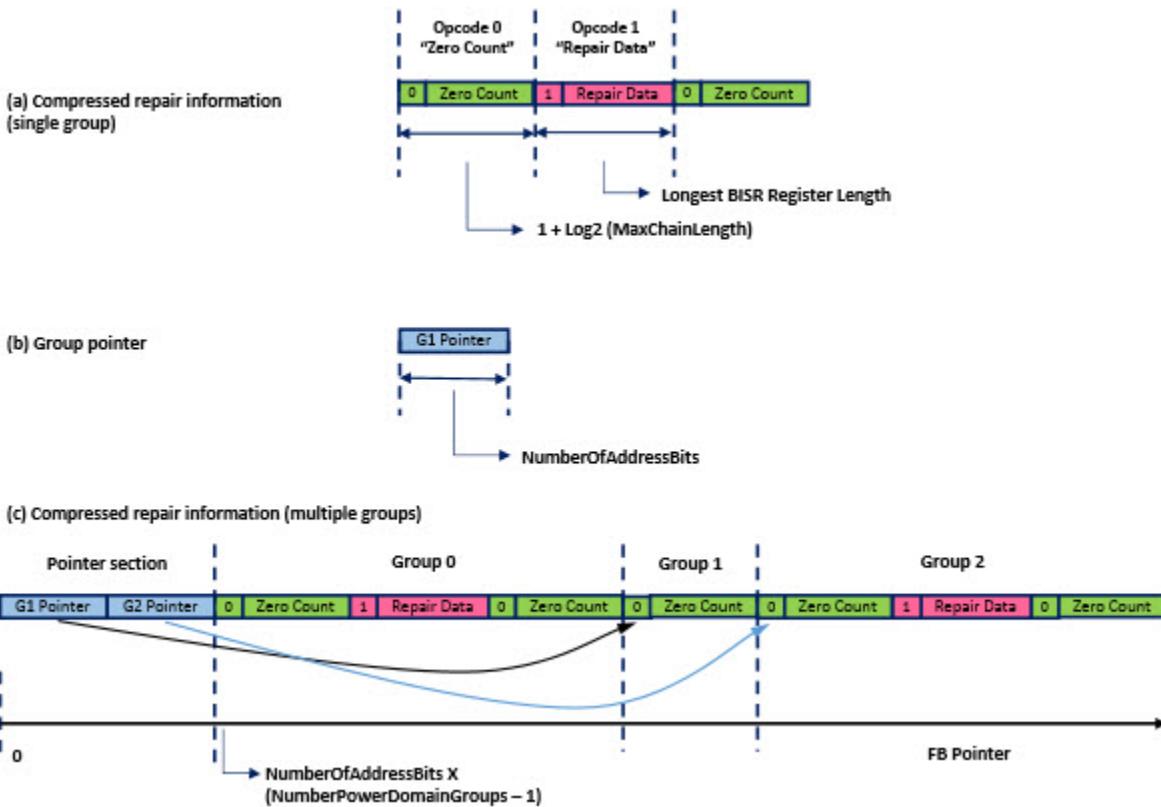
In a *Repair Data* word, the first bit is part of the repair data in addition to being an *Opcode*. *Repair Data* words do not necessarily align with BISR register boundaries because the controller does not know the location of those boundaries. For this reason, a *Repair Data* word can be incomplete when it is the last word in the BISR chain.

[Figure 5-42\(a\)](#) 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 data of each group is compressed as described above. In addition, pointers are stored in the fuse box to find the repair information specific to groups other than the first group. [Figure 5-42\(b\)](#) shows that each pointer requires a number of fuses, which is the number of fuse box address bits. [Figure 5-42\(c\)](#) shows that all pointers are inserted at the beginning of the fuse box starting at address 0.

Figure 5-42. Fuse Box Organization

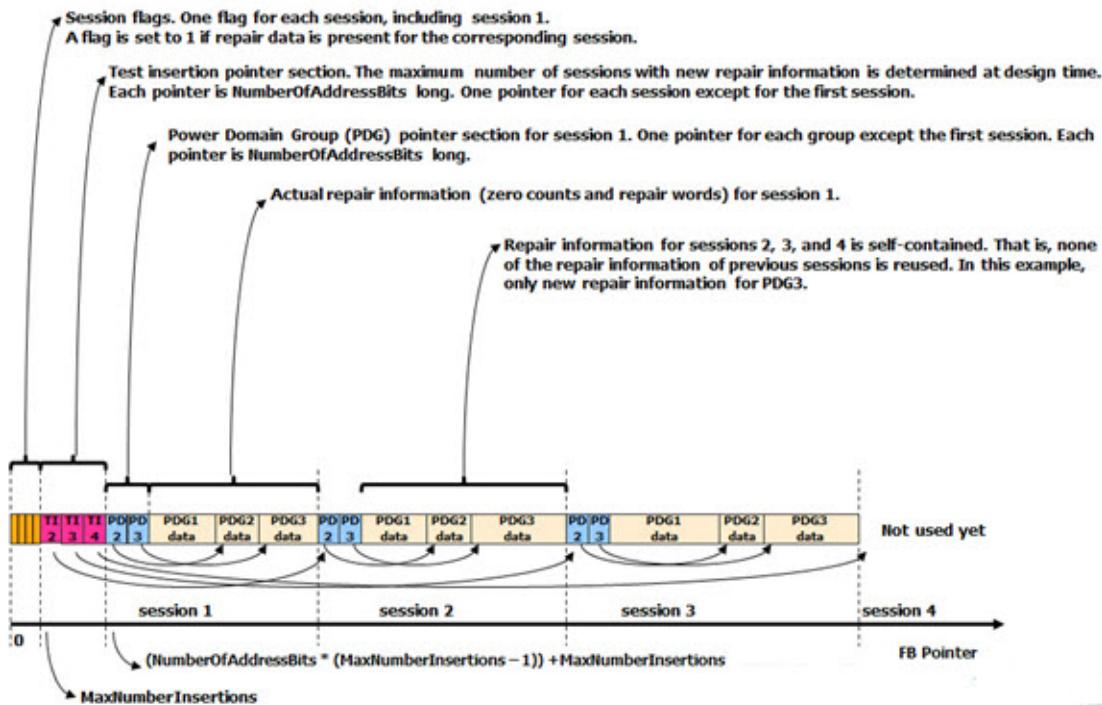


Note

 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 the *CompressBisrChain* script, see the “[“CompressBisrChain Script”](#) section.

When hard incremental repair is used (`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 5-43](#) 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 next block of fuses is a pointer section. Each session has one pointer except for the first session, which is located immediately after the pointer section. The content for each programming session is organized the same way as shown in [Figure 5-42](#). 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 5-43. Fuse Box Organization (`max_fuse_box_programming_sessions > 1`)



CompressBisrChain Script 231

CompressBisrChain Script

The *CompressBisrChain* script includes optimization techniques that reduce the number of fuses when Power Domain Groups (PDGs) with the same BISR chain length are present. In the case of multi-session fuse programming, additional optimization techniques reuse fuses from previous sessions whenever possible.

The 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 will not be 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 will need to be repaired.

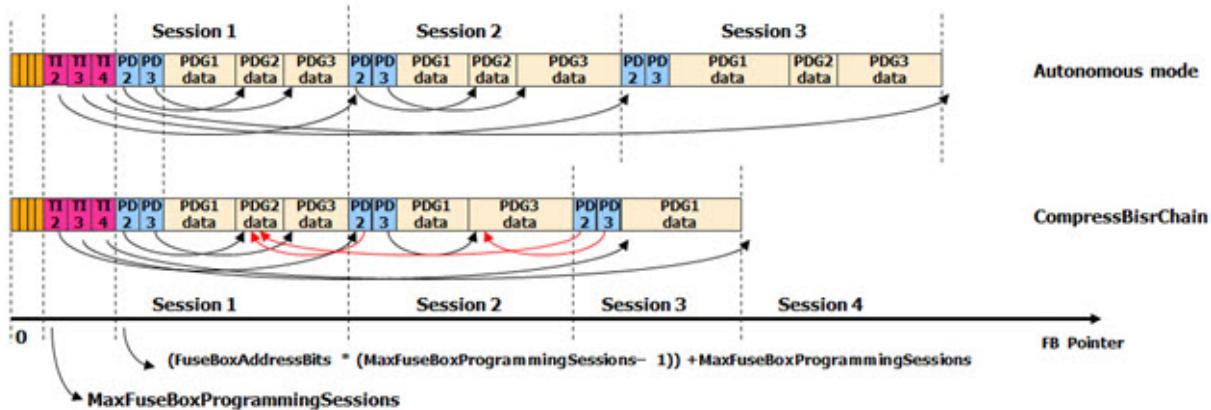
Figure 5-44 compares the [FuseBox Access](#) and [Autonomous Self Fuse Box Program](#) methods for the example shown in Figure 5-43. For Session 1, the two methods give the same result, assuming that all PDGs are of different lengths. In Session 2, reusing the repair data of PDG2 is possible because 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. Due to 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 5-44](#) 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 5-44. 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 5-45](#). 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.

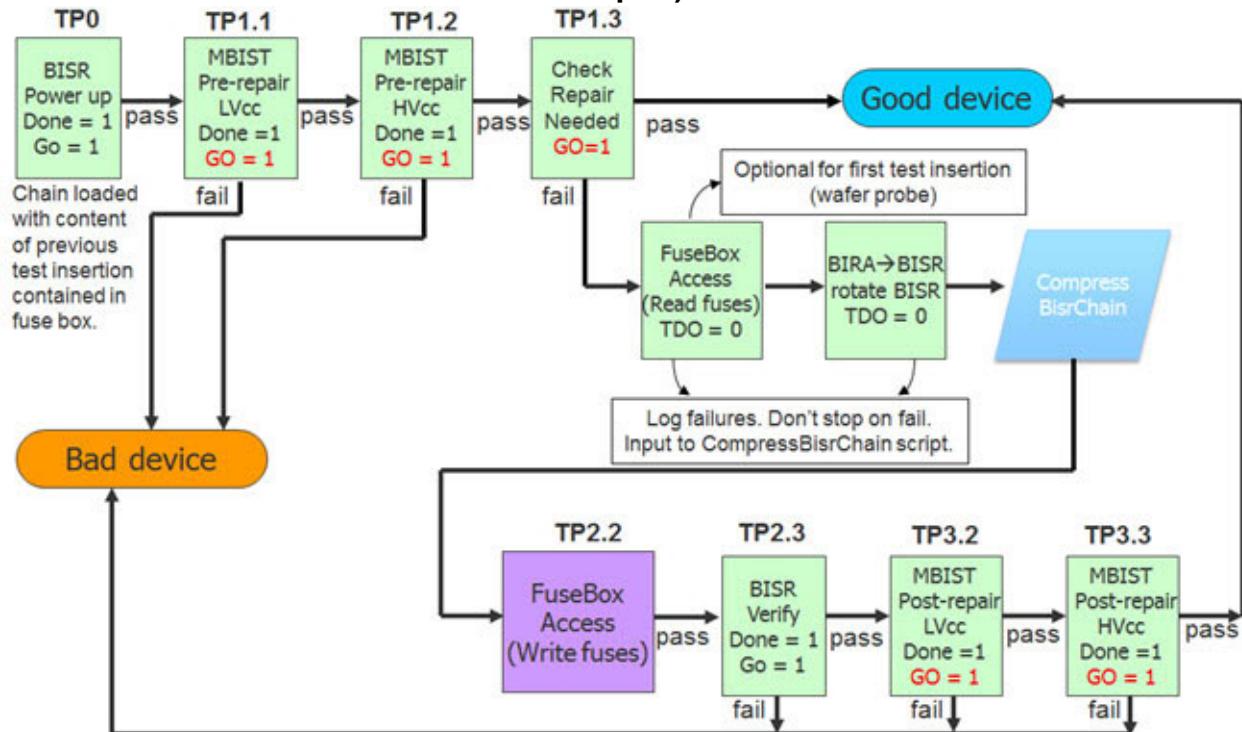
```
TestStep(membisr_CheckRepairNeeded) {
    ...
}
TestStep(membisr_ExtractFuseBoxSessionInfo) {
    MemoryBisr {
        run_mode: fuse_box_access;
        Controller(FB_INST1) {
            FuseBoxAccessOptions {
                operation : read;
                // Restrict read to the first 512 locations 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 5-45. Manufacturing Flow Using CompressBisrChain Script (Incremental Repair)



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	235
BIRA Initialization	237
BIRA Repair Status Bits Checking	238
Absence of Fuse Programming Step	245
Handling of Blocks Without Incremental Repair Capability	245
Considerations Specific to Hard Incremental Repair	246

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. 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 are stored uncompressed in the fuse box. Subsequent repair solutions can be written directly into the fuse box without affecting the

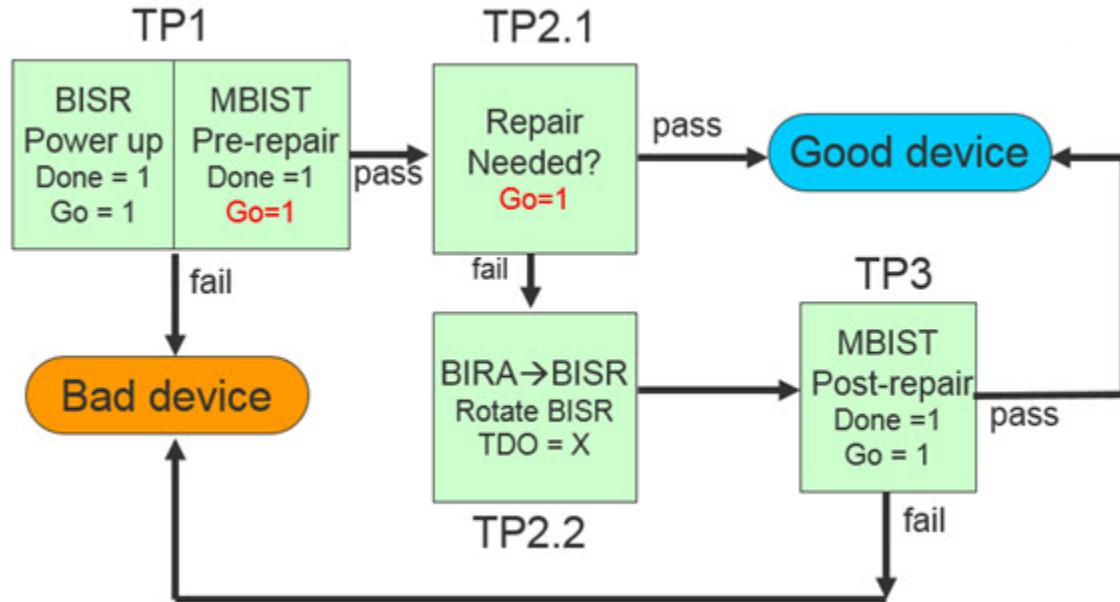
repair solution from previous insertions. This allows an unlimited number of [self_fuse_box_program](#) autonomous mode programming sessions. The number of fuses required for this use must be equal to, or greater than the total number of bits in the BISR chain. Disabling the fuse box compression hardware 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.

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 5-46](#) illustrates the flow for soft incremental repair, which is the same for both applications. To simplify [Figure 5-46](#), 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.

Figure 5-46. 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.

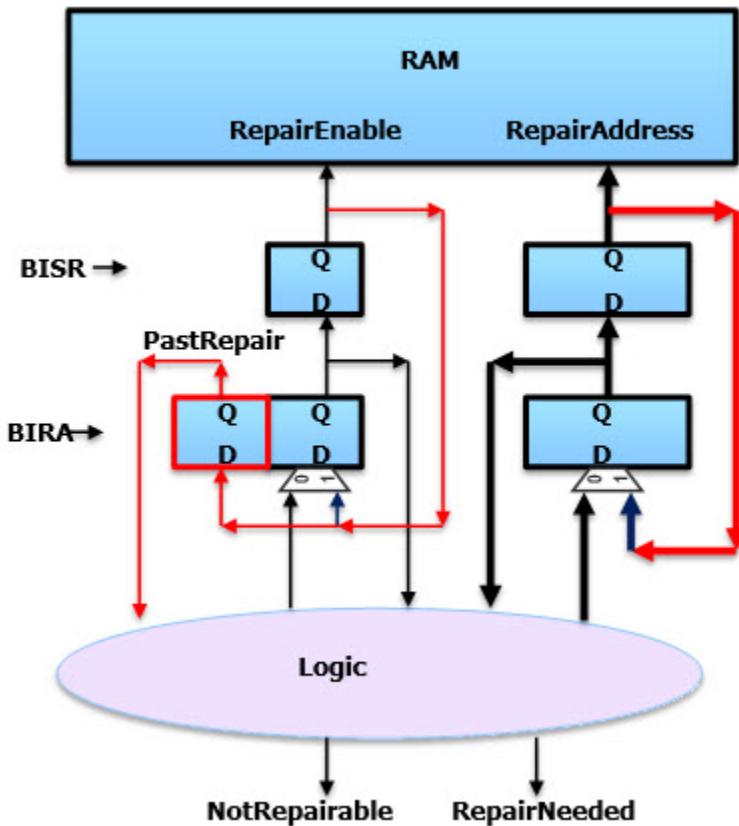
BIRA Initialization

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 executed and BIRA enabled, the BIRA fuse registers are loaded with the content of the corresponding BISR registers.

[Figure 5-47](#) 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 5-47. BISR-to-BIRA Transfer Connection



The mux at the input of the BIRA registers holding the repair enable and repair address allow 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 will appear to be covered and not need new repairs, and the MemoryBIST post-repair pattern will not be executed. 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 Tesson Shell MemoryBIST. The circuit shown on [Figure 5-47](#) is applicable to memories having a repair enable input. A slightly different circuit is used for memories that do not have such an input. Tesson 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 due to repair in previous test insertions.

[Figure 5-48](#) 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 5-48. Detection of Repair Needed Condition

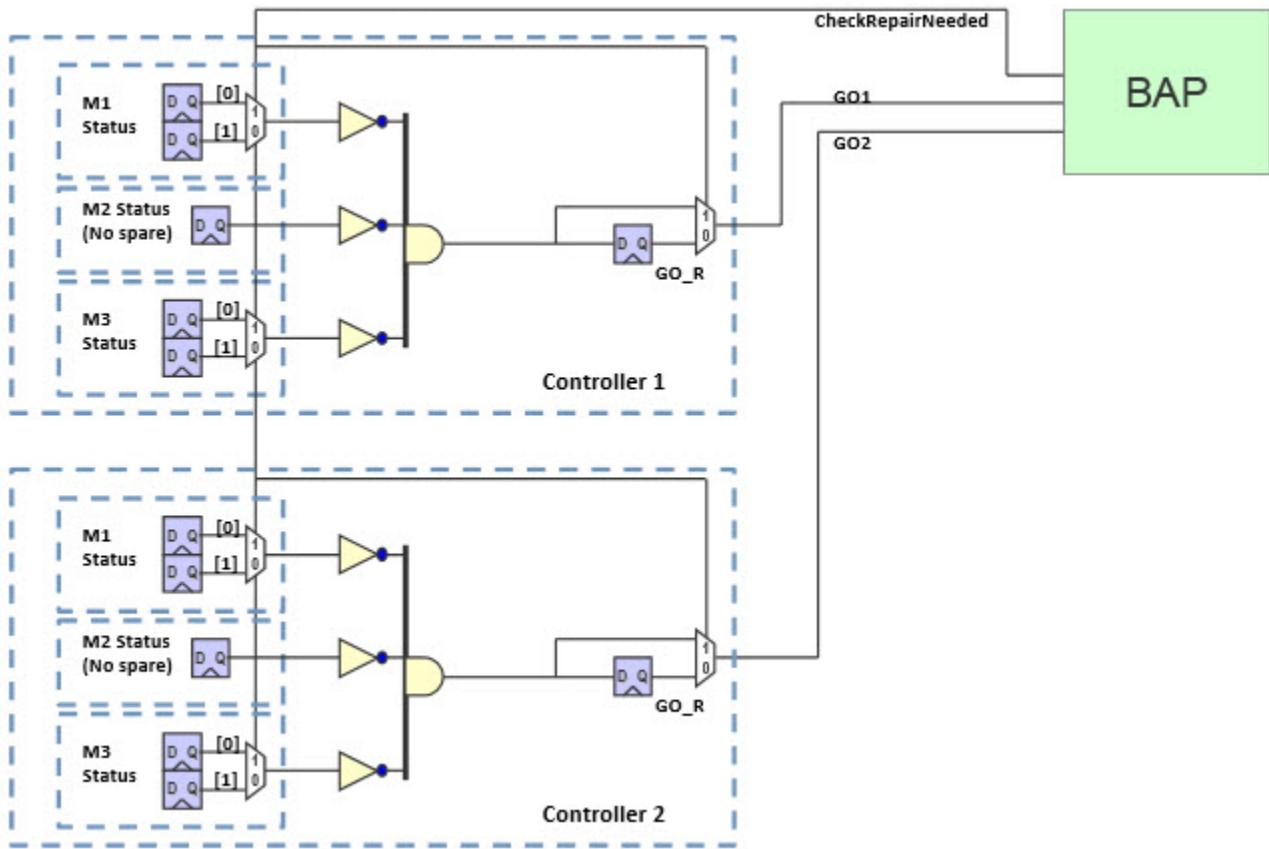


Figure 5-48 shows the hardware enabling this mechanism, including the control signal CheckRepairNeeded. The example shows two controllers. Each controller has two memories with spare resources and one memory without. All memories have a repair status register, although the memory without spare resources only has the equivalent of REPAIR_STATUS[1], which indicates a non-repairable status, as shown in Table 5-6 below. This register is set as soon as an error is found for this memory, which 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 REPAIR_STATUS[0] to the controller output is combinational as the GO_R register may not be clocked at the time the check is performed.

Table 5-6. 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 5-34, are summarized in Table 5-7 below.

Table 5-7. 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 5-8 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 5-9) 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 issued 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 5-8. 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	A6	A2 ₀₀	A2 _{0X}

Table 5-8. Action Table for GO_ID (cont.)

compare_go	compare_memory_go	compare_go_id	extract_repair_fuse_map	check_repair_status		
				Off	On	non_repairable
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+A6 ^(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. **Recommended settings for MBIST pre-repair**
5. **Recommended settings for MBIST post-repair**

Table 5-9. 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>).

Table 5-9. Action Descriptions (cont.)

Action	Description
A3 — *	Scan-out SHORT_SETUP; Compare GO_ID_REG[x:0]. Warnings: GO_ID_REGS are not “sticky” (i.e. 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).
A6	Scan-out BIRA_SETUP; Compare REPAIR_STATUS[1:0] 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 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 5-10](#). 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 will correspond to the last strobe value from the last BIST controller step executed. For local comparators with a repairable memory, this will correspond to the last strobe value for the current memory.

A warning is issued 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 5-34](#) 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 5-10. GO_ID Behavior in BIRA Mode

Comparator Type	Repairable Memory		Non-Repairable Memory
	IO Repair	Any Other Repair Method	
Shared	Non-Sticky	Non-Sticky	Non-Sticky
Local	Sticky	Non-Sticky	Sticky

Table 5-11. GO_ID Behavior in Stop-On-Error Mode

Comparator Type	Repairable Memory	Non-Repairable Memory
Shared	Non-Sticky	Non-Sticky
Local	Non-Sticky	Non-Sticky

Table 5-12. GO_ID Behavior in Go/NoGo Mode

Comparator Type	Repairable Memory	Non-Repairable Memory
Shared	Sticky	Sticky
Local	Sticky	Sticky

Table 5-13. 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 5-46](#) 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(RemoteRepairFlow) {
        TestStep (0) {
            ...
        }
        TestStep (CheckRepairNeeded) {
            AdvancedOptions {
                split_patterns_file : on;
            }
            MemoryBist {
                run_mode : check_repair_needed;

                // Only controllers inside this TestStep will have their
                // repair status bit checked. All others will be 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 due to small voltage and/or 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.

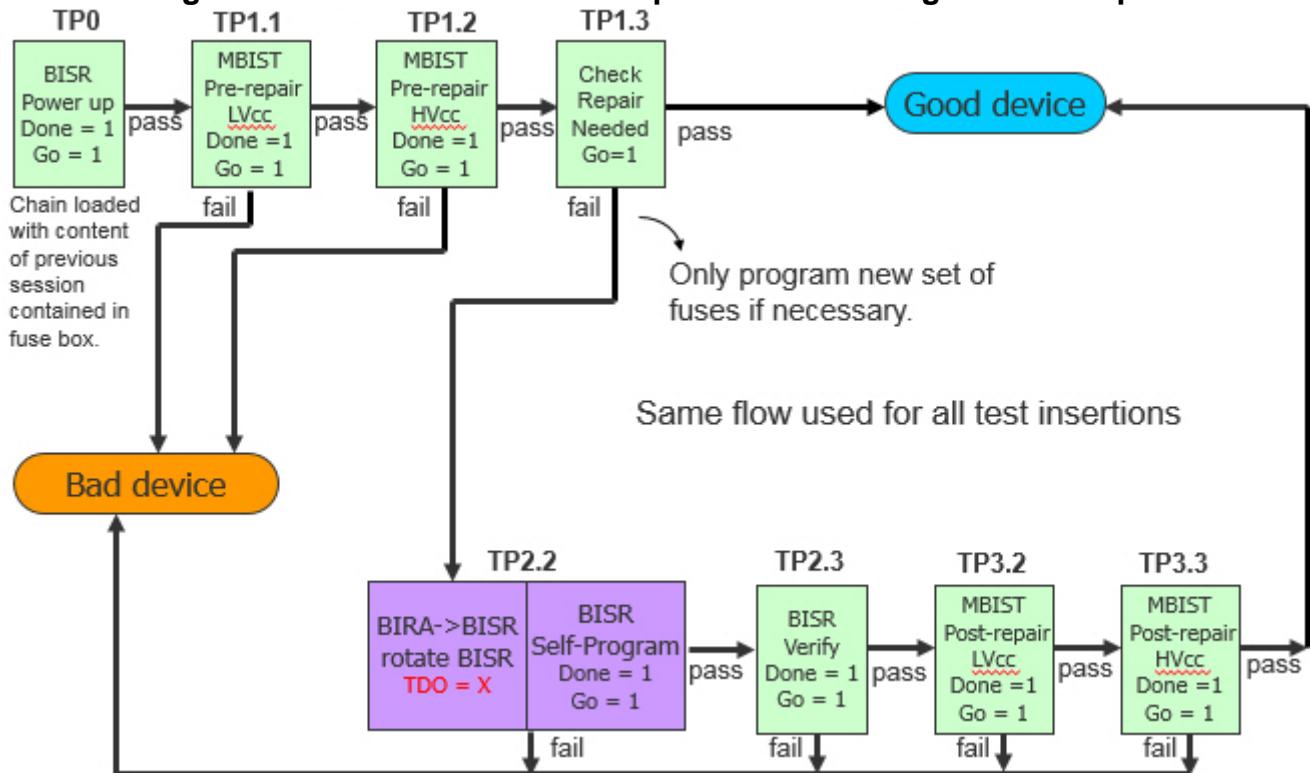
The differences are as follows:

- 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 before. 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. Another difference is that it is necessary to use the method described in the “[BIRA Repair Status Bits Checking](#)” section to detect the “Repair Needed” condition.

In the example shown in [Figure 5-49](#), 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 executed 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 5-49. Hard Incremental Repair Manufacturing Flow Example



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 will outline the requirements and steps to incorporate the sharing of BIRA and BISR hardware among multiple memories to realize these benefits.

Repair Sharing Overview	248
Repair Sharing Conditions	248
Implementing Repair Sharing	252
Creating and Modifying Repair Share Groups	252
BISR Segment Order File	256
BISR Instance Location	257

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. Tesson 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 since 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.

In order 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, which can be part of a single or multiple memories. Tesson MemoryBIST will analyze 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 in order to share repair logic among memories in a repair group. Tesson MemoryBIST performs rule checks to ensure the compatibility of all memories that are sharing the same BIRA and BISR modules.

The following conditions are to be met for memories to share repair logic:

- **Memories must share GO_ID comparators**

Memories that are targeted to share repair logic must also share comparators. This implies that the comparators are located in the controller with the Controller/[Step](#)/comparator_location:shared_in_controller property setting. Memories that use local comparators are limited to sharing repair logic between its own segments.

- **Memories must use the same repair type**

Allowed repair types are Row/word-only, Column/IO-only and Row&Column.

- **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 takes must meet the user specification**

- 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 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 5-50](#) 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 5-50. 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.

The following example illustrates this situation. Three memories, using row repair as outlined in [Figure 5-51](#), have 16, 12 and 8 rows respectively. If a defect is found in row 12 to 15 of Mem1, Mem2 will receive 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 will depend on the memory sizes, so that the repair groups are balanced in terms of their total number of bits.

Figure 5-51. 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 desired memory instances by following the methods outlined in this section.

Creating and Modifying Repair Share Groups	252
BISR Segment Order File	256
BISR Instance Location.....	257

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 allows 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 5-52](#) below. Since 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” will override the `DefaultsSpecification RepairOptions repair_sharing` setting for the specified memories and memory clusters.

Figure 5-52. 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 5-53](#) 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 5-53. 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 since 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 will be 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 5-54](#) is created that will show 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 desired changes are made, `process_dft_specification` can be run.

Figure 5-54. 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];
                    // Default value = 100; Default unit = kilobits
                repair_group_scope : (controller) | logical_memory
                    | physical_memory;
            }
        }
    }
}

```

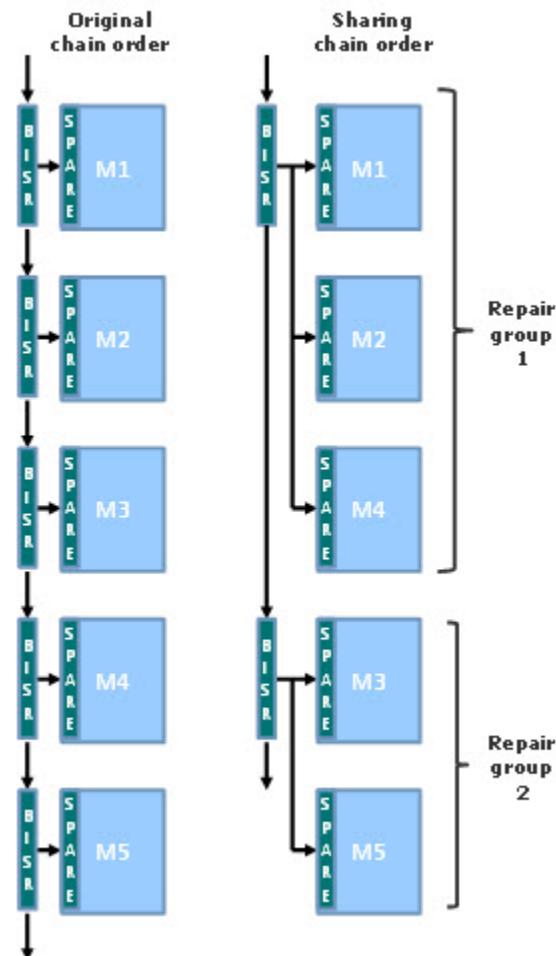
BISR Segment Order File

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 5-55](#) 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 5-55. BISR Chain Ordering with Repair Sharing

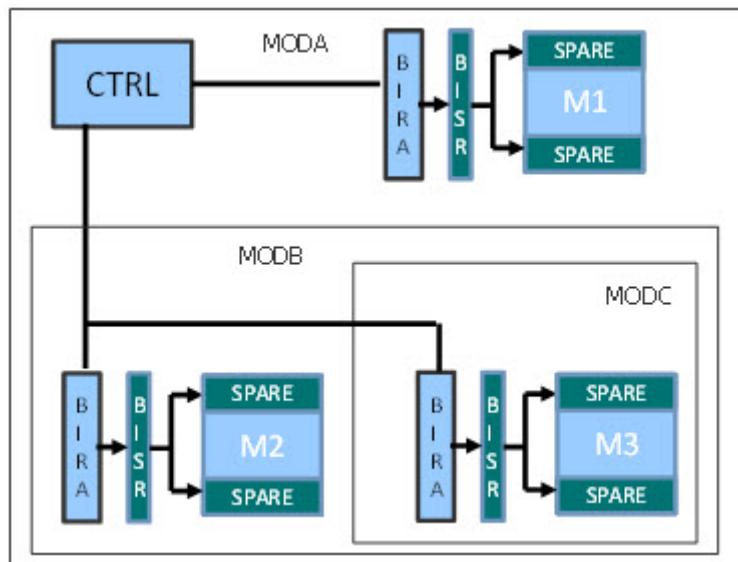


BISR Instance Location

The BISR registers will be 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 5-56](#), 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.

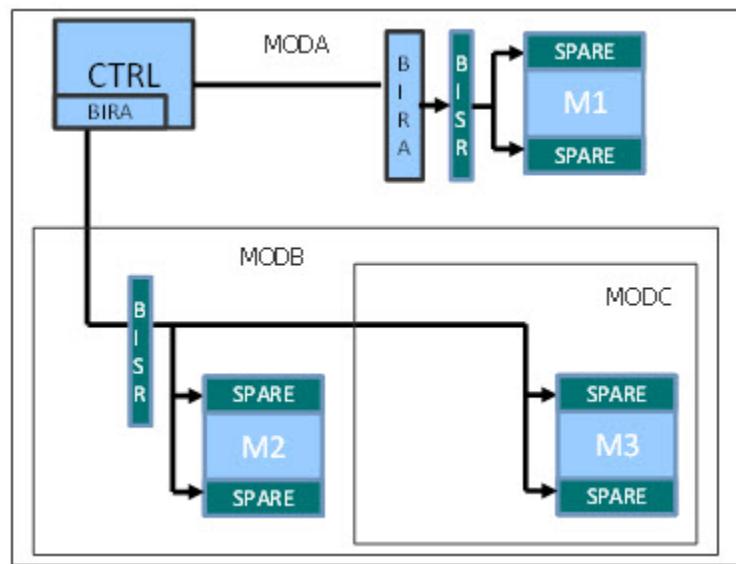
Figure 5-56. BISR Register Location with Local Comparators



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.

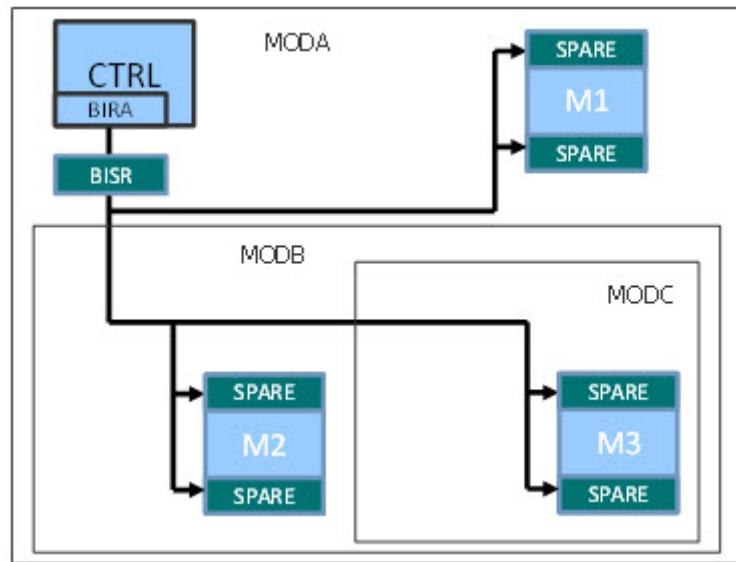
In the example shown in [Figure 5-57](#), M2 and M3 share comparators that are located in the controller along with the BIRA logic. The shared BISR register is located in MODB which is the common ancestor of the two memories. For this example, M1 has its own repair group or is excluded from repair sharing due to having local comparators.

Figure 5-57. BISR Register Location Example with Two Repair Groups



The example shown in [Figure 5-58](#) 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 5-58. 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 will describe the architectural modifications created as well as how to implement fast BISR loading.

Fast BISR Loading Overview	259
Fast BISR Architecture	259
Implementing Fast BISR Loading	264
Fuse Box Interface	264
DFT Insertion	265
Pattern Generation	266

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 allows a significant speed improvement when powering up a device or power region. Only repairable memories with parallel repair interfaces are allowed to be used with fast BISR loading implementations. Repairable memories with serial repair interfaces are not allowed.

The fuse box must be large enough to allow storing the repair information for the entire chip. It is not necessary to further increase the fuse box size if [Incremental Repair](#) is used. Since the entire fuse box content is driven from fuse box interface ports, a compatible interface defined by the eFuse macro must be used. Mentor Graphics provides compatible fuse box interfaces for TSMC technologies. Others must be provided by the user.

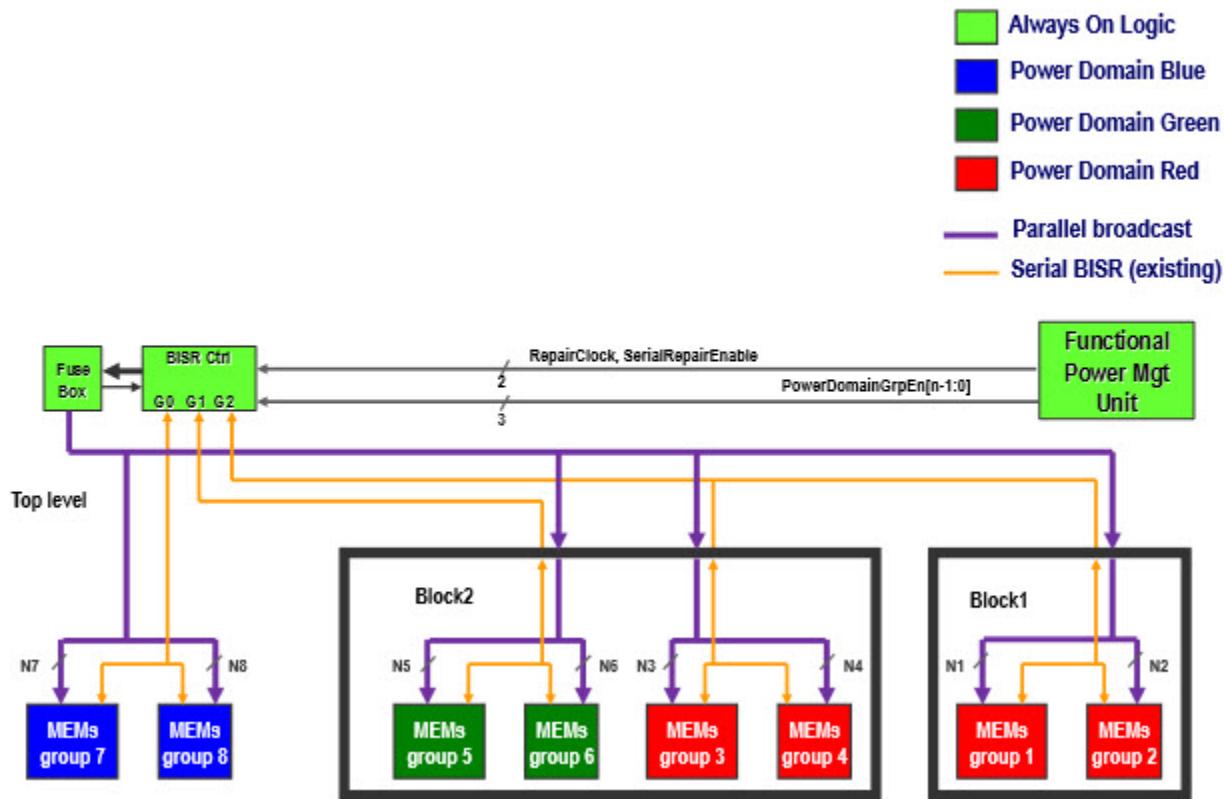
Chips that incorporate large numbers of memories may require implementing [Repair Sharing](#) to reduce the number of unique memory repair inputs and reduce routing congestion.

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.

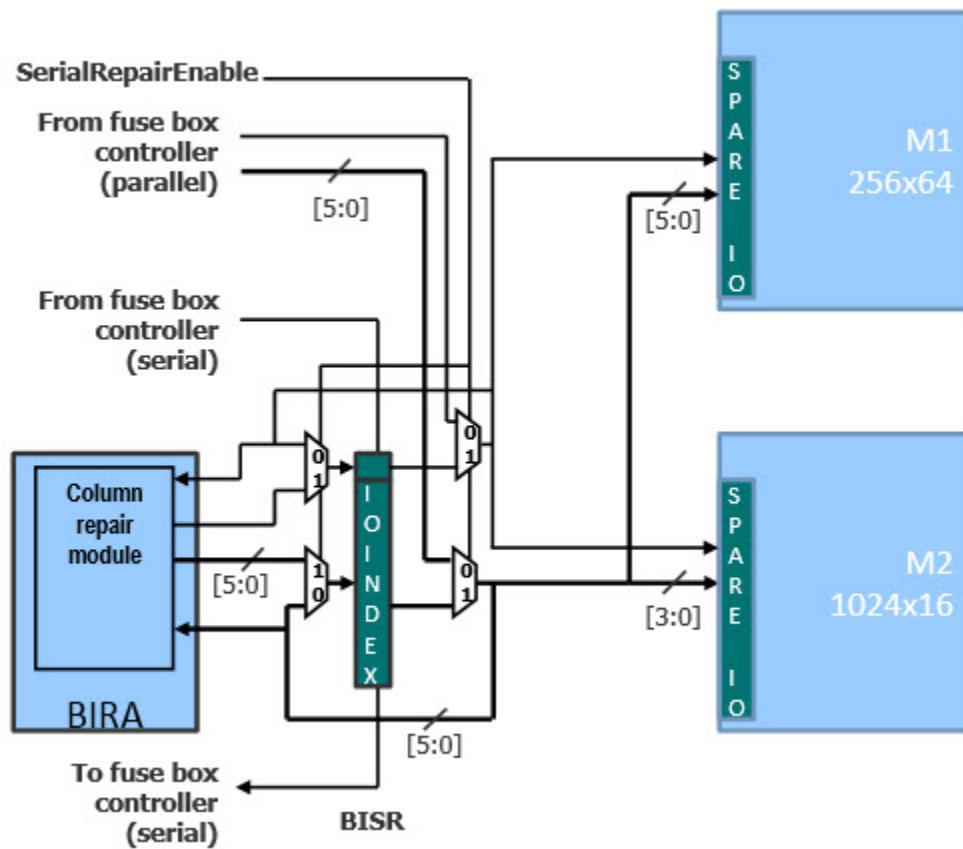
After implementation, both parallel and serial repair buses will be in place, with the parallel bus used to quickly load the BISR registers during power-up and the serial interface used for [Autonomous](#) mode BISR chain rotation operations.

Figure 5-59. 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 5-60](#). These multiplexers allow 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 5-60. Parallel Load Multiplexers



The second set of multiplexers, located between the BIRA module and the BISR register, allows 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 fuse box controller if it is present. If the fuse box controller is not yet present in the design, this signal will be 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 in order to successfully implement fast BISR loading. The combinations listed in [Table 5-14](#) identify both supported and unsupported cases, as well as options available to the user for design variations.

Table 5-14. 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 desired, 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: <ul style="list-style-type: none"> 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. The user has the option to disable both the parallel and serial inputs of the child block if desired.
Serial	Parallel Serial	Unsupported case. The user has two options: <ul style="list-style-type: none"> 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 5-14](#) are provided in [Table 5-15](#) below.

Table 5-15. 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 5-15. 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	264
DFT Insertion	265
Pattern Generation	266

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 allow 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 Tesson 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 will be 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 5-15](#).

```
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 5-60](#). 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 will 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 fuse box controller and the raw uncompressed repair data is stored inside the fuse box. A test data register (TDR) is also added in the fuse box controller to drive the SerialRepairEnable output port. Child blocks are processed as outlined in [Table 5-14](#) and parallel load multiplexers are added to compatible blocks that contain one or more repairable memories with parallel repair inputs.

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 5-60](#) is set to “0”. This will allow 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 will 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 will allow 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 will also be captured inside the BIRA registers at the beginning of the shift cycle.

Generally, there is no fuse box 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 user and consists of adding extra TestSteps that typically involve activating a [FuseBoxAccess](#) to initiate the fuse box to read buffer transfer.

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 *memory cluster* or *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 memory cluster module and 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.

Limitations	270
Applying Memory BIST to a MemoryCluster	270
Shared Bus Support Features.....	273
Shared Bus Interface MemoryBIST Implementation Flow.....	275
Library Requirements	276
Design Loading	279
Specify and Verify DFT Requirements	279
Create DFT Specification	280
Process DFT Specification	282
Extract ICL	284
Create Patterns Specification	284
Process Patterns Specification	286
Run and Check Test Bench Simulations.....	288
Test Logic Synthesis.....	288
Handling MemoryCluster Modules with Repairable Memories.....	289
BIRA and BISR Generation for a Memory Cluster Module	289
Design and Library File Prerequisites for BIRA and BISR	291

Limitations

When implementing Tessent MemoryBIST with shared bus memory interfaces, the following limitations apply:

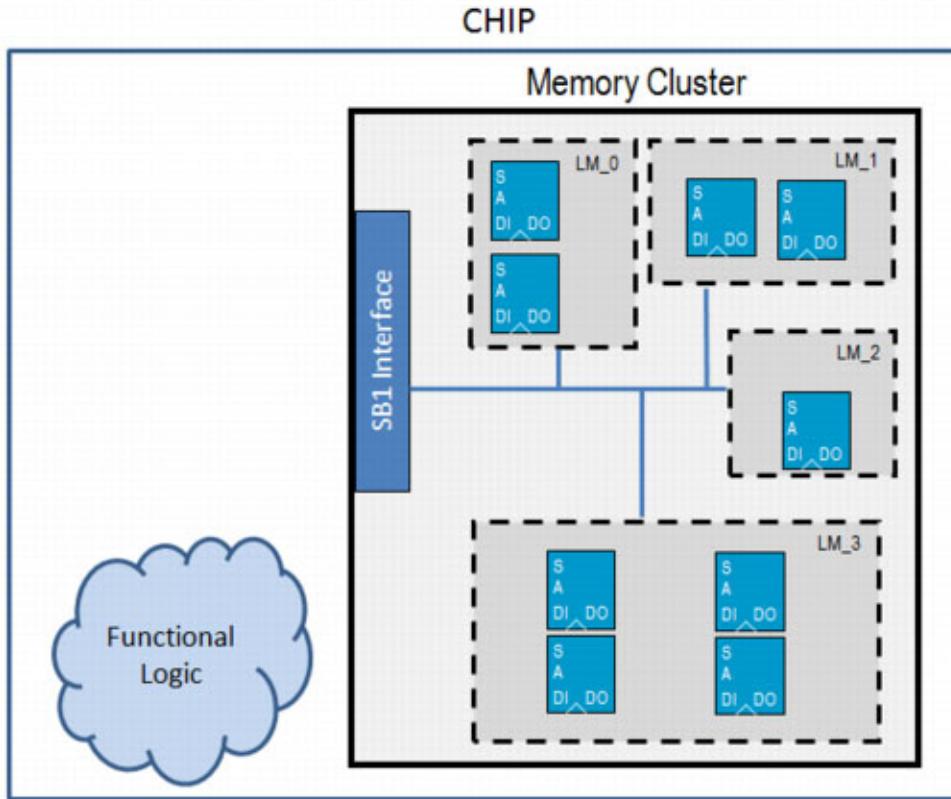
- Shared bus interfaces cannot be driven in parallel by the same memory BIST controller, even if the shared bus interfaces are identical in terms of function.
- Memory cluster modules with identical logical memories that are accessed concurrently through the shared bus and also have common data inputs are not currently supported.
- 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 memory cluster, any memory bypass or scan observation logic required for scan test must be integrated directly inside the memory cluster.
- For hardware optimization reasons, only shared comparators are supported for memory clusters. The comparator circuit is instantiated in the memory BIST controller so that it can be shared among logical memories.
- Only 1R1W and 1RW memories are supported with the share bus feature.

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 6-1](#).

Figure 6-1. Example of CHIP with 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 memory cluster module with a single shared bus interface, a 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 memory cluster module.

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.

Figure 6-2. Composition of Logical Memory LM_1

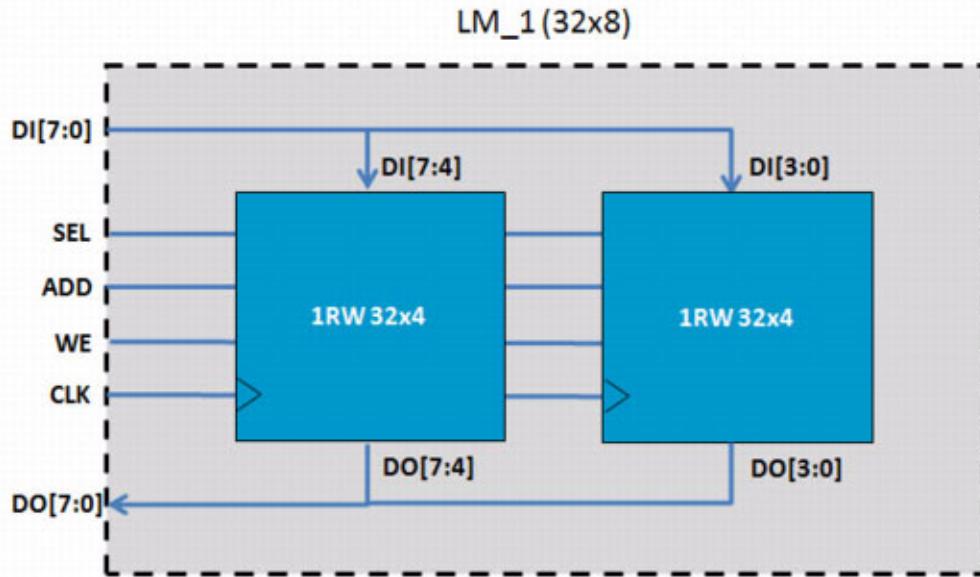
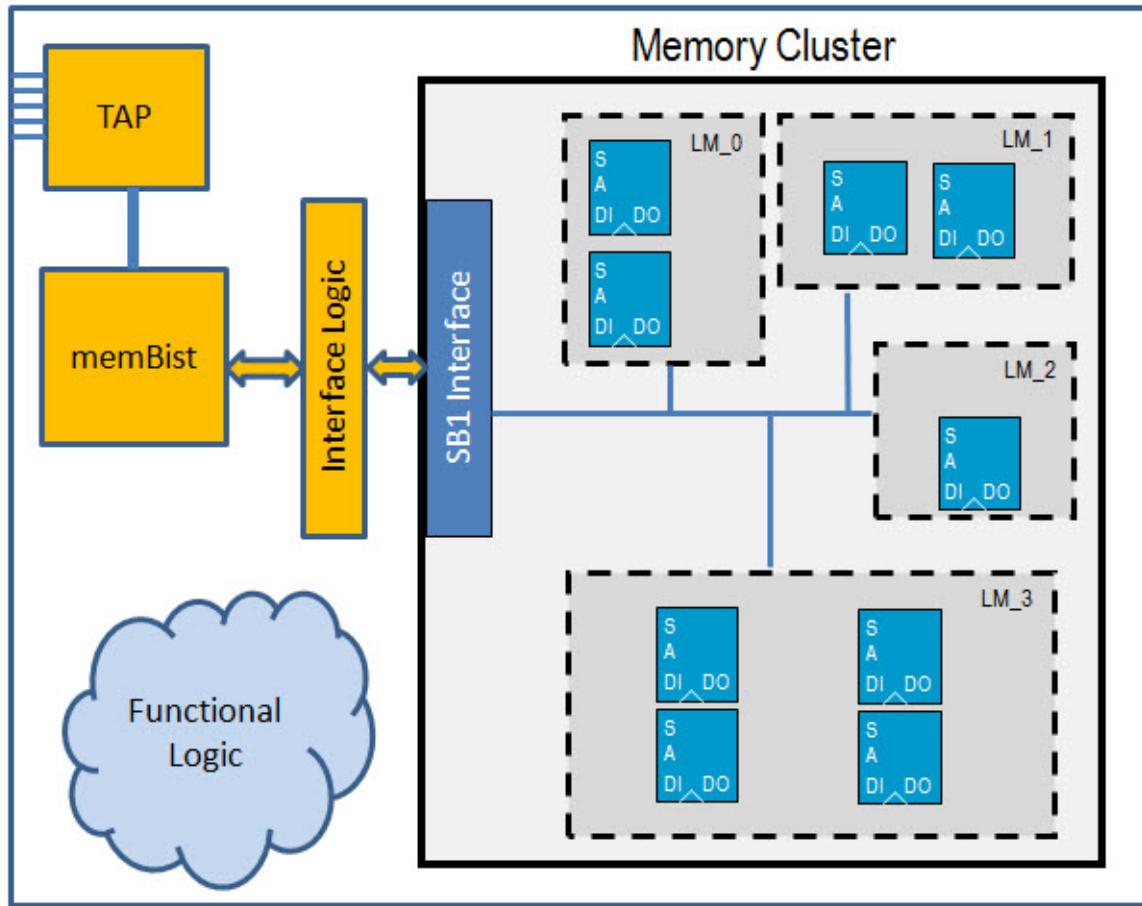


Figure 6-2 shows the composition of logical memory LM_1. The logical memory receives and drives 8 bits of the data channel on the shared bus interface. Two physical memories are instantiated in a horizontal stacking configuration. The 4-bit data ports of the physical memories combine to form the 8-bit datapath of LM_1.

During memory BIST planning phase, one dedicated memory BIST controller is assigned per memory cluster module. The memory BIST controller and interface logic are instantiated at the same level as the memory cluster module as illustrated in Figure 6-3.

Figure 6-3. Memory Cluster Module After Embedded Test Insertion



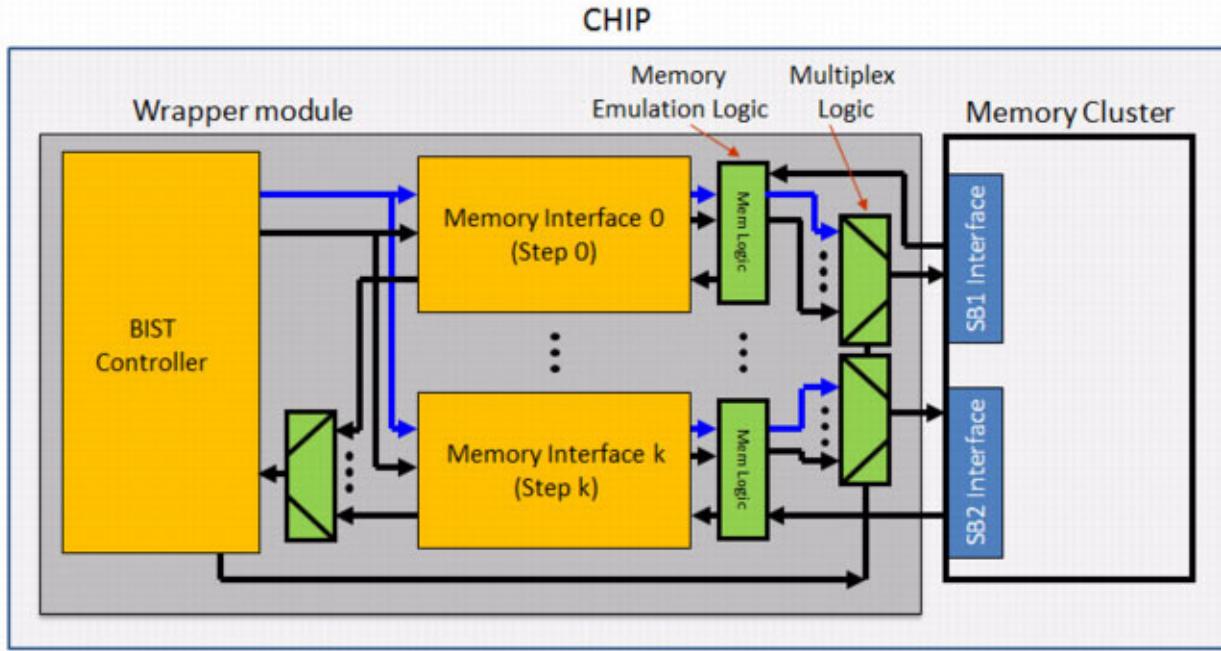
If the design contains standard memories, a different memory BIST controller is assigned to test these memories. A memory BIST controller that is assigned to a memory cluster module cannot be used to test memories outside the memory cluster. Multiple memory clusters can be instantiated inside the design.

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.

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 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 execute 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 allows 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	276
Design Loading	279
Specify and Verify DFT Requirements	279
Create DFT Specification	280
Process DFT Specification	282
Extract ICL	284
Create Patterns Specification	284
Process Patterns Specification	286
Run and Check Test Bench Simulations	288
Test Logic Synthesis	288

Library Requirements

Three memory library files are required:

- Memory Cluster Tessent Core Description
 - Associated with each memory cluster module
 - Describes shared bus interface pins
 - Lists all logical memories contained within the memory cluster module
 - Describes pin mappings between logical memories and the shared bus interface
- Logical Memory Tessent Core Description
 - Associated with each logical memory in the memory cluster module
 - Created or modified by the user based on physical memories used
 - References library files of individual physical memories making up the logical memory
- Physical Memory Tessent Core Description
 - Associated with each physical memory
 - Used in the standard Tessent MemoryBIST flow
 - Does not need modifications if generated by a memory compiler

Memory Cluster Core Library File [277](#)

Logical Memory Core Library File [278](#)

Physical Memory Core Library File [279](#)

Memory Cluster Core Library File

The library file syntax for memory cluster modules is shown in the figure below:

Figure 6-5. Memory Cluster Core Library File 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;
            }
            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 {
                    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 memory cluster, for the 1RW memory, you specify the *LogicalMemoryAddress[]: InterfaceAddress[]* mapping in the *PinMappings* wrapper of the memory cluster library file. The Memory BIST controller drives both the read and write address ports, allowing the 1RW address port to be attached to either the read or write address port of the memory cluster.

Logical Memory Core Library File

The logical memory library file uses a superset of the memory core library syntax, specifically the addition of MemoryGroupAddressDecoding and PhysicalToLogicalMapping wrappers.

Figure 6-6 shows the library file syntax for logical memories.

Figure 6-6. Logical Memory Core Library File Syntax

```
Core(core_name) {
    Memory {
        Algorithm: algo_name;
        Port(port_name) {
            Direction: InOut | (Input) | Output;
            Polarity: (ActiveHigh) | ActiveLow;
            Function: function_type;
        }
        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 {
                PhysicalMemoryDataInput[<indexList>]:
                    LogicalMemoryDataInput[<indexList>];
                PhysicalMemoryDataOutput[<indexList>]:
                    LogicalMemoryDataOutput[<indexList>];
                PhysicalMemoryAddress[<indexList>]:
                    LogicalMemoryAddress[<indexList>];
                PhysicalMemoryWriteAddress[<indexList>]:
                    LogicalMemoryWriteAddress[<indexList>];
                PhysicalMemoryReadAddress[<indexList>]:
                    LogicalMemoryReadAddress[<indexList>];
                PhysicalMemoryGroupWriteEnable[<indexList>]:
                    LogicalMemoryGroupWriteEnable[<indexList>];
            }
        }
    }
}
```

Physical Memory Core Library File

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.

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 and/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.

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:

- At least one [PhysicalToLogicalMapping](#) wrapper exists in the logical memory core library that defines the relationship between physical and logical memory ports,
- 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 to that 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 your 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 id memory id id instance
// code code
// -----
// 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 in the manner as standard memory BIST flow by finding all modules (both Tessent instruments and non-Mentor Graphics 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 “Using DFTVisualizer” 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 will be created using process_patterns_specification. You can edit or configure the default patterns specification to generate the desired patterns specification.

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;
                }
            }
        }
    }
}
}

```

Note: There is reference to an iCall to initialize the cluster module. This will be 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 will default 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 memory cluster module implements interface ports that use the *InterfaceReset* or *BistOn* port functions. The memory cluster initialization sequence is driven by TDR bits and is executed before launching the memory BIST controller. The initialization sequence is referenced in the PatterensSpecification/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 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 Test Bench 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.....	289
Design and Library File Prerequisites for BIRA and BISR	291

BIRA and BISR Generation for a Memory Cluster Module

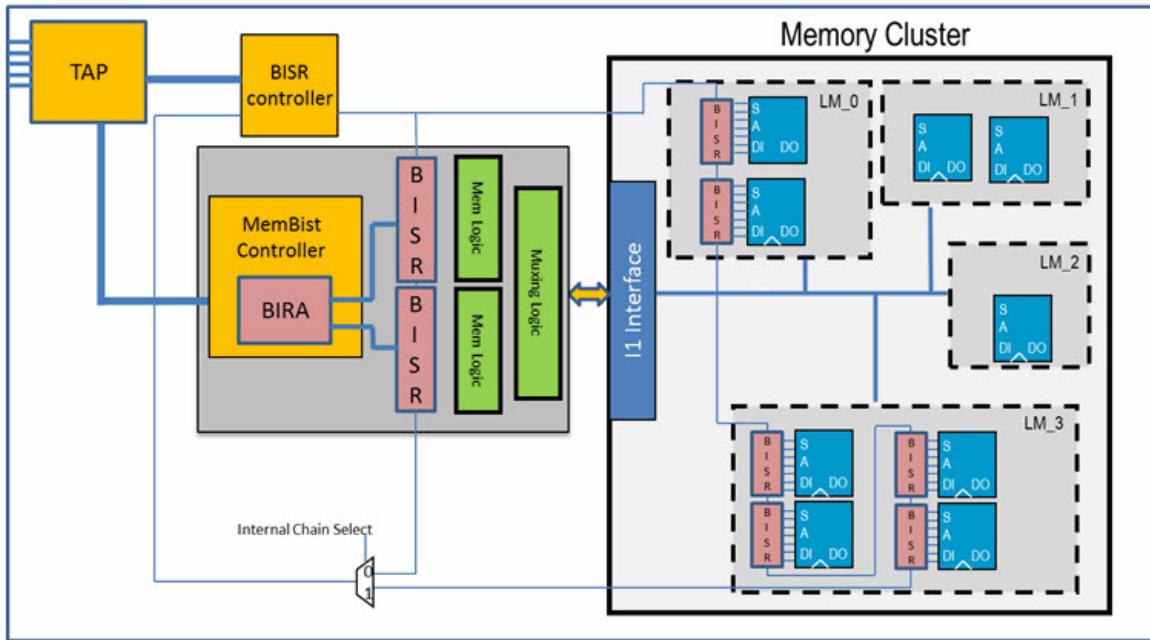
Tessent MemoryBIST is able to insert the BIRA and BISR logic for memories accessed through a shared bus interface. By default, BIRA logic is placed outside the memory cluster module and the BISR logic is placed inside the memory cluster module near the memory to be repaired.

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, allowing direct access at the cluster modules level.

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 6-7](#) provides an overview of a design after BIRA and BISR insertion. The BIRA and BISR logic are shown in salmon.

Figure 6-7. Design Overview of a Memory Cluster Module with BISR

CHIP



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 fuse box 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 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	291
Memory Cluster Library File Preparation for BIRA and BISR.....	291

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, allowing direct access of the physical memory at the cluster modules level.

In order for the `memory_access_level` to work correctly for physical memories with repair, the `MemoryInstanceName` property needs to be specified in the `PhysicalToLogicalMapping` memory wrapper in the logical memory core library.

Chapter 7

Using Tesson User-Defined Algorithms

This chapter describes how to implement custom user-defined algorithms (UDAs) under Tesson Shell MemoryBIST flow.

Usage Context	293
Overview & Terminology	293
Defining a Custom Algorithm	295
Example Algorithm: March C- Algorithm	296
Coding the Algorithm	297
Making a UDA Available to TS-MBIST	303
Selecting a UDA for MBIST Execution	304

Usage Context

Tesson MemoryBIST is provided with a library of predefined memory test algorithms generically referred to as the Tesson Library Algorithms. Memory core description files then explicitly point to those algorithms, so they get implemented during memory BIST insertion.

It is however possible to implement additional algorithms which are not part of the Tesson library; these are typically called custom or User-Defined Algorithms (UDAs). This chapter explains how to implement them using a Tesson Shell flow.

Tip  **Note:** The syntax for UDAs is exactly the same whether a custom algorithm is hard-coded (i.e. defined at design time) or soft-coded (i.e. defined at silicon test time).

Overview & Terminology

There is ample literature covering memory test in general, test algorithms and memory fault coverage, so this section will only focus on the bare essentials.

A typical memory test usually applies a series of specific sequences across the entire memory array. A very simple March test might therefore be textually 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, each phase being 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 to be used nor the exact address locations to test for. To be applied to an actual memory, the test algorithm would therefore need an initialization sequence specifying such information.

Still, a typical Tessent MemoryBIST UDA can be described in a similar fashion; 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 - i.e. phase - and so on, until the entire algorithm has been executed.

Since a single memory BIST controller can test multiple memories in parallel, the default minimum and maximum address locations will 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 back at the above 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 and/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 (i.e. bottom, top, elsewhere)?
- In what direction is the address counter going (i.e. incrementing, decrementing, idle)?
- Is it incrementing/decrementing over the entire address range or just across some specific address segments (i.e. banks, rows or columns)?
- What is the increment/decrement value (i.e. +/-1, +/-2, etc. or something else)?
- What memory operation(s) is (are) applied at any given address, before moving on to the next address location (i.e. write, read, read-modify-write, write-read, no-op, etc.)?
- 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 (i.e. a scrambling table)?

Once you have answers to the above questions, coding a UDA will be a lot simpler. Many algorithm characteristics will directly translate into Tesson MemoryBIST properties.

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:

$$\{\uparrow\downarrow(w0); \uparrow(r0, w1); \uparrow(r1, w0); \downarrow(r0, w1); \downarrow(r1, w0); \uparrow\downarrow(r0)\}$$

Note that in the above algorithm variant, phases corresponding to steps 1 and 6 actually show arrows pointing both up and down; these phases can be executed either way. For simplicity, this example will still assume “up” and “down” directions for steps 1 and 6 (respectively).

Coding the Algorithm 297

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	297
MicroProgram Wrapper.....	298
Algorithm Optimization	303

TestRegisterSetup Wrapper

The TestRegisterSetup wrapper is used to determine 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 will be used? [Answer: SyncWR]
- How will the address counter count (i.e. column or row first)? [Answer: Row first]
- What default write & expect data pattern will be used? [Answer: All zeros]

The test register setup portion of this algorithm would therefore look like this (line numbers were added to the left for easier referencing):

```

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  : 8'b00000000;
13                load_expect_data : 8'b00000000;
14            }
15        }
16        MicroProgram {
17    }
18  }
19 }
```

Line 4 above indicates the desired operation set to choose operations from. 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 and/or observed?
- When should the algorithm advance to its next step or phase?

Procedure

1. The MBIST controller first writes the data pattern (e.g. zeros) 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        }
```

Above, 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 and/or row increment should occur. Line 3 implies that MBIST should write the data_reg value (i.e. zeros) to the memory.

Lines 15-18 state that when row and column address segments will both reach their maximum values (or “end count”), the algorithm should then move on to the next step (or “instruction”).

Note

 Tesson MemoryBIST uses “post-increment”-style address counters. Any memory operation is therefore fully executed 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 will wrap around and roll back to its minimum value. Conversely, a decrementing address segment reaching its minimum value will wrap around to its maximum value.

Given the above, this algorithm phase will thus effectively write zeros across the entire memory array. Once it reaches the top of the memory, it will perform one last write and will then wrap around to the bottom of the memory (i.e. 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 will be written to it. Lines 15-16 thus properly reflect which data to write and expect.

Since the algorithm will read and then write 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 will be selected (as indicated by lines 19-20).

3. This step turns out to be exactly the same as step 2 above, 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 above requires additional explanations.

Recall that the very next step (step 4) will perform 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 above with a read-modify-write operation performed at the maximum memory address, and then roll back (i.e. wrap around) to the minimum address location.

However, since 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 will thus start from the top of the memory - i.e. at its maximum address.

4. This phase of the algorithm is just like step 2, however it's performed with decrementing (instead of incrementing) address segments. At this point, you may already have figured its 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 end count properties with decrementing address segments. Tesson MemoryBIST considers the step as 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 above. The very next (and final) algorithm step will also be executed in a decrementing address order, so the BIST controller is allowed to roll back to the top once it completes the phase above.

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 (i.e. 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 will be 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 (i.e. in a soft-programmable MBIST controller). This may be more than what is actually available in a silicon device.

Tesson MemoryBIST supports the use of [RepeatLoopA/RepeatLoopB](#) wrapper loops, making it simple to re-execute 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.

Making a UDA Available to TS-MBIST

This procedure demonstrates how to implement a custom UDA (User-Defined Algorithm) with Tesson MemoryBIST. This part of the flow is normally performed via the DFT specification.

A custom UDA that has been separately validated or certified can then be implemented in larger designs. UDAs can either be hardcoded in silicon or softcoded 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 hardcoded or softcoded. The UDA must however be provided at design time if it will be hardcoded. When softcoded, the UDA can be modified even post-silicon and is only shifted into MBIST controllers at run time.

Chapter 3, “[Planning and Inserting MemoryBIST](#)”, covers the implementation of MemoryBIST referencing the built-in library algorithms. The procedure listed below focuses on the additional steps required for hardcoding UDAs into a design.

Procedure

1. Confirm the UDA has been separately certified or validated before implementing it.
2. Invoke Tesson Shell, set the tool context to dft -rtl mode, read cell libraries, set design sources and read all HDL for your design as per Chapter 3.

3. Make sure the UDA is read during setup with the `read_core_descriptions` command:

```
SETUP> read_core_descriptions \
.../data/design/mem/algo_march.tcd_mem_lib
```

4. Proceed with a normal TS-MBIST flow and generate a DFT specification. The UDA can be implemented if a corresponding MemoryOperationsSpecification wrapper appears in it.
5. Edit the DFT specification and add an `AdvancedOptions::extra_algorithms` property for the desired memory BIST controller instance(s), pointing to your UDA name(s).
6. Proceed with the rest of the standard MBIST flow, as described in Chapter 3. The indicated UDA(s) will be hardcoded into the MBIST controller's RTL.
7. When generating testbenches or patterns, follow the steps indicated in the next section (“Selecting a UDA for MBIST Execution”) for selecting & running the desired UDA.

Results

UDAs implemented with the above procedure become part of the RTL of specific MBIST controllers that are generated during this run. They are thus considered as being hard-coded. From this point on, they can be invoked exactly like any built-in Tessent library algorithms.

Selecting a UDA for MBIST Execution

This procedure explains how to execute a custom UDA (User-Defined Algorithm) that was previously hardcoded in a given Tessent MemoryBIST controller. This part of the flow is normally performed using the pattern specification.

Generated memory BIST controllers typically test multiple memories in parallel. Each tested memory has a default associated algorithm indicated in its memory TCD file. When a pattern specification is first created, tool automation considers all default algorithms and accordingly determines a suitable test plan that will test those memories in the most efficient way.

Unless a custom algorithm (also known as a user-defined algorithm or UDA) is defined as the default algorithm for a given memory, it will not be executed automatically. 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 hardcoded algorithms.

Prerequisites

This procedure assumes the UDA has already been hardcoded in a target memory BIST controller.

Procedure

1. Generate a patterns specification.
2. Edit the pattern specification and add an AdvancedOptions/apply_algorithm property to the desired memory BIST controller instance(s) and pointing to your UDA name.
3. Verify and process the patterns specification using the process_patterns_specification command.
4. Perform the rest of the standard MBIST pattern generation flow. This typically means running the run_testbench_simulations and check_testbench_simulations commands. The selected algorithm (whether a UDA or any other Tesson Library algorithm) will then be executed on the specified controller step.

Results

The UDA selected via the above procedure is executed by the specified memory BIST controller(s) in the desired controller step.

Examples

The example dofile provided below makes a UDA named “march” available to Tesson Shell MemoryBIST and then assigns it to memory BIST controller instance c1. The UDA is then selected for BIST execution and gets 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
DftSpecification(blockA,rtl)/MemoryBist/Controller(c1)/AdvancedOptions { march }
write_config_data generated.dft.spec -replace

process_dft_specification

extract_icl

set spec [create_patterns_spec]
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
write_config_data generated.pat.spec -replace

process_patterns_specification

run_testbench_simulations
check_testbench_simulations
```

Tip

- i** With the set_config_value command, determining the exact hierarchical names to use can sometimes be challenging. You can use introspection (i.e. get_config_elements) to determine the appropriate hierarchical path; you can also use the <N> syntax form, where <N> represents the Nth occurrence of a given wrapper in a DFT or pattern specification.
-

For instance, here is the pattern specification generated using the above example dofile:

```

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_tesson_mbist_c1_controller_inst) {
                    AdvancedOptions {
                        apply_algorithm : march;
                    }
                    RepairOptions {
                        check_repair_status : on;
                    }
                    DiagnosisOptions {
                        compare_go : on;
                        compare_go_id : on;
                    }
                }
            }
        }
    }
}

```

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_tesson_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

```

Since the command targets the very first occurrence of “PatternsSpecification”, the second occurrence of “Patterns”, the first occurrence of “TestStep”, and so on.

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	310
Diagnosis Objectives	310
Diagnosis Levels.....	310
Memory BIST Diagnosis Capabilities	311
Achieving Specific Diagnosis Level	312
Diagnosing Failing Memories Only	312
Diagnosing Failing Addresses and Bit-Mapping in a Memory	313
Enhanced Stop-On-Nth-Error Approach	314
Performing Enhanced Stop-On-Nth-Error Diagnosis for Bitmap Applications	315
Using Diagnosis with Local Comparators	316

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	310
Diagnosis Levels	310
Memory BIST Diagnosis Capabilities	311

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 and/or data bit.

Disabling of all diagnosis features will make 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.

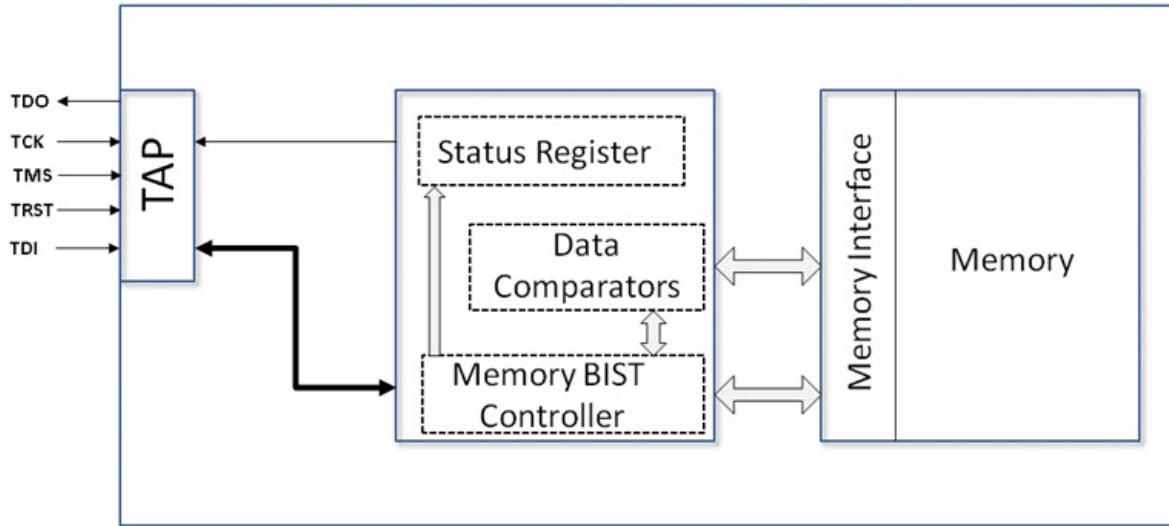
- *Offline Bit-Mapping*
 - Mapping the failure to bits in memory.
 - Useful when detailed root cause analysis is to be performed for yield improvements.
Note that in this case, diagnosis time is not relevant.

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	312
Diagnosing Failing Addresses and Bit-Mapping in a Memory	313

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:
 - The controller has either a single memory or step or is run with *freeze_step* set to *nn*
 - 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 test bench 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 due to 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 will stop the algorithm execution and hold the current state of the controller for analysis at the moment where a new error is detected and when the failure counter reaches 0.

Due to 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 in order 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. Due to 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 in order 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 in order 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/StopOnErrorHandler` wrapper.

Performing Enhanced Stop-On-Nth-Error Diagnosis for Bitmap Applications **315**

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.

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.

Using Diagnosis with Local Comparators

The diagnosis flow is essentially unchanged when local comparators are involved. The only difference is that the individual comparator status bits (*CMP_STAT_ID*) are distributed across the memory interfaces.

Chapter 9

Common Implementation Flows

This chapter describes various flows, or methods, of how to use Tessent MemoryBIST to implement your desired memory test solution in a design. It explains and shows examples of using the most common flows that are generally used for creating and inserting the memory BIST logic. The flows topic is covered at a higher level and won't repeat the detailed implementation specifics found in other sections of this manual and the Tessent Shell Reference Manual.

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 will be implemented in pieces as the design information is completed.

Top-Down Flow	317
Bottom-Up Flow.....	319

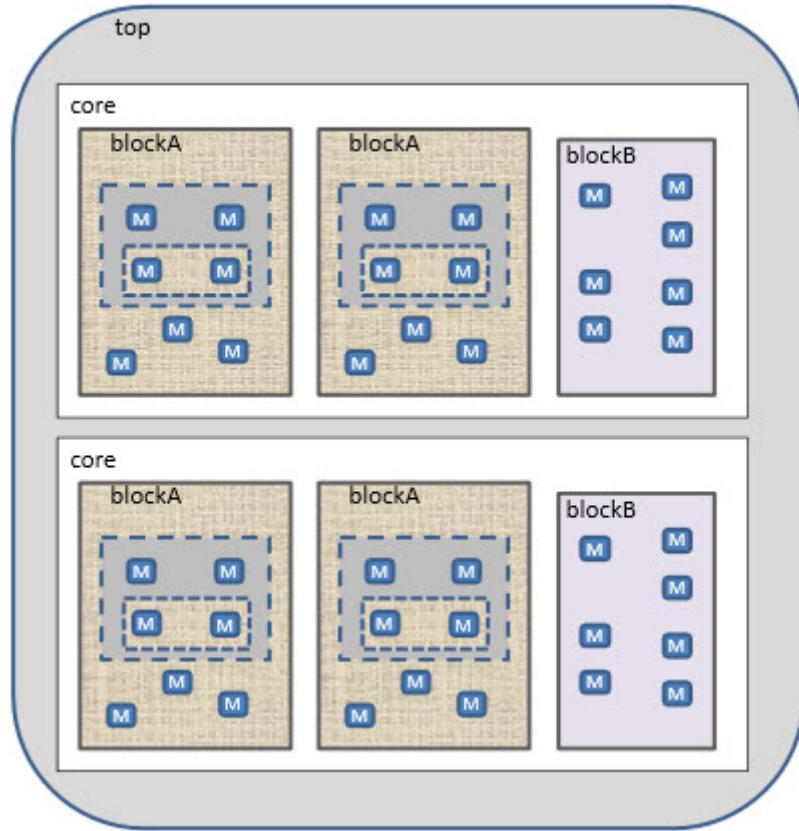
Top-Down Flow

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.

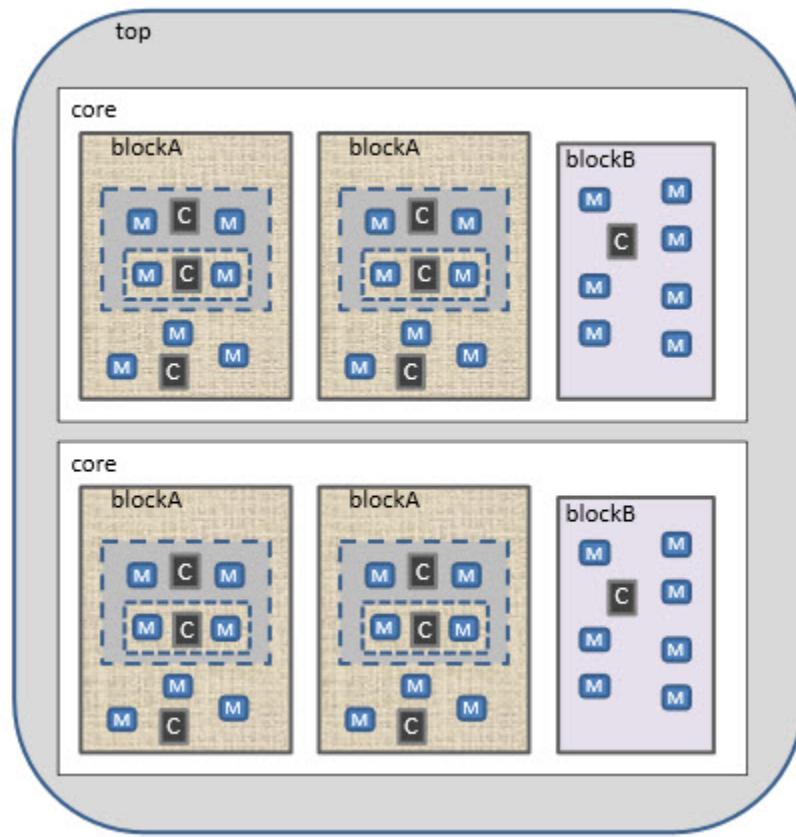
For example purposes, the block-based design diagram shown in [Figure 9-1](#) illustrates a hierarchical design that includes memories spread throughout the design. The top level of the design includes two identical cores. Each core has two copies of blockA and one of blockB. Within blockA there are 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 since 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 due to 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 test benches 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 allows 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](#).

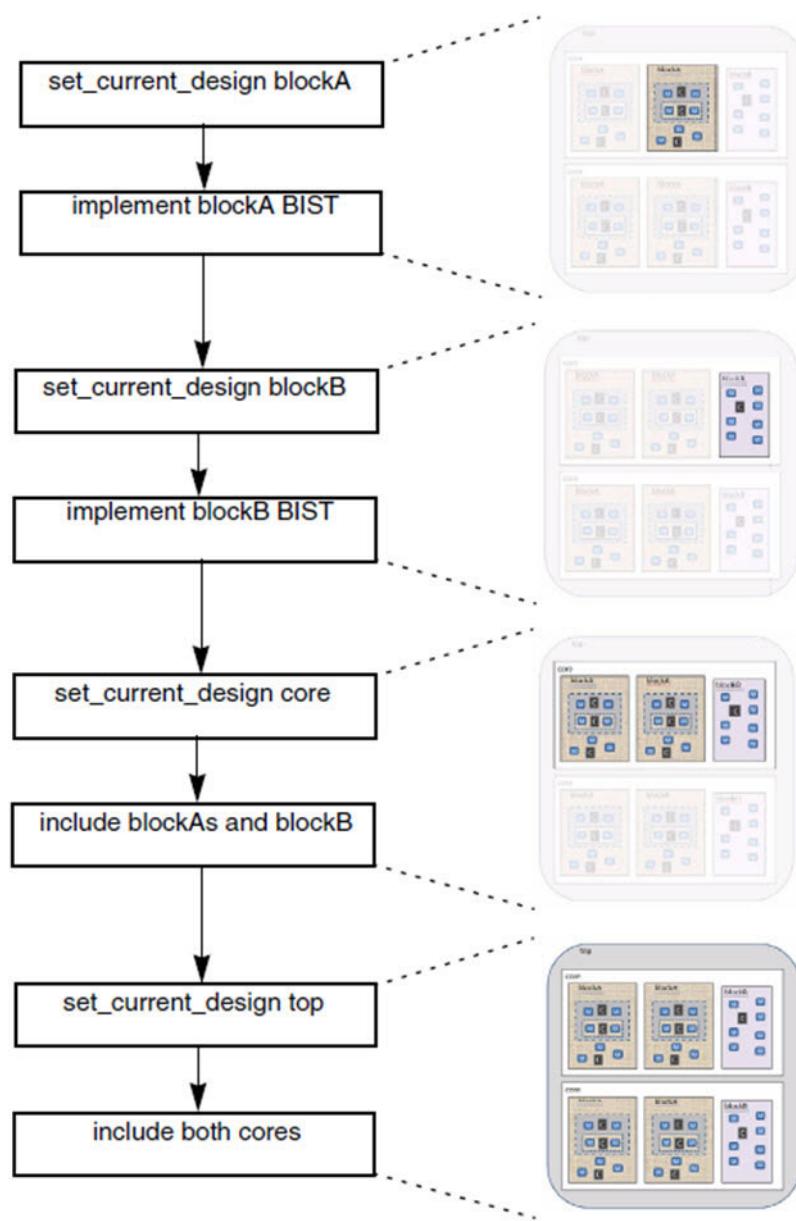
This one tool invocation methodology creates one large Tessent Shell Data Base (TSDB) that contains all of the data, but within that TSDB the results from the separate steps for the blocks and cores are separated. An example and description of this data base structure is provided in the “[Tessent Shell 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 will be 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 since 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	-SStatic	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.
<i>Italic</i>	dofile <i>filename</i>	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_dofile_abort <u>ON</u> <u>OFF</u>	An underlined item indicates either the default argument or the default value of an argument.
...	add_clocks <i>off_state</i> <u>primary_input_pin</u> ... [-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
<i>Italic</i>	<i>scan_in</i> : <i>port_pin_name</i> ;	An italic font indicates a user-supplied value.
Underline	wgl_type : <u>generic</u> <u>lsi</u> ;	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 : <i>port_naming</i> , ...;	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	325
Memory	327
Port	341
AddressCounter	352
PhysicalAddressMap	355
PhysicalDataMap	359
GroupWriteEnableMap.....	361
RedundancyAnalysis	363
RedundancyAnalysis/RowSegment	366
RedundancyAnalysis/ColumnSegment	369
PinMap	373
IclPorts	377
MemoryCluster	380
MemoryBistInterface	381
MemoryBistInterface/LogicalMemoryToInterfaceMapping	382
MemoryGroupAddressDecoding	384
PhysicalToLogicalMapping	386
PinMappings	387
FuseBoxInterface	390
Interface	393

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]". Since the tool will automatically look 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 don't need to specify this when loading a memory TCD file. The tool will 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
        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

Specifies the memory 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 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:

- The ReadOnly algorithm can only be used for MemoryType ROM.
- A ROM memory type can only use the ReadOnly algorithm.
- 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.
- 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.
- 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:

- The OperationSet property must specify an ATD waveform.
- If your address counter is not segmented, the value specified for NumberOfWords must be even.
- 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. 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:

- SMarchCHKB
- SMarchCHKBci
- SMarchCHKBcil
- 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.

The specified integer value must be between 1 and the value specified for the NumberOfBits property. The auto setting uses the value specified for the NumberOfBits property. The following example specifies that the memory array layout replicates in groups of 2 contiguous bits.

```
BitGrouping: 2;
```

- **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 allows 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 allows 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 allows 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:

- Two ReadWrite ports
- One Read-only port and one Write-only port
- Any number of Read-only ports and two Write-only ports
- DataOutHoldWithInactiveReadEnable : on | 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 : none | 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 | off;

A property that specifies the memory module containing scan circuitry that will be 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 will 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:

- If MemoryType is sram this property defaults to 1RW.
- If MemoryType is rom, this property defaults to 1R.

The following usage conditions apply:

- If MemoryType is rom, you cannot specify *W* or *RW*.
- If MemoryType is sram, you must specify at least one *R* and one *W* or one *RW*.
- MemoryHoldWithInactiveSelect : on | 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 | sram | dram;
A property that specifies the type of memory. The default value us 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* | auto;
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.

The default value is estimated as follows:

$$b * (0.004 + (c * 0.00008))$$

Where b is the number of bits per word and c is the number of columns or column mux option.

- 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 Synopsys, 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.

- NumberOfBits : *int* | auto;

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 $2n$, where n is the number of bits in that address segment.

Note



Mentor Graphics 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. ETAssemble either uses existing flip-flops within the interface for these sample points or adds additional flip-flops with the XOR gates if necessary. A value of off omits sample points from the interface.

- OperationSet : Async | AsyncWR | ROM | (Sync) | SyncWR | SyncWRvc | *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.

Note



Mentor Graphics provides a variety of library operation sets: two for asynchronous memories without clocks, two for asynchronous memories with at least one clock, one for read-only memories, and three for synchronous memories. These operation sets are generic. You can optionally create custom operation sets for the memories in your design.

Valid values are as follows:

- 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).
- AsyncWR — is a reserved string that specifies a library asynchronous operation set for a multi-port memory.
- ROM — is a reserved string that specifies the library operation set for ROMs.
- 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. Sync is the default operation set.
- SyncWR — is a reserved string that specifies a library synchronous operation set for a multi-port memory.

- 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 using the Programmable controller.
- *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.

Tip

 Mentor Graphics 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.

- 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.

- 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 desired retention time values 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. Since 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
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. Since 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 executing algorithms on a programmable controller using a custom operation set. ConcurrentRead should be used 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 will be turned 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. A value of off disables the shadow write. This is the default.

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:

- The memories must be synchronous SRAM. Preferably, Mentor Graphics 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.
- 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.
- The memories must have a BitGrouping of 1.
- 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
- The algorithm used must be SMarchCHKBci, SMarchCHKBcil, or SMarchCHKBvcd.
- If ShadowWrite is On, you must specify a row address. For information on specifying a row address, refer to the AddressCounter wrapper.
- 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:

- 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.

- none — Specifies that no bypass logic will be added in the memory interface. Specify none if you have a bidirectional data bus or the memory implements an internal bypass logic.
- 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

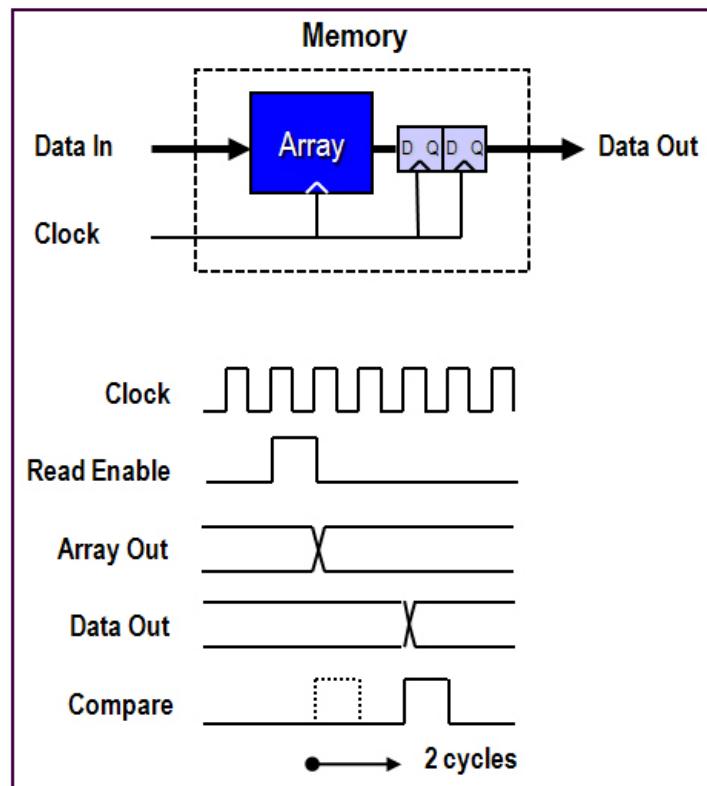
Example 1

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-1](#) illustrates 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 will be 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



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 defines the direction, function, and bus parameters for a signal port. The memory TCD includes a Port wrapper for each port on the 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 |
                                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.

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.

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 Mentor Graphics 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 : input | 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 | auto;

The DisableDuringScan property specifies if specific 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`, `User0`, `User1`, `User2`, `User3`, `User4`, `User5`, `User6`, `User7`. 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 will be 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`, `User1`, `User2`, `User3`, `User4`, `User5`, `User6`, `User7`, 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 a 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 <code>activate</code> property of the <code>DramSignals</code> 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.

Table A-3. Valid Function Values (cont.)

Function	Description
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 <i>BisrSerialData</i> port function is mandatory for memories with a serial BISR interface. You must specify the port direction using <i>Direction</i> : input output inout. In most cases, a memory with a serial BISR interface must have two <i>BisrSerialData</i> 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 <i>BisrSerialData</i> 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 <i>BIST_EN</i> 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 <i>BIST_ON</i> 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 <i>DramSignals</i> wrapper in the operation set.
Clock	Specifies a memory clock port.
Data	Specifies the data port. When <i>Data</i> is specified for the Function property, <i>Direction</i> indicates if <i>Data</i> is an input, output, or bidirectional port
GroupWrite Enable	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 <i>LogicalPort</i> property in their Port wrappers. The number of data bits in the group is defined using the <i>BitsPerWriteEnable</i> property. The group write enable ports can be controlled using the <i>even_group_write_enable</i> and <i>odd_group_write_enable</i> properties of the <i>Cycle</i> wrapper.
LogicHigh	Specifies that the associated port is to be tied to a logic high value.

Table A-3. Valid Function Values (cont.)

Function	Description
LogicLow	Specifies that the associated port is to be tied to a logic low value.
<u>None</u>	Specifies a port that does not need to be controlled during memory BIST. This is the default. The functional connection to this port will be 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 will not intercept the memory port.
Open	Specifies an unused output port of the memory that will 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.
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.
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.

Table A-3. Valid Function Values (cont.)

Function	Description
ShadowAddress Enable	<p>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.</p> <p>A memory with a port of this function can have ShadowWrite: <i>on</i>, 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.</p>
User0, User1, User2, User3, User4, User5, User6, User7	Allow 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, user1, user2, user3, user4, user5, user6, and user7 in the operation set.
ValidData	Specifies an output signal from the user logic which 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:

- 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 will 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 port Function value of *none* is specified, no mux is inserted and the user must specify a safevalue, which is used 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.
 - [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 memoryBIST mode	LogicHigh or LogicLow
Port is driven in functional mode but needs to be static in memory BIST mode	One of User0 to User7 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 User0-7.
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 execute 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 SDO_{UT} port:

```
Port (SDOUT) {  
    Function: BisrSerialData;  
    Direction: Output;  
    Retimed: On;  
}
```

- safevalue : *binary*; // Default: x

The safevalue property specifies an assumed value that is applied to the memory input ports with Function none during the controller assembly simulation only. The controller assembly module instantiates the controller, interfaces, and memory modules. It is used to verify the operation of the BIST circuitry.

Tessent MemoryBIST does not add additional logic to a memory port for which safevalue is specified. The assumed value is applied from the controller assembly test bench. Once the memory controller and interfaces are inserted into your design, the memory inputs defined with Function None will be driven by their functional values.

For single bit ports, the specified value can be indicated 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 controller assembly testbench. X denotes the value as ambiguous. This is the default. The value of 1'b0 or 0 sets the port to logic “0” in the controller assembly testbench. The value of 1'b1 or 1 sets the port to logic “1” in the controller assembly testbench.

For bussed ports, the specified value is required to be a binary value, matching the bit width of the port, or an error is generated. The exception is when specifying “X” for all bits of a bussed port. In this case, setting this property to X will assign “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. Further, the safevalue property only has an effect when simulating the controller assembly module.

- **EmbeddedTestLogic**

The EmbeddedTestLogic wrapper describes any embedded test interface feature that exist for the functional port being defined.

The following usage conditions apply:

- EmbeddedTestLogic wrappers cannot be defined for ports with the following functions: BistOn, BistEn, ScanTest, LogicLow, LogicHigh, Open, None.
- 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 Mentor Graphics tools or any simulator.

The following usage conditions apply:

- 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.
- 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
    }
}
```

- 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.
- The existence of this property must be consistent among all functional ports having the same function and assigned to the same logical port.
- 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/TestOutput : *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 Mentor Graphics tools or any simulator.

The following usage conditions apply:

- If this property is used with an Input port, then the TestInput property must also appear.
- The bus range must be the same as the range specified for the functional port being defined.
- 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;
}
```

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 allows 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 and/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 and/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.

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 will default 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:

- 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).
- 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.
- 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.
- If at least one address bit position is multiplexed, either all row address bits or all column address bits must be multiplexed as well.
- Address multiplexing is supported only for single-port (1RW) memories.

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 will default 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](#)

[CoreMemoryNumberOfWords](#)

[CoreMemoryCountRange](#)

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 will issue 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 will issue 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];  
}
```

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 facilitate layout, you must provide this information so Tessent MemoryBIST can correctly generate the checkerboard patterns.

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.

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 is 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.

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 will be 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 will be controlled by GWE bits 0 and 2 with the [Cycle/even_group_write_enable](#) property and data input bits 4 and 9 will be controlled by GWE bits 1 and 3 with the [Cycle/odd_group_write_enable](#) property. Note that data input bit 5 will not be 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.
-

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 wrapper is defined within the RedundancyAnalysis wrapper. If only one RowSegment wrapper is present, the row segment will encompass the entire memory address space.

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

- 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 wrapper is defined within the RedundancyAnalysis wrapper. If only one ColumnSegment wrapper is present, the column segment will encompass the entire memory address space.

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 RowSegment wrapper enables you to identify a segment of the memory which contains spare row elements. The row segment can cover the whole address space or a subset of the address space. Each segment will cover 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.

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:

- 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 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, column segment Bank2 will be enabled when address ports AD[11] is logic 1 and AD[10] is logic 0.

```

RowSegmentRange {
    SegmentAddress [1] : AddressPort (AD [11]) ;
    SegmentAddress [0] : AddressPort (AD [10]) ;
}
ColumnSegment (Bank2) {
    RowSegmentCountRange [2'b10:2'b10] ;
    .
    .
}
```

- 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:

- Fuse[3] — logs the address driven on the port AD[9] for the defective element
- Fuse[2] — logs the address driven on the port AD[8] for the defective element
- Fuse[1] — logs the address driven on the port AD[7] for the defective element
- 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 which 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 ColumnSegment wrapper enables you to identify a segment of the memory which contains spare IO/Column elements. The column segment 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.

Parameters

- **ColumnSegment(*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:

- 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 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, column segment Bank2 will be enabled when address ports AD[11] is logic 1 and AD[10] is logic 0.

```
RowSegmentRange {
    SegmentAddress[1] : AddressPort(AD[11]);
    SegmentAddress[0] : AddressPort(AD[10]);
}
ColumnSegment(Bank2) {
    RowSegmentCountRange[2'b10:2'b10];
.
.
}
```

- **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 Bank2 will be enabled when address ports AD[11] is logic 1 and AD[10] is logic 0.

```
ColumnSegmentRange {
    SegmentAddress [1] : AddressPort (AD [11]) ;
    SegmentAddress [0] : AddressPort (AD [10]) ;
}
ColumnSegment (Bank2) {
    ColumnSegmentCountRange [2'b10:2'b10] ;
.
.
}
```

- 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 will be allocated to this segment. The *port_name* value is a valid data port name. 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:

- 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 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 will be 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 will be added as the MSB of the fuse register.

These usage conditions apply:

- This property must be specified for all or none of ColumnSegment.
- If this property is defined, the specified value must be all zeroes.
- If this property is defined, the same value must be specified for all ColumnSegments.
- 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 will identify 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 allows 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

The contents of the PinMap wrapper allows you to apply the repair solution, computed by the BIRA engine, to the memory repair ports or the serial repair register. These mappings can be used to create the self-repair registers and connect them to the associated BIRA ports to enable efficient implementation of self-repair logic.

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.

For a memory with serial repair interface, you can specify the sequence of the internal BISR chain register within the memory.

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*] allows 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 will result 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 allows 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 will log 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 allows 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 will log 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 will be driven to the logic low value.

RepairRegister[*bit*] identifies the index of the internal BISR chain that will be 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]) {
        Attribute(attribute_name) : attribute_value ; // repeatable
    }
    DataOutPort (port_name[range]) {
        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 will be 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. The `create_dft_specification` command will consider the following ICL attributes. Other valid attributes may be specified and will be 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 will be 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 will be 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 bused ports. Refer to the [create_dft_specification](#) [DataInPorts](#) ICL element description on how to configure the TDR.

- 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 bused ports. Refer to the [create_dft_specification](#) [DataOutPorts](#) ICL element description on how to configure the TDR.

- 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 will collect all of the attributes specified within. If an attribute is duplicated, the value that is specified last will be retained.

Examples

Example 1

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 2

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 will be 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;
}
```

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

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/ 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 {
                }
            }
        }
    }
}
```

Description

The LogicalMemoryToInterfaceMapping wrapper is used in the [MemoryCluster](#)/
[MemoryBistInterface](#) wrapper.

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 Memory Cluster library and another within the Logical Memory core library.

Syntax

Usage 1: Memory Cluster Library

The MemoryGroupAddressDecoding wrapper in the [Memory Cluster](#) library 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 Library

The MemoryGroupAddressDecoding wrapper in the [Logical Memory](#) library file 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 memory cluster library file as shown in [Figure 6-5](#).

Usage 2: The MemoryGroupAddressDecoding wrapper is used in the logical memory core library 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 [MemoryGroupAddressDecoding](#) port function.
- 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. 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 {
            }
        }
    }
}
```

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.

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.

Syntax

Usage 1: Memory Cluster Library

The PinMappings wrapper 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.

```
Core(core_name) {
    MemoryCluster {
        MemoryBistInterface(id) {
            LogicalMemoryToInterfaceMapping(logical_memory_id) {
                PinMappings {
                    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

The PinMappings wrapper specifies the mappings of the physical memory ports to the logical memory ports.

```
Core(core_name) {
    Memory {
        PhysicalToLogicalMapping(physical_memory_id) { // Repeatable
            MemoryTemplate: physicalTemplateName;
            PinMappings {
                PhysicalMemoryDataInput[<indexList>]:
                    LogicalMemoryDataInput[<indexList>];
                PhysicalMemoryDataOutput[<indexList>]:
                    LogicalMemoryDataOutput[<indexList>];
                PhysicalMemoryAddress[<indexList>]:
                    LogicalMemoryAddress[<indexList>];
                PhysicalMemoryWriteAddress[<indexList>]:
                    LogicalMemoryWriteAddress[<indexList>];
                PhysicalMemoryReadAddress[<indexList>]:
                    LogicalMemoryReadAddress[<indexList>];
                PhysicalMemoryGroupWriteEnable[<indexList>]:
                    LogicalMemoryGroupWriteEnable[<indexList>];
            }
        }
    }
}
```

Parameters

Usage 1

- *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[]M*
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 memory cluster, for the 1RW memory, you specify the *LogicalMemoryAddress[]: InterfaceAddress[]* mapping. The Memory BIST controller drives both the read and write address ports, allowing the 1RW address port to be attached to either the read or write address port of the memory cluster.

Usage 2

- *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.

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 will be 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 takes to unload all the fuses when generating 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 will wait longer than it needs to in reality.
- `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 allowing 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 will take two cycles when accessing an internal fuse box and three cycles for an external fuse box.

- `programming_method : buffered | unbuffered ;`

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 | auto ;`

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
            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 lists 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

- 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 will be 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 will be 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 will be 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 will be 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 will be 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 will be 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 will be 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 you 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 programming 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 will be 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 will be 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 will be 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 will be 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 will be 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](#)

Appendix B

Configuration-Based Specification

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 will be 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 DFTVisualizer Configuration Data window.
delete_config_element	Deletes one or more configuration elements. Only elements which 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 Configuration Data 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
<code>get_config_selected_nodes</code>	Returns a collection containing the configuration elements corresponding to the tree nodes currently selected in the Configuration Data window.
<code>get_config_value</code>	Returns a value associated with a configuration element based on the specified option.
<code>move_config_element</code>	Moves a configuration element from one location to another.
<code>read_config_data</code>	Reads the content of configuration data files or a string into the Tesson Shell environment.
<code>report_config_data</code>	Reports the content of configuration wrappers in the transcript as it appears inside the file when the configuration data is written to a file using the <code>write_config_data</code> command.
<code>report_config_messages</code>	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.
<code>report_config_syntax</code>	Reports the legal configuration syntax for a specified configuration object.
<code>set_config_value</code>	Sets the value of an element in the configuration data.
<code>write_config_data</code>	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
<i>Italic</i>	<code>scan_in : port_pin_name;</code>	An italic font indicates a user-supplied value.
Underline	<code>wgl_type : <u>generic</u> lsi;</code>	An underlined item indicates the default value.
	<code>logic_level : <u>both</u> high low;</code>	The vertical bar separates a list of values from which you must choose one. Do not include the bar in the configuration file.
...	<code>port_naming : port_naming, ...;</code>	Ellipses indicate a repeatable value. The comment “// repeatable” also indicates a repeatable value.
//	<code>// default: ijtag_so</code>	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	400
Algorithm	401
AddressGenerator	406
DataGenerator	420
MicroProgram	423
Instruction/AdvancedOptions	425
Instruction/DataCommands	429
Instruction/AddressCommands	437
Instruction/CounterCommands	445
Instruction/NextConditions	447
OperationSet	466
SignalPipelineStages	468
Operation	470

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 content of the multiple MemoryOperationsSpecification wrappers will be 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 which 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 |
                                                num_address_bits_minus_one |
                                                num_data_bits_minus_one;
                                                // 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 |
                                                num_column_address_bits |
                                                num_row_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 will be hard coded into the controller.

You can define one or more algorithms in the Tessent MemoryBIST Algorithm file. There are multiple ways to define to the tools the algorithms you want.

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 algorithms in the DefaultsSpecification wrappers. Use the algorithm and extra_algorithm properties in the DefaultsSpecification/DftSpecification/MemoryBist/ControllerOptions wrapper.

You can edit the algorithm property and the extra_algorithm property of DftSpecification/MemoryBist/AdvancedOptions/Controller.

Parameters

- TestRegisterSetup/allow_multi_size_memory_parallel_test : on | off
This property indicates whether the current algorithm will be 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 which 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 allows you to specify the endcount value for the general purpose module CounterA. This endcount 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 will increment 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.

- integer — decimal number specifying the endcount value. The integer specified for the property must be in the range [0:2n-1]. n is the number of bits in CounterA .
- MinBank — specifies the starting bank address of the controller step to which the algorithm is applied and will be the lowest starting bank address of all memories tested in the controller step.
- MaxBank — specifies the ending bank address of the controller step to which the algorithm is applied and will be the highest ending bank address of all memories tested in the controller step.
- MinColumn — specifies the starting column address of the controller step to which the algorithm is applied and will be the lowest starting column address of all memories tested in the controller step.

- MaxColumn — specifies the ending column address of the controller step to which the algorithm is applied and will be the highest ending column address of all memories tested in the controller step.
- MinRow — specifies the starting row address of the controller step to which the algorithm is applied and will be the lowest starting row address of all memories tested in the controller step.
- MaxRow — specifies the ending row address of the controller step to which the algorithm is applied and will be the highest ending row address of all memories tested in the controller step.
- NumAddressBits — specifies the total number of bank, column and row address bits used in the controller step to which the algorithm is applied.
- NumBankAddressBits — specifies the total number of bank address bits used in the controller step to which the algorithm is applied.
- NumColumnAddressBits — specifies the total number of column address bits used in the controller step to which the algorithm is applied.
- NumRowAddressBits — 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 allows you to specify the endcount value for the general purpose module DelayCounter. This endcount 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 will increment 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 endcount value. The integer must be in the range [0:2n-1]. n is the number of bits in DelayCounter.
- MinBank — specifies the starting bank address of the controller step to which the algorithm is applied and will be the lowest starting bank address of all memories tested in the controller step.
- MaxBank — specifies the ending bank address of the controller step to which the algorithm is applied and will be the highest ending bank address of all memories tested in the controller step.
- MinColumn — specifies the starting column address of the controller step to which the algorithm is applied and will be the lowest starting column address of all memories tested in the controller step.

- MaxColumn — specifies the ending column address of the controller step to which the algorithm is applied and will be the highest ending column address of all memories tested in the controller step.
- MinRow — specifies the starting row address of the controller step to which the algorithm is applied and will be the lowest starting row address of all memories tested in the controller step.
- MaxRow — specifies the ending row address of the controller step to which the algorithm is applied and will be the highest ending row address of all memories tested in the controller step.
- NumAddressBits — specifies the total number of bank, column and row address bits used in the controller step to which the algorithm is applied.
- NumBankAddressBits — specifies the total number of bank address bits used in the controller step to which the algorithm is applied.
- NumColumnAddressBits — specifies the total number of column address bits used in the controller step to which the algorithm is applied.
- NumRowAddressBits — 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 | auto;

The optional property allows 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 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 Tesson MemoryBIST as follows:

- *On* — when an equation of the Core/Memory/PhysicalDataMap wrapper in the memory TCD file contains a not or xor term.
- *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 allows you to identify an OperationSet from which the microprogram select Operations to be performed on the memory.

- TestRegisterSetup/target_memory : all_compatible | 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 will not be included into 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 will not be hard coded into the controller if no step exists with only row_only memories.

Algorithms making use of the Column Address counter will automatically be excluded from 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 will be discarded.

- TestRegisterSetup/treat_bank_as_row_msb : on | off ;

This property indicates whether the bank address counter will be automatically configured as an extension of the row address counter. The Z address segment will be linked to the X1 address segment. The Z address segment will count 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 will be linked to the X1 address segment. A value of off specifies the Z address segment will not be linked to the X1 address segment.

Further, this property will be 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 will specify z_carry_out.

AddressGenerator

The AddressGenerator wrapper groups the properties of the address generator which 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 which require initialization prior to execution of the microprogram. The wrapper contains two possible wrappers, AddressRegisterA and AddressRegisterB.

Parameters

- `load_bank_address_max : bits | auto;`

The optional property allows 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.
- 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.

- 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.
- 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.
- 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 | auto;**

The optional property allows 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:

- 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.
- 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.
- 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.
- 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.
- 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 allows 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:

- 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.
- 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 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.
- 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.
- 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 | auto;**

The optional property allows 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:

- 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.
- 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 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.
- 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.
- 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 | auto;**

The optional property allows 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:

- 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.
- 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 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.
- 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.
- 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 | auto;`

The optional property allows 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:

- 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.

- 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 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.
- 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.
- 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 : min_bank | max_bank;
                    load_column_address : min_column | max_column;
                    load_row_address : 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. Each address register represents 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 allow 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 may be specified in the AddressGenerator wrapper.

Parameters

- AddressRegisterA | AddressRegisterB/load_bank_address : min_bank | max_bank;
The optional property allows 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 will be 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 will be highest ending bank address of all memories tested in the controller step.
- AddressRegisterA | AddressRegisterB/load_column_address : min_column | max_column;
The optional property allows 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.
MinColumn — The value of min_bank specifies the starting column address of the controller step to which the algorithm is applied and will be 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 will be the highest ending column address of all memories tested in the controller step.
- AddressRegisterA | AddressRegisterB/load_row_address : min_row | max_row |
The optional LoadRowAddress property allows 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 will be 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 will be the highest ending row address of all memories tested in the controller step.

- AddressRegisterA | AddressRegisterB/number_x0_bits : *int*; // default: 0

The optional property allows 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:

- The number of row address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD must be greater than zero.
 - The minimum value for *int* is zero.
 - 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:(2n-1)]. That is the row address CountRange for the X0 address segment must be a full binary count from all zeros to all ones.
 - number_c0_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 RowAddressCount 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 allows 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:
- The number of column address bits specified in any Core/Memory/AddressCounter wrapper of the memory TCD must be greater than zero.
 - The minimum value for *int* is zero.
 - 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:(2n-1)]. That is the column address CountRange for the Y0 address segment must be a full binary count from all zeros to all ones.
- 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 ColumnAddressCount set to On.
 - No y1_address is set to Hold in any Microprogram/Instruction/AddressCommands wrapper.
- AddressRegisterA | AddressRegisterB/x0_carry_in : none | 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:

- none — specifies that there is no carry in required for this counter segment. This is the default. This segment will count when instructed to count by the x0_address property of the Microprogram/Instruction/AddressCommands wrapper.
- 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.
- 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.
- 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.
- 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 : none | 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:

- none — specifies that there is no carry in required for this counter segment. This segment will count when instructed to count by the y0_address property of the Microprogram/Instruction/AddressCommands.
- 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.
- 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.
- 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.
- 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 : none | 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 will count 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:
 - max_y0_segment_bits set to 1.
 - The selected operation set has an operation with ColumnAddressCount set to On.
 - 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 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 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 : none | 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:

- `none` — specifies that there is no carry in required for this counter segment. This segment will count when instructed to count by the `z_address` property of the Microprogram/Instruction/AddressCommands wrapper.
- `x1_carry_out` — specifies that there is a carry out from the X1 address segment 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.

- `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.
- `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.
- `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 either 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 which 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 which require initialization prior to execution of the microprogram. Using this wrapper you can define for example the exact patterns that will be loaded into the WriteData register or the ExpectData register.

Parameters

- `load_write_data : pattern;`

The optional `load_write_data` property allows 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:

<code>binary</code>	A specified binary bit pattern to be written. The default value is 0s — for example 0b00..00.
<code>all_one</code>	A bit pattern of all 1s is loaded for writing — for example 0b11..11.
<code>all_zero</code>	A bit pattern of all 0s is loaded for writing — for example 0b00..00.
<code>alternating_zero</code>	A bit pattern of 0b10..10 is loaded for writing.
<code>alternating_one</code>	A bit pattern of 0b01..01 is loaded for writing.

- `load_expect_data : pattern;`

The optional `load_expect_data` property allows 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 : none | *int*;

The optional invert_data_with_row_bit property allows you to specify a row address bit which will invert 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 which 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 : none | *int*;

The optional invert_data_with_column_bit property allows you to specify a column address bit which will invert 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 which 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 MicroProgram wrapper allows you to describe the instructions scanned into the controller which control the execution of the memory test algorithm.

Syntax

```
MemoryOperationsSpecification {
    Algorithm(algorithm_name) {
        Microprogram {
            Instruction(instruction_name) {
                operation_select      : operation_name ; // default: NoOperation
                branch_to_instruction : instruction_name ;
                AdvancedOptions {
                }
                DataCommands {
                }
                AddressCommands {
                }
                CounterCommands {
                }
                NextConditions {
                }
            }
        }
    }
}
```

Description

The MicroProgram wrapper allows you to describe the instructions scanned into the controller which control the execution of the memory test algorithm. 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.

Parameters

- **Instruction(*instruction_name*)**
The Instruction is a mandatory wrapper containing properties which 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 allows you to identify an Operation from the selected operation set, which specifies the memory operation to be executed.
Valid values are as follows:
 - NoOperation — specifies that no operation has been selected. This is the default.
 - AutoRefresh — selects the AutoRefresh operation to be executed.

- *operation_name* — is a string which 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 allows you to specify the instruction which the pointer control selects as the next instruction for execution if all requested conditions are true (except for the MemoryOperationsSpecification/Algorithm(algorithm_name)/Microprogram/ Instruction(instruction_name)/NextConditions/RepeatLoop condition—that is all Repeat wrappers have not been executed). The next conditions are described in the NextCondition wrapper.

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 allows 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_write_enable : on | off;
                    disable_memories_without_output_enable      : on | off;
                    disable_memories_without_read_enable        : on | off;
                    disable_memories_without_select             : on | off;
                    disable_memories_without_write_enable       : on | off;
                    inhibit_refresh                           : on | off;
                }
            }
        }
    }
}
```

Description

The AdvancedOptions wrapper allows you to account for specific memory features when describing the memory test algorithm. 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_write_enable : on | off;

The disable_memories_without_group_write_enable 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 value off does not disable memories without the group write enable control signal for the specified instruction. This is the default.

A bit corresponding to the group 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 group write enable control signal AND the controller is soft programmable.
- 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_output_enable : on | off;

The disable_memories_without_output_enable 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 value off does not disable memories without the output enable control signal for the specified instruction. This is the default.

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_read_enable : on | off;

The disable_memories_without_read_enable 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 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. This is the default.

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:

- 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.
- 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:

- No select control signal is present. The Core/Memory/Port wrapper does not specify any port with function Select.
- 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 value off for the specified instruction, does not disable memories without the select control signal or that have Core/Memory/MemoryHoldWithInactiveSelect set to Off. This is the default.

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:

- 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.
- 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_write_enable : on | off;

The disable_memories_without_write_enable 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 on disables memories without the write enable control signal for the specified instruction.

A value off does not disable memories without the write enable control signal for the specified instruction. This is the default.

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 | off;

The InhibitRefresh property allows 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 allows 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 DataCommands wrapper allows you to manipulate the data pattern written to the memory input and expected from the memory output. You can select the pattern based on the output of the expect and write data registers as well as rotate 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 allows you to compare normally expected data and read data, or to disable any StrobeDataOut signal during execution 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 value off expected data and read data are compared normally. This is the default.

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 : 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 ;`

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.

- `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.
- `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.
- `reset_data_reg` — selects expect data of constant zeros. The value is not modified by any inversion with address bits.
- `set_data_reg` — selects expect data of constant ones. The value is not modified by any inversion with address bits.

These usage conditions apply:

- 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

- 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.
- 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.
- 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.
- 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.
- 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.

- *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)

- The polynomials used to generate the pseudo-random sequence are fixed and implemented as a Linear Feedback Shift Register (LFSR) of type I.
- Both the write data and expect data register must be initialized with the same seed. This seed cannot be all 0s.
- The read address sequence must exactly match the write address sequence so that the comparators do not report mis-compare.
- A memory location must be read before writing a new pseudo-random value into that location so that the comparators do not report mis-compare.

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* : 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 ;

The write_data property specifies the write data to be written to the memory.

Valid values are as follows:

- zero — selects write data of all zeros. This is the default
- one — selects write data of all ones.
- data_reg — selects the write data register loaded with a value specified by the DataGenerator/[load_write_data](#) property.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- reset_data_reg — selects write data of constant zeros. The value is not modified by any inversion with address bits.
- set_data_reg — selects write data of constant ones. The value is not modified by any inversion with address bits.

These usage conditions apply:

- 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

- 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.
- 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.
- 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.
- 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.
- 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.
- *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)

- The polynomials used to generate the pseudo-random sequence are fixed and implemented as a Linear Feedback Shift Register (LFSR) of type I.
- Both the write data and expect data register must be initialized with the same seed. This seed cannot be all 0s.
- The read address sequence must exactly match the write address sequence so that the comparators do not report mis-compare.
- A memory location must be read before writing a new pseudo-random value into that location so that the comparators do not report mis-compare.

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 allows 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 allows you to program a single instruction which can, for example, write a background of “zeros” but write “ones” to a single cell in the memory array.

Valid values are as follows:

- off — disables the inversion of the write or expect data registers when AddressRegisterA equals AddressRegisterB. This is the default.

- invert_write_data — enables the inversion of the applied write data registers when AddressRegisterA equals AddressRegisterB.
- invert_expect_data — enables the inversion of the applied expect data registers when AddressRegisterA equals AddressRegisterB.
- 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 allows 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.

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 allows 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 2 address registers. 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 allows you to select address register A or address register B and to manipulate the address registers. The content of the selected address register will be 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 executed on.

Valid values are as follows:

- select_a — applies AddressRegisterA to the memory address bus, and the address segment commands are executed on AddressRegisterA. This is the default.
- select_b — applies AddressRegisterB to the memory address bus, and the address segment commands are executed on AddressRegisterB.
- 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 executed on AddressRegisterA.
- select_a_copy_to_b — applies AddressRegisterA to the memory address bus, the address segment commands are executed on AddressRegisterA, and the result is copied to AddressRegisterB.
- select_b_copy_to_a — applies AddressRegisterB to the memory address bus, the address segment commands are executed on AddressRegisterB, and the result is copied to AddressRegisterA.
- select_a_rotate_b — applies AddressRegisterA to the memory address bus, the address segment commands are executed on AddressRegisterA, and the AddressRegisterB is rotated left.
- select_b_rotate_a — applies AddressRegisterB to the memory address bus, the address segment commands are executed on AddressRegisterB, and the AddressRegisterA is rotated left.
- select_a_rotate_left_b — applies AddressRegisterA to the memory address bus, the address segment commands are executed on AddressRegisterA, and AddressRegisterB is rotated towards the MSB. This command is equivalent to select_a_rotate_b.
- select_b_rotate_left_a — applies AddressRegisterB to the memory address bus, the address segment commands are executed on AddressRegisterB, and AddressRegisterA is rotated towards the MSB. This command is equivalent to select_b_rotate_a.
- select_b_rotate_right_a — applies AddressRegisterB to the memory address bus, the address segment commands are executed on AddressRegisterB, and AddressRegisterA is rotated towards the LSB.

These usage conditions apply:

- The address register selected is valid for the execution of the selected operation.
- The *copy* command is performed on the last tick of the selected operation.

- 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 will 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 executed 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 allows 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 value *on* 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 to Increment or Decrement is executed normally. This is the default.

These usage conditions apply:

- For an instruction which 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.
- 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;
        }
    }
}
```

- `x1_address : hold | increment | decrement | load_min | load_max;`

The `x1_address` property allows 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 : hold | increment | decrement | load_min | load_max;

The x0_address property allows 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` : hold | `increment` | `decrement` | `load_min` | `load_max`;

The `y1_address` property allows you to specify the address counting command for the Y1 segment of the column address.

Valid values are as follows:

- `hold` — specifies that the Y1 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 Y1 segment of the selected address register.
- `decrement` — specifies a decrementing count direction from maximum to minimum for the Y1 segment of the selected address register.
- `load_min` — loads the Y1 address segment with the value defined by the [AddressGenerator/load_row_address_min](#) property.
- `load_max` — loads the Y1 address segment with the value defined by the [AddressGenerator/load_row_address_max](#) property.

The following usage conditions apply:

- 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.
- 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 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 : hold | increment | decrement | load_min | load_max;

The y0_address property allows 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 : hold | increment | decrement | load_min | load_max;

The z_address property allows 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 allows 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 CounterCommands wrapper allows you to specify how the CounterA and the DelayCounter are updated at the end of the instruction. You can independently preserve the counter value or request the counter to increment by 1.

Parameters

- counter_a : hold | increment;

The counter_a property allows 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;

        }
    }
}
```

- 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 which 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 mandatory NextConditions wrapper contains properties which are used to sequence the microprogram instructions. The instruction sequencing is 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.

Parameters

- `x0_end_count : on | off;`

The `x0_end_count` property allows 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 allows 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:

- 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.
- 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 allows 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:

- The [Instruction/AddressCommands/y0_address](#) property is set to increment, and the Y0 segment has reached the maximum of the column address counrange specified in the [AddressCounter/Function/CountRange](#) property of the memory TCD file.
- The [Instruction/AddressCommands/y0_address](#) property is set to decrement, and the Y0 segment has reached the minimum of the column address counrange 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:

- The Controller/AlgorithmResourceOptions/address_segment_x0_y0_allowed is on in the MemoryBist wrapper of the DftSpecification.
- 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.
- 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 allows 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 counrange 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 counrange 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 allows 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 counrange 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 counrange 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.

- 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 allows 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 value *on* 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 allows 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 value *on* 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-executed 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-executes each Repeat sub-wrapper containing properties which modify the address and data commands. However, each of these Repeat wrappers is also influenced by the outer loop Repeat wrapper's properties which modify the address and data commands.

These usage conditions apply:

- Each of the RepeatLoopA and RepeatLoopB wrappers may be used only once in the MicroProgram wrapper.
- 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.
- 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 allows you to specify the instruction which the pointer control selects as the next instruction for execution if all requested conditions are true (except for the RepeatLoop condition—that is all Repeat wrappers have not been executed). 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:

- The *instruction_name* must identify an Instruction wrapper name.
- *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.
- 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.
- 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 which allow 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 allows you to specify that the address segment commands executed by instructions during the Repeat will either be executed as specified by the instruction or the command will be modified to a complimentary command.

Valid values are as follows:

- no_change — specifies that the address sequencing is to remain as programmed for execution of the Repeat wrapper. This is the default.
- 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:

- 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

- 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 <i>Repeat</i> is not active	RepeatLoop(B)/ Repeat/ address_sequence “_” indicates the <i>Repeat</i> 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	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
<i>increment</i>	<i>no_change</i>	<i>no_change</i>	<i>increment</i>
<i>increment</i>	<i>no_change</i>	<i>inverse</i>	<i>decrement</i>
<i>increment</i>	<i>inverse</i>	<i>no_change</i>	<i>decrement</i>
<i>increment</i>	<i>inverse</i>	<i>inverse</i>	<i>increment</i>
<i>increment</i>	-	<i>no_change</i>	<i>increment</i>
<i>increment</i>	-	<i>inverse</i>	<i>decrement</i>
<i>decrement</i>	<i>no_change</i>	-	<i>decrement</i>
<i>decrement</i>	<i>inverse</i>	-	<i>increment</i>
<i>decrement</i>	<i>no_change</i>	<i>no_change</i>	<i>decrement</i>
<i>decrement</i>	<i>no_change</i>	<i>inverse</i>	<i>increment</i>
<i>decrement</i>	<i>inverse</i>	<i>no_change</i>	<i>increment</i>
<i>decrement</i>	<i>inverse</i>	<i>inverse</i>	<i>decrement</i>
<i>decrement</i>	-	<i>no_change</i>	<i>decrement</i>
<i>decrement</i>	-	<i>inverse</i>	<i>increment</i>
<i>load_min</i>	<i>no_change</i>	-	<i>load_min</i>
<i>load_min</i>	<i>inverse</i>	-	<i>load_min</i>
<i>load_min</i>	<i>no_change</i>	<i>no_change</i>	<i>load_min</i>
<i>load_min</i>	<i>no_change</i>	<i>inverse</i>	<i>load_min</i>
<i>load_min</i>	<i>inverse</i>	<i>no_change</i>	<i>load_min</i>
<i>load_min</i>	<i>inverse</i>	<i>inverse</i>	<i>load_min</i>
<i>load_min</i>	-	<i>no_change</i>	<i>load_min</i>
<i>load_min</i>	-	<i>inverse</i>	<i>load_min</i>
<i>load_max</i>	<i>no_change</i>	-	<i>load_max</i>
<i>load_max</i>	<i>inverse</i>	-	<i>load_max</i>
<i>load_max</i>	<i>no_change</i>	<i>no_change</i>	<i>load_max</i>
<i>load_max</i>	<i>no_change</i>	<i>inverse</i>	<i>load_max</i>
<i>load_max</i>	<i>inverse</i>	<i>no_change</i>	<i>load_max</i>

Table B-4. Nested address_sequence Property Modification (cont.)

x1_address	RepeatLoop(A)/ Repeat/ address_sequence “_” indicates the <i>Repeat</i> is not active	RepeatLoop(B)/ Repeat/ address_sequence “_” indicates the <i>Repeat</i> 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 : no_change | inverse;

The expect_data_sequence property allows you to specify that the expect data commands executed by instructions during the Repeat will either be executed as specified by the instruction or the command will be 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

- 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
<i>data_reg_rotate</i>	-	<i>no_change</i>	<i>data_reg_rotate</i>
<i>data_reg_rotate</i>	-	<i>inverse</i>	<i>inverse_data_reg_rotate</i>
<i>inverse_data_reg_rotate</i>	<i>no_change</i>	-	<i>inverse_data_reg_rotate</i>
<i>inverse_data_reg_rotate</i>	<i>inverse</i>	-	<i>data_reg_rotate</i>
<i>inverse_data_reg_rotate</i>	<i>no_change</i>	<i>no_change</i>	<i>inverse_data_reg_rotate</i>
<i>inverse_data_reg_rotate</i>	<i>no_change</i>	<i>inverse</i>	<i>data_reg_rotate</i>
<i>inverse_data_reg_rotate</i>	<i>inverse</i>	<i>no_change</i>	<i>data_reg_rotate</i>
<i>inverse_data_reg_rotate</i>	<i>inverse</i>	<i>inverse</i>	<i>inverse_data_reg_rotate</i>
<i>inverse_data_reg_rotate</i>	-	<i>no_change</i>	<i>inverse_data_reg_rotate</i>
<i>inverse_data_reg_rotate</i>	-	<i>inverse</i>	<i>data_reg_rotate</i>
<i>inverse_data_reg_rotate_with_invert</i>	<i>no_change</i>	-	<i>data_reg_rotate_with_invert</i>
<i>data_reg_rotate_with_invert</i>	<i>inverse</i>	-	<i>inverse_data_reg_rotate_with_invert</i>
<i>data_reg_rotate_with_invert</i>	<i>no_change</i>	<i>no_change</i>	<i>data_reg_rotate_with_invert</i>
<i>data_reg_rotate_with_invert</i>	<i>no_change</i>	<i>inverse</i>	<i>inverse_data_reg_rotate_with_invert</i>
<i>data_reg_rotate_with_invert</i>	<i>inverse</i>	<i>no_change</i>	<i>inverse_data_reg_rotate_with_invert</i>
<i>data_reg_rotate_with_invert</i>	<i>inverse</i>	<i>inverse</i>	<i>data_reg_rotate_with_invert</i>
<i>data_reg_rotate_with_invert</i>	-	<i>no_change</i>	<i>data_reg_rotate_with_invert</i>
<i>data_reg_rotate_with_invert</i>	-	<i>inverse</i>	<i>inverse_data_reg_rotate_with_invert</i>
<i>inverse_data_reg_rotate_with_invert</i>	<i>no_change</i>	-	<i>inverse_data_reg_rotate_with_invert</i>
<i>inverse_data_reg_rotate_with_invert</i>	<i>inverse</i>	-	<i>data_reg_rotate_with_invert</i>

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_invert
inverse_data_reg_rotate_with_invert	inverse	no_change	data_reg_rotate_with_invert
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_invert

- write_data_sequence : no_change | inverse;

The write_data_sequence property allows you to specify that the commands executed by instructions during Repeat will either be executed as specified by the instruction, or the command will be modified to a complimentary command.

Valid values are as follows:

- no_change — specifies that the Write data sequencing is to remain as programmed for execution of the Repeat. This is the default.
- 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:

- 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
<i>zero</i>	<i>no_change</i>	<i>zero</i>
<i>zero</i>	<i>inverse</i>	<i>one</i>
<i>one</i>	<i>no_change</i>	<i>one</i>
<i>one</i>	<i>inverse</i>	<i>zero</i>
<i>date_reg</i>	<i>no_change</i>	<i>date_reg</i>
<i>date_reg</i>	<i>inverse</i>	<i>inverse_date_reg</i>
<i>inverse_date_reg</i>	<i>no_change</i>	<i>inverse_date_reg</i>
<i>inverse_date_reg</i>	<i>inverse</i>	<i>date_reg</i>
<i>date_reg_rotate</i>	<i>no_change</i>	<i>date_reg_rotate</i>
<i>date_reg_rotate</i>	<i>inverse</i>	<i>inverse_date_reg_rotate</i>
<i>inverse_date_reg_rotate</i>	<i>no_change</i>	<i>inverse_date_reg_rotate</i>
<i>inverse_date_reg_rotate</i>	<i>inverse</i>	<i>date_reg_rotate</i>
<i>date_reg_rotate_with_invert</i>	<i>no_change</i>	<i>date_reg_rotate_with_invert</i>
<i>date_reg_rotate_with_invert</i>	<i>inverse</i>	<i>inverse_date_reg_rotate_with_invert</i>
<i>inverse_date_reg_rotate_with_</i> <i>invert</i>	<i>no_change</i>	<i>inverse_date_reg_rotate_with_invert</i>
<i>inverse_date_reg_rotate_with_</i> <i>invert</i>	<i>inverse</i>	<i>date_reg_rotate_with_invert</i>

- 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
<i>zero</i>	<i>no_change</i>	-	<i>zero</i>
<i>zero</i>	<i>inverse</i>	-	<i>one</i>
<i>zero</i>	<i>no_change</i>	<i>no_change</i>	<i>zero</i>
<i>zero</i>	<i>no_change</i>	<i>inverse</i>	<i>one</i>

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	<i>inverse</i>	<i>no_change</i>	<i>one</i>
zero	<i>inverse</i>	<i>inverse</i>	<i>zero</i>
zero	-	<i>no_change</i>	<i>zero</i>
zero	-	<i>inverse</i>	<i>one</i>
one	<i>no_change</i>	-	<i>one</i>
one	<i>inverse</i>	-	<i>zero</i>
one	<i>no_change</i>	<i>no_change</i>	<i>one</i>
one	<i>no_change</i>	<i>inverse</i>	<i>zero</i>
one	<i>inverse</i>	<i>no_change</i>	<i>zero</i>
one	<i>inverse</i>	<i>inverse</i>	<i>one</i>
one	-	<i>no_change</i>	<i>one</i>
one	-	<i>inverse</i>	<i>zero</i>
date_reg	<i>no_change</i>	-	<i>date_reg</i>
date_reg	<i>inverse</i>	-	<i>inverse_date_reg</i>
date_reg	<i>no_change</i>	<i>no_change</i>	<i>date_reg</i>
date_reg	<i>no_change</i>	<i>inverse</i>	<i>inverse_date_reg</i>
date_reg	<i>inverse</i>	<i>no_change</i>	<i>inverse_date_reg</i>
date_reg	<i>inverse</i>	<i>inverse</i>	<i>date_reg</i>
date_reg	-	<i>no_change</i>	<i>date_reg</i>
date_reg	-	<i>inverse</i>	<i>inverse_date_reg</i>
inverse_date_reg	<i>no_change</i>	-	<i>inverse_date_reg</i>
inverse_date_reg	<i>inverse</i>	-	<i>date_reg</i>
inverse_date_reg	<i>no_change</i>	<i>no_change</i>	<i>inverse_date_reg</i>
inverse_date_reg	<i>no_change</i>	<i>inverse</i>	<i>date_reg</i>
inverse_date_reg	<i>inverse</i>	<i>no_change</i>	<i>date_reg</i>
inverse_date_reg	<i>inverse</i>	<i>inverse</i>	<i>inverse_date_reg</i>
inverse_date_reg	-	<i>no_change</i>	<i>inverse_date_reg</i>
inverse_date_reg	-	<i>inverse</i>	<i>date_reg</i>

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
<i>date_reg_rotate</i>	<i>no_change</i>	-	<i>date_reg_rotate</i>
<i>date_reg_rotate</i>	<i>inverse</i>	-	<i>inverse_date_reg_rotate</i>
<i>date_reg_rotate</i>	<i>no_change</i>	<i>no_change</i>	<i>date_reg_rotate</i>
<i>date_reg_rotate</i>	<i>no_change</i>	<i>inverse</i>	<i>inverse_date_reg_rotate</i>
<i>date_reg_rotate</i>	<i>inverse</i>	<i>no_change</i>	<i>inverse_date_reg_rotate</i>
<i>date_reg_rotate</i>	<i>inverse</i>	<i>inverse</i>	<i>date_reg_rotate</i>
<i>date_reg_rotate</i>	-	<i>no_change</i>	<i>date_reg_rotate</i>
<i>date_reg_rotate</i>	-	<i>inverse</i>	<i>inverse_date_reg_rotate</i>
<i>inverse_date_reg_rotate</i>	<i>no_change</i>	-	<i>inverse_date_reg_rotate</i>
<i>inverse_date_reg_rotate</i>	<i>inverse</i>	-	<i>date_reg_rotate</i>
<i>inverse_date_reg_rotate</i>	<i>no_change</i>	<i>no_change</i>	<i>inverse_date_reg_rotate</i>
<i>inverse_date_reg_rotate</i>	<i>no_change</i>	<i>inverse</i>	<i>date_reg_rotate</i>
<i>inverse_date_reg_rotate</i>	<i>inverse</i>	<i>no_change</i>	<i>date_reg_rotate</i>
<i>inverse_date_reg_rotate</i>	<i>inverse</i>	<i>inverse</i>	<i>inverse_date_reg_rotate</i>
<i>inverse_date_reg_rotate</i>	-	<i>no_change</i>	<i>inverse_date_reg_rotate</i>
<i>inverse_date_reg_rotate</i>	-	<i>inverse</i>	<i>date_reg_rotate</i>
<i>date_reg_rotate_with_invert</i>	<i>no_change</i>	-	<i>date_reg_rotate_with_invert</i>
<i>date_reg_rotate_with_invert</i>	<i>inverse</i>	-	<i>inverse_date_reg_rotate_with_invert</i>
<i>date_reg_rotate_with_invert</i>	<i>no_change</i>	<i>no_change</i>	<i>date_reg_rotate_with_invert</i>
<i>date_reg_rotate_with_invert</i>	<i>no_change</i>	<i>inverse</i>	<i>inverse_date_reg_rotate_with_invert</i>
<i>date_reg_rotate_with_invert</i>	<i>inverse</i>	<i>no_change</i>	<i>inverse_date_reg_rotate_with_invert</i>
<i>date_reg_rotate_with_invert</i>	<i>inverse</i>	<i>inverse</i>	<i>date_reg_rotate_with_invert</i>

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 allows 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 to *increment* or *decrement* is executed 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 allows you to compare normally expected data and read data, or to disable any StrobeDataOut signal during execution 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

 Mentor Graphics 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.

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:

- 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).
- AsyncWR — is a reserved string that specifies a library asynchronous operation set for a multi-port memory.
- ROM — is a reserved string that specifies the library operation set for ROMs.
- 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.

- SyncWR — is a reserved string that specifies a library synchronous operation set for a multi-port memory.
- SyncWRvcld — 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 SMArchCHKBvcld algorithm.
- *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, Tessonnt MemoryBIST searches the specified memory library files for an OperationSet wrapper with the same identifier.

SignalPipelineStages

The optional **SignalPipelineStages** wrapper allows 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 {
                user0 : int ; // default: 0
                user1 : int ; // default: 0
                user2 : int ; // default: 0
                user3 : int ; // default: 0
                user4 : int ; // default: 0
                user5 : int ; // default: 0
                user6 : int ; // default: 0
                user7 : 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 optional SignalPipelineStages wrapper allows you to declare the number of pipelining stages required for a BIST function in a particular OperationSet. The waveform for the BIST function 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.
- `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 - user7: int ; // default: 0`
The `user<n>` ($n=0,\dots,7$) 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, Tesson 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 allows 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

The AddressOverrides wrapper allows you to define properties for forcing a particular value on the row address, column address or bank address. All event sequences are synchronous with respect to *cycles* of an implied BIST clock.

Parameters

- *row_address* : *binary*;

The *row_address* property is optional. This property allows you to identify the row address bits to be forced and specify the value which 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 will 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 will be forced to 'x'.

- column_address : *binary*;

The column_address property is optional. This property allows you to identify the column address bits to be forced and specify the value which 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 will 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 allows you to identify the bank address bits to be forced and specify the value which 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 will 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 allows 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 allows 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 which 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 | auto;

The *read_enable* property controls the memory port which uses the Function property (Core/Memory/Port) specified to *ReadEnable*.

Valid values are as follows:

- on — activates the *ReadEnable* signal.
- off — deactivates the *ReadEnable* 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 *readenable*.

- **output_enable:** on | off | auto;

The *output_enable* property allows you to enable output drivers of the embedded memory.

Valid values are as follows:

- on — activates the output enable signal.
- off — deactivates the output enable signal.
- 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 | off;

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:

- Memory output data must be valid at the end of the clock cycle in which `strobe_data_out` occurs.
- All OperationSet wrappers require at least one operation that contains a `strobe_data_out`.
- A `strobe_data_out` property can be specified on any or all Cycles of any Operation as timing permits.
- 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.
- 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 allows you to control the memory WriteEnable signal.

Valid values are as follows:

- `on` — activates the WriteEnable signal.
- `off` — deactivates the WriteEnable 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 `WriteEnable`.

Refer to the [Note](#) in the `odd_group_write_enable` property description for further information.

- even_group_write_enable: on | off | auto;

The even_group_write_enable property enables you to control the memory GroupWriteEnable signals which 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 will be inverted in the cycle where even_group_write_enable is off and odd_group_write_enable is on.

Starting with the v2018.4 release, group write enables can be controlled during concurrent write operations. They were previously forced active during these operations. This feature may be useful when creating custom algorithms and operation sets for memories with write-only ports. The value of even_group_write_enable and odd_group_write_enable is used for both the active and concurrent write port.

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 | auto;

The odd_group_write_enable property enables you to control the memory GroupWriteEnable signals which 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 will be inverted in the cycle where odd_group_write_enable is off and even_group_write_enable is on.

Starting with the v2018.4 release, group write enables can be controlled during concurrent write operations. They were previously forced active during these operations. This feature may be useful when creating custom algorithms and operation sets for memories with write-only ports. The value of even_group_write_enable and odd_group_write_enable is used for both the active and concurrent write port.

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 | auto:
 - 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 | auto;

The column_address_count_enable property allows 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:

- on — enables counting of the Y Column Address segment.
- 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:

- This property is used only for programmable memory BIST controllers.
- 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;
```

- 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 execute 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 will be incremented instead as if row_address_count_enable: on had been specified in the operation.

Number of Column Address Bits	Automatic Modification
1	No change required to the algorithm or operation.
2 or more	The algorithm and operation will be automatically modified to segment the address register (number_y0_bits: 1 and number_x0_bits: 1).

Note

 Algorithms explicitly specifying address register segmentation (i.e. 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.

- 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;	Disables the implementation of the carry-look-ahead (CLA) circuit.
Algorithm	<pre>AddressRegisterA { x1_carry_in: y1_carry_out; y0_carry_in: x1_carry_out; y1_carry_in: none; }</pre>	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	<pre>Operation (Write2CellsFastY) { Cycle { select: on; write_enable: on; read_enable: off; AdvancedSignals { column_address_count_enable: on; } } }</pre>	This is a typical example. The operation length is arbitrary and column_address_count_enable can be present in any cycle.
-----------	---	---

These settings allow full flexibility of the address sequence but the resulting circuit will run at significantly lower speed than for the default (i.e. max_y0_segment_bits: 1 and max_x0_segment_bits: 1). The circuit will also be 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 will typically require the use of freeze_step: *int*. This will increase test time due to more frequent controller setup operations.

- row_address_count_enable: on | off | auto;

The row_address_count_enable property allows 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:

- on — enables counting of the *X* Row Address segment.
- 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:

- This property is used only for programmable memory BIST controllers.

- 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;
```

- 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 (i.e. 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.

- 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	<code>max_y0_segment_bits: auto;</code>	Disables the implementation of the carry-look-ahead (CLA) circuit.
Algorithm	<pre>AddressRegisterA { y1_carry_in: x1_carry_out; x0_carry_in: y1_carry_out; x1_carry_in: none; }</pre>	This setting is only an example. The essential setting is highlighted in red.
Operation	<pre>Operation (Write2CellsFastX) { Cycle { select: on; write_enable: on; read_enable: off; AdvancedSignals { row_address_count_enable: on; } } Cycle { } }</pre>	This is a typical example. The operation length is arbitrary and <code>row_address_count_enable</code> can be present in any cycle.

These settings allow full flexibility of the address sequence but the resulting circuit will run at significantly lower speed than for the default (i.e. `max_x0_segment_bits: 1`). The circuit will also be 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 will be selecting `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:

- You need to have specific operation name that use this property.
- The `switch_address_register` property works for operations with two *Cycles*.
- The `switch_address_register` property can only be force to *On* in the second *Cycle* of the Operation.
- If the operation requires three or 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.
- The instruction before the current instruction must have `expect_data` property defined in the Instruction/DataCommands wrapper.
- The assumption is made that the *Away* cell and the *Home* cell hold different values.
- `address_override_enable: on | off | auto;`
 - 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 | off;`

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:

- *on* — inverts the expect data.
- *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 | off;`

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:

- *on* — inverts the write data.
- *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 which uses the Function property specified to user<n>.

Syntax

```
MemoryOperationsSpecification {
    OperationSet(operation_set_name) {
        Operation(operation_name) {
            Cycle {
                UserSignals {
                    user0 : on | off | auto;
                    user1 : on | off | auto;
                    user2 : on | off | auto;
                    user3 : on | off | auto;
                    user4 : on | off | auto;
                    user5 : on | off | auto;
                    user6 : on | off | auto;
                    user7 : on | off | auto;
                }
            }
        }
    }
}
```

Description

The UserSignals wrapper enables you to control the memory port which 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 - user7: on | off | auto;

The user<n> (n=0,...,7) property controls the memory port which 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 which 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 which uses the Function property specified to precharge.

Valid values are as follows:

- on — activates the *Precharge* signal.
- off — deactivates the *Precharge* 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 *Precharge*.

- cas: on | off | auto;

The cas property controls the memory port which uses the Function property specified to cas.

Valid values are as follows:

- on — activates the *CAS* signal.
- off — deactivates the *CAS* 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 Function *CAS*.

- ras: on | off | auto;

The ras property controls the memory port which uses the Function property specified to ras.

Valid values are as follows:

- on — activates the *RAS* signal.
- off — deactivates the *RAS* 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 *RAS*.

- refresh: on | off | auto;

The refresh property controls the memory port which uses the Function property specified to refresh.

Valid values are as follows:

- on — activates the *Refresh* signal.

- off — deactivates the *Refresh* 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 *Refresh*.

- address_select: column | row | auto;

The address_select property controls the row or column values driven onto a multiplexed address port.

Valid values are as follows:

- column — drives the column address values to the address port when multiplexed addressing is used. For non-multiplexed addresses, this Cycle assignment is ignored.
- row — drives the row segment values to the address port when multiplexed addressing is used. For non-multiplexed addresses, this Cycle assignment is ignored.
- 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 allows 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;
                }
            }
        }
    }
}
```

Description

The ConcurrentPortSignals wrapper allows you to activate and deactivate concurrent memory read and write properties to help detect faults specific to multi-port memories. All event sequences 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 allow 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 allows 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:

- Concurrent read is not supported for ROM or 1RW memories.
- 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:

- `on` — activates the concurrent read.

- off — deactivates the concurrent read.
- 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 allowing 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 will be performed when all of the following conditions are present:

- 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.
- 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 will cause 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.

- 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 allows 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:

- Concurrent write is restricted to 2RW, 1R1W and nR2W memories.
- 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:

- **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

- **off** — Deactivates the concurrent write operation.
- **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 allowing 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 | auto;

The read_column_address property allows you to select the column address of the concurrent read operation.

The concurrent read column address has the following limitations:

- The inactive read address is limited to the adjacent column or row.
- read_column_address is not supported for ROM or 1RW memories.
- 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:

- 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.
- off — deactivates the concurrent read column address. The address of the inactive port is controlled by the normal test port logic.
- 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:

- `read_column_address` takes effect only if `Cycle/ConcurrentPortSignals/read_enable` is set to *on*.
- `read_column_address` is ignored if `Core/Memory/ConcurrentRead` property is set to *off*.
- `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.
- 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 allowing concurrent read during any read or write cycle.
- `read_row_address: on | off | auto`;

The `read_row_address` property allows you to select the row address of the concurrent read operation.

The concurrent read row address has the following limitations:

- The inactive read address is limited to the adjacent row.
- `read_row_address` is not supported for ROM or 1RW memories.
- 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:

- `on` — activates the concurrent read row address. Concurrent read inverts the least significant row address bit on the inactive read port.
- `off` — deactivates the concurrent read row address. The address of the inactive port is controlled by the normal test port logic.
- 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:

- `read_row_address` takes effect only if `read_enable` is set to *On*.
- `read_row_address` is ignored if the `ConcurrentRead` property of `Core/Memory` is set to *Off*.
- `read_row_address` is ignored if the memory does not have a row address.
- 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 allowing concurrent read during any read or write cycle.

- write_row_address: on | off | auto;

The write_row_address property allows you to select the concurrent write row address and to activate the concurrent operation.

The concurrent write row address has the following limitations:

- The inactive write address is limited to the adjacent row.
- The concurrent write is restricted to 2RW, 1R1W and n R2W memories. Limiting the memory types will prevent simultaneous write contention.
- 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:

- on — activates the concurrent write row address. Concurrent write inverts the least significant row address bit on the inactive write port.
- off — deactivates the concurrent write row address. The address of the inactive port will be controlled by the normal test port logic.
- 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 will allow concurrent write during any read or write cycle.

- write_column_address: on | off | auto;

The write_column_address property allows you to select the concurrent write column address and to activate the concurrent operation.

The concurrent write column address has the following limitations:

- The inactive write address is limited to the adjacent column.
- The concurrent write is restricted to 2RW, 1R1W, and n R2W memories. Limiting the memory types will prevent simultaneous write contention.
- 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:

- on — activates the concurrent write column address. Concurrent write inverts the least significant column address bit on the inactive write port.
- off — deactivates the concurrent write column address. The address of the inactive port will be controlled by the normal test port logic.

- 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 will allow concurrent write during any read or write cycle.

- `write_data_polarity`: `no_change` | `inverse` | `auto`;

The `write_data_polarity` property allows you to select the data pattern of the concurrent write operation.

Valid values are as follows:

- `no_change` — applies the current active port data to the inactive write port during the concurrent write operation.
- `inverse` — applies the complement of the current active port data to the inactive write port during the concurrent write operation.
- 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:

- 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 will allow concurrent write during any read or write cycle.
- 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
```


Appendix C

MemoryBIST Algorithms

Mentor Graphics 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	500
MemoryBIST Algorithm Detected Faults	500
MemoryBIST Algorithm Test Times	502
Available Library Algorithms	504
SMarch Algorithm	504
ReadOnly Algorithm	506
SMarchCHKB Algorithm	507
SMarchCHKBci Algorithm	510
SMarchCHKBcil Algorithm	513
SMarchCHKBvcd Algorithm	517
LVMarchX Algorithm	529
LVMarchY Algorithm	534
LVMarchCMinus Algorithm	539
LVMarchLA Algorithm	544
LVRowBar Algorithm	550
LVColumnBar Algorithm	554

LVGalPat Algorithm	558
LVGalColumn Algorithm	565
<i>LVGalRow Algorithm</i>	573
LVCheckerboard1X1 Algorithm	580
LVCheckerboard4X4 Algorithm	585
LVWalkingPat Algorithm	590
LVBitSurroundDisturb Algorithm	596
LVAddressInterconnect Algorithm	612
LVDataInterconnect Algorithm	617

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.
- **BR_C** — Begin row cycle.
- **ER_C** — End row cycle.
- **%operationType** — Number of clock cycles specified for an operation.
- **MemRst** — Test phase executed when the following property is *On*:
PatternsSpecification/Patterns/TestStep/MemoryBist/AdvancedOptions/memory_reset
- **Wait** — Optional pause for parallel retention test.
- **PRT** — Test phase executed 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	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	
Stuck-Open Faults	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	
Address Decoder Faults	F	F	F	F	F	F	F	F	F	F	F	F	F	F		F	F	
Transition Faults	F	F	F	F	F	F	F	F	F	F	F	F	F	F		F	F	
Inversion Coupling Faults	F	F	F	F	F	F	F	F	F	F	F	F	F	F		F	F	
Idempotent Coupling Faults	F	F	F	F	F			F	F	F	F	F	F	F		F	F	
Dynamic Coupling Faults	F	F	F	F	F			F	F	F	F	F	F	F		F		
Data Retention Faults		F	F	F	F													
Write Recovery Faults	F	F	F	F	F	F	F										F	
Destructive Read Faults	F	F	F	F	F						F	F	F				F	
Read Disturb Faults	F	F	F	F	F	F	F	F	F	F	F	F	F	F		F		
Single Port Bitline Coupling Faults		G	F	F	F													
Access Transistor Current Leakage Faults				F	F						F	F	F			F	F	
DataPath Shorts					F													
Bit/Group/Global Write Enable Faults					F													
Read Enable Faults						F												
Memory Select Faults						F												
Multiport Synchronous Bitline Coupling Faults	G	G	G	G	F		G	G									G	
Multi-Port Interference Faults	G	G	G	G	F		G	G	G	G	G	G	G	G	G	G	G	

Table Legend:

F — Full coverage
G — Good coverage
 no entry — Low/No coverage

MemoryBIST Algorithm Test Times

The algorithm complexity and a test time calculation are provided in the table for each Tessonnt 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
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 $N_x \times N_y \times N_z$
- **N_x** — Number of rows
- **N_y** — Number of columns
- **N_z** — 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 Application Note MG596384, *Optimizing TMB Custom Algorithms and Operation Sets*, for further information.

During [Parallel Static Retention Testing](#) (PSRT), the algorithms listed in [Table C-3](#) will have shorter test durations due to the reduced number of algorithm test phases that are executed 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 executed 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
SMarchCHKBvcd	8N	0.04	34

Available Library Algorithms

The following describes each algorithm in the Mentor Graphics algorithm library.

SMarch Algorithm	504
ReadOnly Algorithm	506
SMarchCHKB Algorithm	507
SMarchCHKBci Algorithm	510
SMarchCHKBcil Algorithm	513
SMarchCHKBvcd Algorithm	517
LVMarchX Algorithm	529
LVMarchY Algorithm	534
LVMarchCMinus Algorithm	539
LVMarchLA Algorithm	544
LVRowBar Algorithm	550
LVColumnBar Algorithm	554
LVGalPat Algorithm	558
LVGalColumn Algorithm	565
LVGalRow Algorithm	573
LVCheckerboard1X1 Algorithm	580
LVCheckerboard4X4 Algorithm	585
LWWalkingPat Algorithm	590
LVBitSurroundDisturb Algorithm	596
LVAddressInterconnect Algorithm	612
LVDataInterconnect Algorithm	617

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.
2*	1,2	1	(RxW1) (R1W1)	ReadModifyWrite ReadModifyWrite	Scans <i>1</i> s through first word.
3*	3,4	1	(R1W0) (R0Rx)	ReadModifyWrite ReadRead	Scans <i>0</i> s through first word.
4*	5	2 to W Fast row	(W0Rx)	Write	Writes <i>0</i> s in all other words.
5**	6,7	1 to W Fast row	(R0W1) (R1W1)	ReadModifyWrite ReadModifyWrite	Reads <i>0</i> s and replaces with <i>1</i> s.
6***	8,9	1 to w Fast row	(R1W0) (R0W0)	ReadModifyWrite ReadModifyWrite	Reads <i>1</i> s and replaces with <i>0</i> s.
7	10,11	w to 1 Fast row	(R0W1)(R1Rx)	ReadModifyWrite ReadRead	Reads <i>0</i> s and replaces with <i>1</i> s, reverse address sequence. Does back-to-back reads before changing address.
8	10,11	W to 1 Fast row	(R1W0) (R0Rx)	ReadModifyWrite ReadRead	Reads <i>1</i> s and replace with <i>0</i> s. Does back-to-back reads before changing address.
9	12,13	1 to W Fast row	(R0Wx) (RxRx)	ReadModifyWrite ReadRead	Only the first read operation is significant.

Where:

- x* Algorithm phases executed during the PSRT start_to_pause sub-phase
- x** Algorithm phases executed during the PSRT pause_to_pause sub-phase
- x*** Algorithm phases executed during the PSRT pause_to_end sub-phase

Detected Faults

The SMarch 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 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 test bench 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 test bench 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	(R0Wx)	ReadModifyWrite	Only the first read operation is significant.

Where:

- x* Algorithm phases executed during the PSRT start_to_pause sub-phase
- x** Algorithm phases executed during the PSRT pause_to_pause sub-phase
- x*** Algorithm phases executed 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 <i>I</i> s through first word.
3*	3,4	1	(R1W0) (R0Rx)	ReadModifyWrite ReadRead	Scans <i>O</i> s 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 <i>ShadowWrite</i> is <i>On</i> . Writes checkerboard data.
		-	Optional Wait	None	Pauses the test to perform the static retention test.
6**	8	1 to W Fast column	(R0W1)	ReadModifyWrite	Reads checkerboard background. Replaces it with Inverse Checkerboard.

Table C-7. Description of SMarchCHKBci Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
6.5	9	1 to W Fast column	(R1W1)	ReadModifyWrite	Reads Inverse Checkerboard Pattern while <i>ShadowWrite</i> is <i>On</i> , 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 to 1 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.
15	19	w to 1	(R0W0)	ReadModifyWrite	Reads 0 while <i>ShadowWrite</i> is <i>On</i> . Writes 0.

Table C-7. Description of SMarchCHKBci Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
16	20,21	1 Fast row	(RxW1) (R1Rx)	ReadModifyWrite ReadRead	Writes <i>I</i> through the first word. A back-to-back read is performed before changing address.
17	22	2 to W Fast row	(W1Rx)	Write	Writes <i>I</i> to all locations.
18	23	1 to W Fast row	(R1W1)	ReadModifyWrite	Reads <i>I</i> while <i>ShadowWrite</i> is <i>On</i> 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 executed during the PSRT start_to_pause sub-phase
- x** Algorithm phases executed during the PSRT pause_to_pause sub-phase
- x*** Algorithm phases executed 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 <i>I</i> s through first word.
3*	3,4	1	(R1W0) (R0Rx)	ReadModifyWrite ReadRead	Scans <i>0</i> s 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 <i>ShadowWrite</i> is <i>On</i> . Writes checkerboard data.
		-	Optional Wait	None	Pauses the test to perform the static retention test.
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 <i>ShadowWrite</i> is <i>On</i> . Writes Inverse Checkerboard Pattern.

Table C-8. Description of SMarchCHKBcil Test Algorithm per Test Port (cont.)

Phase	Instruction #	Address Sequence	Sequence	Operations Used	Description
		-	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 to 1 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	<p>This phase is composed of three operations. For all operations, the column address is constant.</p> <p>The first set of operations is a Read 0, followed by a Write 1.</p> <p>The second set of operations is to perform two back-to-back Read 0 from a different row (on the same column) while <i>ShadowWrite</i> is <i>On</i>. However, the result of the second read operation is not compared.</p> <p>The third set of operations is to Read a 1 from the current row and then restore its value to 0.</p>
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.

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 <i>ShadowWrite</i> is <i>On</i> . 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 executed during the PSRT start_to_pause sub-phase
- x** Algorithm phases executed during the PSRT pause_to_pause sub-phase
- x*** Algorithm phases executed during the PSRT pause_to_end sub-phase

Detected Faults

The SMarchCHKBcil 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 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 datapath 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	<p>This phase is used to detect datapath shorts.</p> <p>The same pattern is written to all words with no data mapping applied.</p> <p>Step 1 — (OP5) Write checkerboard to the datapath and read it back for all words in the first row.</p> <p>Step 2 — (OP5) Write inverse checkerboard to the datapath and read it back for all words in the first row.</p>

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)	OP5,OP6 OP5,OP7 OP5,OP5 OP5,OP8	<p>This phase is used to detect bit/group write enable faults.</p> <p>Repeat all groups of operations for all words in the first row.</p> <p>Step 1 — (OP5) Write and read 0s for the word.</p> <p>Step 2 * (OP6) Write 1s while Even Group Write Enables are <i>On</i> followed by reading data pattern 0101 ... 0101.</p> <p>Step 3 — (OP5) Write and read 0s for the word.</p> <p>Step 4 * (OP7) Write 1s while Odd Group Write Enables are <i>On</i> followed by reading data pattern 1010 ... 1010.</p> <p>Step 5 — (OP5) Write and read 0s for the word.</p> <p>Step 6 — (OP5) Write 1s while all Group Write Enables are <i>On</i> followed by reading 1s.</p> <p>Step 7 — (OP5) Write and read 1s for the word.</p> <p>Step 8 * (OP8) Attempt to write 0s while all Group Write Enables are <i>Off</i> followed by reading 1s.</p> <p>Note: Steps marked with * indicate that memories without the feature under test are disabled to avoid corruption of their content.</p>

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 1 to C, Row 2 1 to C, Row 1 1 to C, Row 2 1 to C, Row 2 1 to C, Row 1	(W1Rx) (W0Rx) (R1Rx) (R1Rx) (R0Rx) (R0Rx) Idle	OP1 OP1 OP2 OP9 OP2 OP9 OP0	<p>This phase is used to detect Read Enable stuck-active faults.</p> <p>Step 1 — (OP1) Write <i>1s</i> to all words in the first row.</p> <p>Step 2 — (OP1) Write <i>0s</i> to all words in the second row.</p> <p>Step 3 — (OP2) Read <i>1s</i> from all words in the first row.</p> <p>Step 4 * (OP9) Attempt to read <i>0s</i> from all words in the second row while Read Enable is <i>Off</i>.</p> <p>Step 5 — (OP2) Read <i>0s</i> from all words in the second row.</p> <p>Step 6 * (OP9) Attempt to read <i>1s</i> from all words in the first row while Read Enable is <i>Off</i>.</p> <p>Note: Steps marked with * indicate that memories without the feature under test are disabled to avoid corruption of their content.</p>

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 <i>Off</i> . 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 <i>Off</i> . Step 6 — (OP2) Read 0s from all words in the second row. Note: Steps marked with * indicate that memories without the feature under test are disabled to avoid corruption of their content.
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	<p>This phase is only useful for multi-port memories.</p> <p>Step 1 — (OP10) Read checkerboard and write inverse checkerboard data.</p> <p>Step 2 — (OP12) Write checkerboard data restoring the original background and reading it back.</p> <p>During the Read portion of the operation, a <i>write_column_address</i> operation is performed on the adjacent column of inactive write port. The operation is not performed for memories without column address bits.</p> <p>During the Write portion of the operation, a <i>read_column_address</i> 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.</p>
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	<p>Step 1 — (OP1) Write checkerboard data and read it back.</p> <p>Step 2 — (OP3) Read checkerboard data and write it back before changing the address.</p> <p>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.</p>
5.5	35-36	1 to W Fast column	(R0W0)	OP11	<p>Read checkerboard background while ShadowWrite is <i>On</i>, and ShadowRead is <i>Off</i> (Read is performed at the address specified by the test algorithm from all inactive R/W ports).</p> <p>Write checkerboard with ShadowRead <i>On</i> and ShadowWrite <i>Off</i>.</p>
			Optional Wait	None	Pause the test to perform the static retention test.
6**	37-38	1 to W Fast column	(R0Rx)	OP2	Read checkerboard background data followed by a read operation whose data out will not be compared.
6.1**	39-40	1 to W Fast column	(R0W1)	OP3	Read checkerboard background while ShadowRead is <i>Off</i> . Replace it with inverse checkerboard data while ShadowRead is <i>On</i> . ShadowWrite is <i>Off</i> 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	<p>This phase is only useful for multi-port memories.</p> <p>Step 1 — (OP10) Read inverse checkerboard and write checkerboard data.</p> <p>Step 2 — (OP12) Write inverse checkerboard data restoring the original background and reading it back.</p> <p>During the Read portion of the operation, a <i>write_column_address</i> operation is performed on the adjacent column of inactive write port. The operation is not performed for memories without column address bits.</p> <p>During the Write portion of the operation, a <i>read_column_address</i> 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.</p>
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	<p>Step 1 — (OP1) Write inverse checkerboard data and read it back.</p> <p>Step 2 — (OP3) Read inverse checkerboard data and write it back before changing the address.</p> <p>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.</p>
6.5	46-47	1 to W Fast column	(R1W1)	OP11	<p>Read inverse checkerboard background while ShadowWrite is <i>On</i>, and ShadowRead is <i>Off</i> (Read is performed at the address specified by the test algorithm from all inactive R/W ports). Write inverse checkerboard with ShadowRead <i>On</i> and ShadowWrite <i>Off</i>.</p>
			Optional Wait	None	Pause the test to perform the static retention test.
7***	48-49	1 to W Fast column	(R1Rx)	OP2	Read inverse checkerboard background data followed by a read operation whose data out will not be compared.
7.1	50-51	1 to W Fast column	(R1W0)	OP3	Read inverse checkerboard data while ShadowRead is <i>Off</i> . Replace it with checkerboard data while ShadowRead is <i>On</i> .
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)	OP3 OP3 OP3	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.

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)	OP3 OP3 OP3	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 executed during the PSRT start_to_pause sub-phase
- x** Algorithm phases executed during the PSRT pause_to_pause sub-phase
- x*** Algorithm phases executed 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
OP10	ReadModifyWrite_Column_ShadowWriteRead

Table C-10. Mapping of Operation Code to Operation Name (cont.)

Code	Operation Name
OP11	ReadModifyWrite_Row_ShadowWriteRead
OP12	WriteRead_Column_ShadowReadWrite

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 Mentor Graphics 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.

The SMarchCHKBvcd algorithm can, in rare cases, produce incorrect results during repair analysis. The problem occurs when all of the following conditions are present:

- Row repair is used and spare row blocks are composed of a single row, OR column repair is used and the spare column blocks are composed of a number of spare columns that is less than the number of column addresses. The problem is NOT present when using IO repair, either alone or with row repair.
- **DataOutHoldWithInactiveReadEnable** or **MemoryHoldWithInactiveSelect** is set to *On*. This means that at least one of the specialized tests on the read enable (phase 3.6) or chip select (phase 3.7) ports is applied.
- A fault is present in either the first or second row at the last column address of a bank. The specialized tests are applied to the first two rows of each bank.

The fault appears as if it were present at all addresses of the row that does not contain the fault. The memory might be incorrectly declared as non-repairable.

To avoid this problem, use a different memory test algorithm for repair analysis (pre-repair). Then, for post-repair testing, you can use the SMarchCHKBvcd algorithm.

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 D-data incrementing from address minimum to address maximum.
3. Read 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:
 - The X1 address segment is incrementing and has reached the maximum OR.
 - The X1 address segment is decrementing and has reached the minimum.

- The bank address segment Z counts when instructed and s carry out from the Y1 address segment is generated. An y1_carry_out is generated when:
 - The Y1 address segment is incrementing and has reached the maximum AND a x1_carry_out is generated OR.
 - 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	RDWD	ReadModifyWrite	Read D-data and Write data D-data. After all addresses have been accessed branch to instruction 1 and repeat one time as follows with RepeatLoopA: <ul style="list-style-type: none"> • 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	RDWD	ReadModifyWrite	Read 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.

Figure C-1. LVMarchX Example Algorithm Wrapper

```
Algorithm (LVMarchX) {
    TestRegisterSetup {
        operation_set_select : ReadLatency_RL11;
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 4'b0000;
            load_column_address_min : 6'b000000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : y1_carry_out;
                y1_carry_in : x1_carry_out;
                x1_carry_in : none;
            }
        }
        DataGenerator {
            load_write_data : 8'b00000000;
            load_expect_data : 8'b00000000;
        }
    }
    MicroProgram {
        Instruction (M0_W0) {
            operation_select : Write;
            AddressCommands {
                address_select : select_s;
                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;
                inhibit_data_compare : on;
            }
            NextConditions {
```

```
z_end_count : on;
x1_end_count : on;
y1_end_count : on;
RepeatLoopA {
    branch_to_instruction : M1_R0_W1;
    Repeat {
        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;
    }
    DataCommands {
        expect_data : data_reg;
        inhibit_data_compare : on;
    }
    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.
- 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 D-data, and read D-data incrementing from address minimum to address maximum.
3. Read 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:
 - The X1 address segment is incrementing and has reached the maximum OR
 - 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:
 - The Y1 address segment is incrementing and has reached the maximum AND a `x1_carry_out` is generated OR.
 - 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	-	-	-	RDWD	ReadModifyWrite	Read D-data and Write data D-data.
	2	-	-	min to max, fast row	RD	Read	Read 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: <ul style="list-style-type: none">• Repeat #1 - repeat instructions with inverted the address sequencing, write data sequencing, and expect data sequencing
2	1	Repeat #1	-	-	RDWD	ReadModifyWrite	Read D-data and Write data 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.

Example Algorithm Wrapper

[Figure C-2](#) illustrates the Algorithm wrapper for the example memory.

Figure C-2. LVMarchY Example Algorithm Wrapper

```

Algorithm (LVMarchY) {
    TestRegisterSetup {
        operation_set_select : ReadLatency_RL11;
        AddressGenerator
            load_bank_address_min : 2'b00;
            load_row_address_min : 4'b0000;
            load_column_address_min : 6'b000000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : y1_carry_out;
                y1_carry_in : x1_carry_out;
                x1_carry_in : none;
            }
        }
        DataGenerator {
            load_write_data : 8'b00000000;
            load_expect_data : 8'b00000000;
        }
    }
    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;
                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;
    }
    DataCommands {
        expect_data : inverse_data_reg;
        inhibit_data_compare : on;
    }
    branch_to_instruction : M1_R0_W1;
    NextConditions {
        z_end_count : on;
        x1_end_county1_end_count : on;
        y1_end_count : on;
        RepeatLoopA {
            branch_to_instruction : M1_R0_W1;
            Repeat {
                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;
    }
    DataCommands {
        expect_data : data_reg;
        inhibit_data_compare : on;
    }
    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.
- 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 D-data incrementing from address minimum to address maximum.
3. Read D-data and write D-data incrementing from address minimum to address maximum.
4. Read D-data and write D-data decrementing from address maximum to address minimum.
5. Read 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:
 - The X1 address segment is incrementing and has reached the maximum OR.
 - The X1 address segment is decrementing and has reached the minimum.
- The bank address segment Z counts when instructed and s carry out from the Y1 address segment is generated. An y1_carry_out is generated when:
 - The Y1 address segment is incrementing and has reached the maximum AND a x1_carry_out is generated OR.
 - 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	RDWD	ReadModifyWrite	Read D-data and Write data D-data. After all addresses have been accessed branch to instruction 1 and repeat three times as follows with RepeatLoopA: <ul style="list-style-type: none"> • 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	RDWD	ReadModifyWrite	Read D-data and Write data D-data.
3	1	Repeat #2	-	max to min, fast row	RDWD	ReadModifyWrite	Read D-data and Write data D-data.
4	1	Repeat #3	-	max to min, fast row	RDWD	ReadModifyWrite	Read D-data and Write data 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.

Figure C-3. LVMarchCMinus Example Algorithm Wrapper

```
Algorithm (LVMarchCMinus) {
    TestRegisterSetup {
        operation_set_select : ReadLatency_RL11;
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 4'b0000;
            load_column_address_min : 6'b000000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : y1_carry_out;
                y1_carry_in : x1_carry_out;
                x1_carry_in : none;
            }
        }
        DataGenerator {
            load_write_data : 8'b00000000;
            load_expect_data : 8'b00000000;
        }
    }
    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;
    Repeat {
        write_data_sequence : inverse;
        expect_data_sequence : inverse;
        inhibit_last_address_count : on;
    }
    Repeat {
        address_sequence : inverse;
    }
    Repeat {
        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;
    }
    DataCommands {
        expect_data : data_reg;
        inhibit_data_compare : on;
    }
    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.
- 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 D-data, write D-data, write D-data, and read D-data incrementing from address minimum to address maximum.
3. Read D-data, write D-data, write D-data, write D-data, and read D-data incrementing from address minimum to address maximum.
4. Read D-data, write D-data, write D-data, write D-data, and read D-data decrementing from address maximum to address minimum.
5. Read D-data, write D-data, write 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:
 - The X1 address segment is incrementing and has reached the maximum OR.
 - The X1 address segment is decrementing and has reached the minimum.
- The bank address segment Z counts when instructed and s carry out from the Y1 address segment is generated. An y1_carry_out is generated when:
 - The Y1 address segment is incrementing and has reached the maximum AND a x1_carry_out is generated OR.
 - 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	-	-	-	RDW D	ReadModifyWrite	Read D-data and Write data D-data.
	2	-	-		WD	Write	Write D-data.
	3	-	-	min to max, fast row	WDR D	WriteRead	<p>Write D-data and Read data D-data. Branch back to Instruction 1 until all addresses are accessed.</p> <p>After all addresses have been accessed branch to instruction 1 and repeat three times as follows with RepeatLoopA:</p> <ul style="list-style-type: none"> • 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	-	-	RDW D	ReadModifyWrite	Read D-data and Write data D-data.
	2	Repeat #1	-		WD	Write	Write D-data.
	3	Repeat #1	-	min to max, fast row	WDR D	WriteRead	<p>Write D-data and Read data D-data. Branch back to Instruction 1 until all addresses are accessed.</p>

Table C-14. Description of LVMarchLA Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
3	1	Repeat #2	-	-	RDW D	ReadModifyWrite	Read D-data and Write data D-data.
	2	Repeat #2	-		WD	Write	Write D-data.
	3	Repeat #2	-	max to min, fast row	WDR D	WriteRead	Write D-data and Read data D-data. Branch back to Instruction 1 until all addresses are accessed.
4	1	Repeat #3	-	-	RDW D	ReadModifyWrite	Read D-data and Write data D-data.
	2	Repeat #3	-	-	WD	Write	Write D-data.
	3	Repeat #3	-	max to min, fast row	WDR D	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.

Figure C-4. LVMarchLA Example Algorithm Wrapper

```
Algorithm (LVMarchLA) {
    TestRegisterSetup {
        operation_set_select LVMarchLA : ReadLatency_RL11;
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 4'b0000;
            load_column_address_min : 6'b0000000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : y1_carry_out;
                y1_carry_in : x1_carry_out;
                x1_carry_in : none;
            }
        }
        DataGenerator {
            load_write_data : 8'b00000000;
            load_expect_data : 8'b00000000;
        }
    }
    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;
            Repeat {
                write_data_sequence : inverse;
                expect_data_sequence : inverse;
                inhibit_last_address_count : on;
            }
            Repeat {
                address_sequence : inverse;
            }
            Repeat {
                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;
    }
    DataCommands {
        expect_data : data_reg;
        inhibit_data_compare : on;
    }
    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.
- 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 D-data to odd columns incrementing from address minimum to address maximum for a single bank.
2. Read D-data from even columns and D-data from odd columns incrementing from address minimum to address maximum for a single bank.
3. Re-execute 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:

$$Z \quad X1 \leftarrow Y1$$

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:
 - The Y1 address segment is incrementing and has reached the maximum OR.
 - 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 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, D_o	Write	Write background of D-data to even column addresses and D-data to odd column addresses.

Table C-15. Description of LVRowBar Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
1	1	-	-	max to min, fast column	RD_e, D_o	Read	Read background of D-data from even column addresses and D-data from odd column 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: <ul style="list-style-type: none">• Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing
3	0	-	Repeat #1	min to max, fast column	WD_e, D_o	Write	Write background of D-data to even column addresses and D-data to odd column addresses.
4	1	-	Repeat #1	max to min, fast column	RD_e, D_o	Read	Read background of 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.

Figure C-5. LVRowBar Example Algorithm Wrapper

```

Algorithm (LVRowBar) {
    TestRegisterSetup {
        operation_set_select : ReadLatency_RL11;
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 4'b0000;
            load_column_address_min : 6'b000000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                x1_carry_in : y1_carry_out;
                y1_carry_in : none;
            }
        }
        DataGenerator {
            InvertDataWithColumnBit : c[0];
            load_write_data : 8'b00000000;
            load_expect_data : 8'b00000000;
        }
    }
    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;
            Repeat {
                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.
- 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 D-data to odd rows incrementing from address minimum to address maximum for a single bank.

2. Read D-data from even rows and D-data from odd rows incrementing from address minimum to address maximum for a single bank.
3. Re-execute 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:

$$Z \quad Y1 \Leftarrow X1$$

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:
 - The X1 address segment is incrementing and has reached the maximum OR.
 - 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 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, D_o	Write	Write background of D-data to even row addresses and D-data to odd row addresses.
1	1	-	-	max to min, fast row	RD_e, D_o	Read	Read background of D-data from even row addresses and D-data from odd row addresses.
2	2	-	-	min to max, bank addresses	-	NoOperation	<p>Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses.</p> <p>After all bank addresses have been accessed branch to instruction 0 and repeat one time as follows with RepeatLoopB:</p> <ul style="list-style-type: none"> • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing
3	0	-	Repeat #1	min to max, fast row	WD_e, D_o	Write	Write background of D-data to even row addresses and D-data to odd row addresses.
4	1	-	Repeat #1	max to min, fast row	RD_e, D_o	Read	Read background of D-data from even row addresses and D-data from odd row addresses.
5	2	-	Repeat #1	min to max, bank addresses	-	NoOperation	<p>Increment the Bank Address counting from min to max. Branch back to Instruction 0 and repeat the test for all bank addresses.</p>

Example Algorithm Wrapper

Figure C-6 illustrates the *Algorithm* wrapper for the example memory.

Figure C-6. LVColumnBar Example Algorithm Wrapper

```

Algorithm (LVColumnBar) {
    TestRegisterSetup {
        operation_set_select : ReadLatency_RL11;
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 4'b0000;
            load_column_address_min : 6'b000000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                y1_carry_in : x1_carry_out;
                x1_carry_in : none;
            }
        }
        DataGenerator {
            InvertDataWithRowBit : r[0];
            load_write_data : 8'b00000000;
            load_expect_data : 8'b00000000;
        }
    }
    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;
            Repeat {
                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.
- 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 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 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-execute 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-execute 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:

$$Z \quad Y1 \Leftarrow X1$$

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:
 - The X1 address segment is incrementing and has reached the maximum OR.
 - 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 Loop A	Repeat Loop B	Address Sequence	Sequence	Operation	Description
0	0	-	-	min to max, fast row	WD	Write	Write background of D-data.
1	1	-	-	-	WD	Write	Write D-data to location addressed by AddressRegisterA.
2	2	-	-	-	RD	Read	Read 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	RDWD	ReadModifyWrite	Read 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-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	<p>Increment the Bank Address counting from min to max.</p> <p>Branch back to Instruction 0 and repeat the test for all bank addresses.</p> <p>After all bank addresses have been accessed branch to instruction 0 and repeat one time as follows with RepeatLoopB:</p> <ul style="list-style-type: none"> • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing.
0	0	-	Repeat #1	min to max, fast row	WD	Write	Write background of D-data.
1	1	-	Repeat #1	-	WD	Write	Write 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 D-data to location addressed by AddressRegisterA.
	3	-	Repeat #1	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	-	Repeat #1	AddressRegisterA min to max, fast row	RDWD	ReadModifyWrite	Read 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

Figure C-7 illustrates the *Algorithm* wrapper for the example memory.

Figure C-7. LVGALPAT Example Algorithm Wrapper

```

Algorithm (LVGALPAT) {
    TestRegisterSetup {
        operation_set_select : ReadLatency_RL11;
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 4'b0000;
            load_column_address_min : 6'b000000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                y1_carry_in : x1_carry_out;
                x1_carry_in : none;
            }
            AddressRegisterB {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                y1_carry_in : x1_carry_out;
                x1_carry_in : none;
            }
        }
        DataGenerator {
            load_write_data : 8'b00000000;
            load_expect_data : 8'b00000000;
        }
    }
    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;
    Repeat {
        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.
- 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 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 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-execute 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-execute 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:

$$Z \quad X1 \leftarrow Y1$$

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:
 - The Y1 address segment is incrementing and has reached the maximum OR.
 - 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 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	-	-	-	WD	Write	Write D-data to location addressed by AddressRegisterA.
2	2	-	-	-	RD	Read	Read 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 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	RDWD	ReadModifyWrite	Read 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 LVGAIColumn Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
3	6	-	-	min to max, bank addresses	-	NoOperation	<p>Increment the Bank Address counting from min to max.</p> <p>Branch back to Instruction 0 and repeat the test for all bank addresses.</p> <p>After all bank addresses have been accessed branch to instruction 0 and repeat one time as follows with RepeatLoopB:</p> <ul style="list-style-type: none"> • 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	-	WD	Write	Write D-data to location addressed by AddressRegisterA.

Table C-18. Description of LVGAIColumn Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
6	2	-	Repeat #1	-	RD	Read	Read 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 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	RDWD	ReadModifyWrite	Read 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

Figure C-8 illustrates the *Algorithm* wrapper for the example memory.

Figure C-8. LVGALColumn Example Algorithm Wrapper

```
Algorithm (LVGALColumn) {
    TestRegisterSetup {
        operation_set_select : ReadLatency_RL11;
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 4'b0000;
            load_column_address_min : 6'b000000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                x1_carry_in : y1_carry_out;
                y1_carry_in : none;
            }
            AddressRegisterB {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                y1_carry_in : none;
                x1_carry_in : none;
            }
        }
        DataGenerator {
            load_write_data: 8'b00000000;
            load_expect_data: 8'b00000000;
        }
    }
    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;
        Repeat {
            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.
- 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 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 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-execute 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-execute 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:

$$Z \quad Y1 \Leftarrow X1$$

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:
 - The X1 address segment is incrementing and has reached the maximum OR.
 - 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

[Table C-19](#) 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	-	-	-	WD	Write	Write D-data to location addressed by AddressRegisterA.

Table C-19. Description of LVGAIRow Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
2	2	-	-	-	RD	Read	Read 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 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	RDWD	ReadModifyWrite	Read 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: <ul style="list-style-type: none">• 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	-	WD	Write	Write D-data to location addressed by AddressRegisterA.
6	2	-	Repeat #1	-	RD	Read	Read 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 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	RDWD	ReadModifyWrite	Read 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 LVGAIRow Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
7	6	-	Repeat #1	min to max, bank addresses	-	NoOperation	<p>Increment the Bank Address counting from min to max.</p> <p>Branch back to Instruction 0 and repeat the test for all bank addresses.</p> <p>After all bank addresses have been accessed repeat phase 0 and 1 with inverted write and expect data.</p>

Example Algorithm Wrapper

Figure C-9 illustrates the *Algorithm* wrapper for the example memory.

Figure C-9. LVGALRow Example Algorithm Wrapper

```
Algorithm (LVGALRow) {
    TestRegisterSetup {
        operation_set_select : ReadLatency_RL11;
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 4'b0000;
            load_column_address_min : 6'b000000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                y1_carry_in : x1_carry_out;
                x1_carry_in : none;
            }
            AddressRegisterB {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                y1_carry_in : none;
                x1_carry_in : none;
            }
        }
        DataGenerator {
            load_write_data: 8'b00000000;
            load_expect_data: 8'b00000000;
        }
    }
    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;
        Repeat {
            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.
- 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:

$$Z \quad X1 \leftarrow Y1$$

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:
 - The Y1 address segment is incrementing and has reached the maximum OR.
 - 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
- D when column address bit 0 = 1'b1 and row address bit 0 = 1'b0
- 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.
2	5	-	-	min to max, bank addresses	-	NoOperation	<p>Increment the Bank Address counting from min to max.</p> <p>Branch back to Instruction 0 and repeat the test for all bank addresses.</p> <p>After all bank addresses have been accessed branch to instruction 0 and repeat one time as follows with RepeatLoopB:</p> <ul style="list-style-type: none"> • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing.
3	0	Repeat #1	-	min to max, fast row	WD	Write	Write 1X1 checkerboard D-data background.
4	1	Repeat #1	-	min to max, fast row	RD	Read	Read 1x1 checkerboard D-data.
5	5	Repeat #1	-	min to max, bank addresses	-	NoOperation	<p>Increment the Bank Address counting from min to max.</p> <p>Branch back to Instruction 0 and repeat the test for all bank addresses.</p>

Example Algorithm Wrapper

Figure C-10 illustrates the *Algorithm* wrapper for the example memory.

Figure C-10. LVCheckerboard1X1 Example Algorithm Wrapper

```

Algorithm (LVCheckerboard1X1) {
    TestRegisterSetup {
        operation_set_select : ReadLatency_RL11;
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 4'b0000;
            load_column_address_min : 6'b000000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                x1_carry_in : y1_carry_out;
                y1_carry_in : none;
            }
        }
        DataGenerator {
            InvertDataWithRowBit: r[0];
            InvertDataWithColumnBit: c[0];
            load_write_data: 8'b00000000;
            load_expect_data: 8'b00000000;
        }
    }
    MicroProgram {
        Instruction (W_1X1_CHECKERBOARD) {
            operation_select : Write;
            AddressCommands {
                address_select : select_a;
                x1_address : increment;
                y1_address : increment;
            }
            DataCommands {
                write_data : zero;
            } 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;
            Repeat {
                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.
- 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:

$$Z \quad X1 \Leftarrow Y1$$

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:
 - The Y1 address segment is incrementing and has reached the maximum OR.
 - 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
- D when column address bit 1 = 1'b1 and row address bit 1 = 1'b0
- 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: <ul style="list-style-type: none"> • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing.
3	0	Repeat #1	-	min to max, fast row	WD	Write	Write 4X4 checkerboard D-data background.
4	1	Repeat #1	-	min to max, fast row	RD	Read	Read 4X4 checkerboard D-data.

Table C-21. Description of LVCheckerboard4X4 Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
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.

Figure C-11. LVCheckerboard4X4 Example Algorithm Wrapper

```
Algorithm (LVCheckerboard4X4) {
    TestRegisterSetup {
        operation_set_select : ReadLatency_RL11;
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 4'b0000;
            load_column_address_min : 6'b000000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                x1_carry_in : y1_carry_out;
                y1_carry_in : none;
            }
        }
        DataGenerator {
            InvertDataWithRowBit: r[1];
            InvertDataWithColumn Bit: c[1];
            load_write_data: 8'b00000000;
            load_expect_data: 8'b00000000;
        }
    }
    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;
            Repeat {
                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.
- 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:

$$Z \quad Y1 \Leftarrow X1$$

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:
 - The X1 address segment is incrementing and has reached the maximum OR.
 - 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	-	-	-	WD	Write	Write 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 D-data when AddressRegisterB equal AddressRegisterA.
	3	-	-	AddressRegisterB min to max, fast row	RDWD	ReadModifyWrite	Read 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.

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	<p>Increment the Bank Address counting from min to max.</p> <p>Branch back to Instruction 0 and repeat the test for all bank addresses.</p> <p>After all bank addresses have been accessed branch to instruction 0 and repeat one time as follows with RepeatLoopB:</p> <ul style="list-style-type: none"> • Repeat #1 - repeat instructions with inverted write data sequencing and expect data sequencing.
3	0	-	Repeat #1	min to max, fast row	WD	Write	Write background of 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	RD	Read	Read D-data from the location addressed by AddressRegisterB. Read D-data when AddressRegisterB equal AddressRegisterA.
	3	-	Repeat #1	AddressRegisterB min to max, fast row	RDWD	ReadModifyWrite	<p>Read D-data and Write D-data to the location addressed by AddressRegisterA.</p> <p>Branch back to Instruction 1 for all AddressRegisterA row and column addresses.</p>

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	<p>Increment the Bank Address counting from min to max.</p> <p>Branch back to Instruction 0 and repeat the test for all bank addresses.</p> <p>After all bank addresses have been accessed repeat phase 0,1, and 2 with inverted write and expect data.</p>

Example Algorithm Wrapper

Figure C-12 illustrates the *Algorithm* wrapper for the example memory.

Figure C-12. LWWalkingPat Example Algorithm Wrapper

```
Algorithm (LWWalkingPat) {
    TestRegisterSetup {
        operation_set_select : ReadLatency_RL11;
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 4'b0000;
            load_column_address_min : 6'b000000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                y1_carry_in : x1_carry_out;
                x1_carry_in : none;
            }
            AddressRegisterB {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                y1_carry_in : x1_carry_out;
                x1_carry_in : none;
            }
        }
        DataGenerator {
            load_write_data: 8'b00000000;
            load_expect_data: 8'b00000000;
        }
    }
    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 {
```


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.
- 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 *on*, 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 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:
 - The X1 address segment is incrementing and has reached the maximum OR.
 - 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 X1 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 LVBBitSurroundDisturb algorithm sequence.

Table C-23. Description of LVBBitSurroundDisturb 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-23. Description of LVBitSurroundDisturb Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
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	WD	Write	Write 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	WD	Write	Write 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: <ul style="list-style-type: none"> • 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	AddressRegisterB, decrement column address	WD	Write	Write 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	WD	Write		Write 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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-execute 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-execute phases 1 through 7 for all bank addresses. Repeat instructions 0-7 once as follows with RepeatLoopA: <ul style="list-style-type: none"> • 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	WD	Write	Write background of 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	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, increment row address	WD	Write	Write D-data at the location addressed by AddressRegisterB. Increment 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 counts from zero to CounterA endcount (twice). Repeat instructions 2-5 three times as follows with RepeatLoopB: <ul style="list-style-type: none"> • 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	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	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	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: <ul style="list-style-type: none"> • 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	WD	Write	Write D-data at the location addressed by AddressRegisterB. Increment AddressRegisterB column address by one.
	3	Repeat #1	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 #1	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 #1	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: <ul style="list-style-type: none"> • 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	WD	Write	Write D-data at the location addressed by AddressRegisterB. Decrement AddressRegisterB column address by one.
	3	Repeat #1	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 #1	Repeat #3	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 #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: <ul style="list-style-type: none"> • 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-execute 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-execute phases 1 through 7 for all bank addresses. Repeat instructions 0-7 once as follows with RepeatLoopA: <ul style="list-style-type: none">• Repeat #1 - branch to instruction 0 and repeat instructions with inverted the write and expect data sequencing

Example Algorithm Wrapper

Figure C-13 illustrates the *Algorithm* wrapper for the example memory.

Figure C-13. LVBitSurroundDisturb Example Algorithm Wrapper

```

Algorithm (LVBitSurroundDisturb) {
    TestRegisterSetup {
        operation_set_select : ReadLatency_RL11;
        LoadCounterA_EndCount : 1;
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 4'b0000;
            load_column_address_min : 6'b000000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                y1_carry_in : x1_carry_out;
                x1_carry_in : none;
            }
            AddressRegisterB {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_y0_bits : 0;
                number_x0_bits : 0;
                z_carry_in : none;
                y1_carry_in : none;
                x1_carry_in : none;
            }
        }
        DataGenerator {
            load_write_data: 8'b00000000;
            load_expect_data: 8'b00000000;
        }
    }
    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;
    CounterCommands {
        counter_a : increment;
    }
    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;
    CounterCommands {
        counter_a : increment;
    }
    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;
    }
}
```

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.
- 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.

LVAddressInterconnect Algorithm

The LVAddressInterconnect algorithm provides coverage of defects that occur on the board-level memory address bus lines: stuck-at, open, delay, and bridging faults. The interconnect algorithm is very fast, typically requiring less than a millisecond. LVAddressInterconnect is performed as follows:

1. Write 0 in low or the base cell.
2. Write 1 to the address where the first address line is logic 1 and the rest is logic zero.
3. Read 0, then write 0 in the base cell.
4. Walk a 1 across the address lines (in a field of zeros) and repeat steps 2 and 3.
5. Walk a 0 across the address lines (in a field of ones) and repeat steps 2 and 3.

Note



Step 5 can be applied only if the memory uses the full address range.

The LVAddressInterconnect algorithm is described in [Table C-24](#).

Test Length

Assuming that the memory has a full address range, the test length of this algorithm is:

$$1 + 6 \log_2 N_{xyz}$$

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 hold the address for the base cell, which is normally the low address.

AddressRegisterB

The AddressRegisterB segments are configured to hold the binary value <#of AddressLines‘b1>.

CounterA EndCount Register

The Counter A EndCount register is used to count the steps to walk the one or the zero across the address bus and should be loaded with the value equal to <AddressBusWidth> - 1.

Algorithm Sequence

[Table C-24](#) describes the LVAddressInterconnect algorithm sequence.

Table C-24. Description of LVAddressInterconnect Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
1	0	-	-	Base cell address	W0	Write	Write zero in the base cell.
2	1	-	-	One address line only is set to logic 1	W1	Write	Write one to the address specified.
3	2	-	-	Base cell address	R0W0	ReadModifyWrite	Read zero from base cell address, then write zero. This operation will pass if there is no stuck-at or stuck-open fault on the address line set to logic 1 in instruction 1. Walk the logic 1 across the address lines and repeat from instruction 1.
4	3	-	-	Set the address for walk zero	-	NoOperation	Set the AddressRegister for walk logic zero.
5	4	-	-	One address line only set to logic 0.	-	NoOperation	Repeat from instruction 2.

Example Algorithm Wrapper

Figure C-14 illustrates the *Algorithm* wrapper for the example memory.

Figure C-14. LVAddressInterconnect Example Algorithm Wrapper

```

Algorithm (LVAddressInterconnect) {
    TestRegisterSetup {
        operation_set_select : ReadLatency_RL11;
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 3'b000;
            load_column_address_min : 3'b000;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                z_carry_in : none;
                y1_carry_in : none;
                x1_carry_in : none;
            }
            load_column_address_min : 3'b001;
            AddressRegisterB {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                z_carry_in : none;
                y1_carry_in : none;
                x1_carry_in : none;
            }
        } //End of AddressGenerator wrapper
        LoadCounterA_EndCount : 7;
        DataGenerator {
        }
    } //End of TestRegisterSetup wrapper
    MicroProgram {
        Instruction (WRITE0_BASE_CELL) {
            operation_select : Write;
            AddressCommands {
                address_select : select_a;
            }
            DataCommands {
                write_data : zero;
            }
            NextConditions {
            }
        }
        Instruction (WRITE1_BASE_CELL) {
            operation_select : Write;
            AddressCommands {
                address_select : select_b;
            }
            DataCommands {
                write_data : one;
            }
            NextConditions {
            }
        }
        Instruction (READ0_WRITE_0_BASE_CELL) {
            operation_select : ReadModifyWrite;
            AddressCommands {

```

```

        address_select : Select_A_Rotate_B;
    }
    CounterCommands {
        counter_a : increment;
    }
    branch_to_instruction : WRITE0_BASE_CELL;
    NextConditions {
        counter_a_end_count : on;
    }
}
Instruction (SET_ADDRESS_WALK0) {
    operation_select : NoOperation;
    AddressCommands {
        address_select : Select_A_Copy_To_B;
        y1_address : decrement;
        x1_address : decrement;
        z_address : decrement;
    }
    NextConditions {
    }
}
Instruction (EXECUTE_WALK0) {
    operation_select : NoOperation;
    address_select : select_b;
    y1_address : decrement;
    NextConditions {
        RepeatLoopA {
            branch_to_instruction : WRITE0_BASE_CELL;
            Repeat {
            }
        }
    }
}
} // End of MicroProgram wrapper
} // End of Algorithm (LVAddressInterconnect) wrapper

```

Fault Coverage

The faults detected by the LVAddressInterconnect algorithm are identified in [Table C-25](#).

Table C-25. LVAddressInterconnect Algorithm Fault Coverage

Fault Type	Fault Coverage
Stuck-At Fault on a single address line at either a logic 1 or logic 0.	Yes
Stuck-Open Faults on address line. A stuck-open fault is a single data line that contains an open (break).	Yes
Bridging Faults. A bridging fault is two or more address lines shorted together. The short can result in either a Wired-AND (0 dominant) or Wired_OR (1 dominant).	Yes

Specification

To test SRAMs using the LVAddressInterconnect algorithm, specify **Algorithm** LVAddressInterconnect 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 LVAddressInterconnect algorithm:

- A full binary count value for the min and max should be assigned to CountRange(BankAddress), CountRange(RowAddress), and/or CountRange(ColumnAddress) in the memory TCD file if coverage of the Wired-OR fault model is desired for address-line shorts.
- All the physical address lines cannot be mapped to bank address only. The memory configuration should have at least two logical addresses.

LVDataInterconnect Algorithm

The LVDataInterconnect algorithm provides coverage of defects that occur on the board-level memory data bus lines: stuck-at, open, delay, and bridging faults. The interconnect algorithm is very fast, typically requiring less than a millisecond. LVDataInterconnect is performed as follows:

1. Initialize the minimum and maximum addresses of all memory banks with 0.
2. Apply read 0, write 1, read 1 to every minimum and maximum address of all memory banks, bank addresses ascending.
3. Repeat step 2 with the inversed data.
4. Repeat steps 2 and 3 with bank addresses descending.
5. Walk and write and read a single 1 (in a field of zeros) across the data lines. Repeat for all minimum and maximum addresses of all memory banks, bank addresses ascending.
6. Walk and write and read a single 0 (in a field of ones) across the data lines. Repeat for all minimum and maximum addresses of all memory banks, bank addresses ascending.

The LVDataInterconnect algorithm is described in [Table C-26](#).

Test Length

$$(26 + 8DW)Z$$

where DW is the datapath width of the memory and Z is the number of memory banks.

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 hold the low address.

AddressRegisterB

The AddressRegisterB segments are configured to hold the high address.

DataRegister

The logical data pattern D (<DataBusWidth>'b1) is loaded into both the write and expect data registers.

CounterA Register

The CounterA register is used to count the steps to walk the one or the zero across the data path and should be loaded with the value equal to <DataBusWidth> -1.

Algorithm Sequence

[Table C-26](#) describes the LVDataInterconnect algorithm sequence.

Table C-26. Description of LVDataInterconnect Algorithm

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
1	0	-	-	min address for every bank starting from lowest	W0	Write	Write zero.
1	1	-	-	max address for every bank starting from lowest	W0	Write	Write zero.

Table C-26. Description of LVDataInterconnect Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
2	2	-	-	min address for every bank starting from lowest	R0W1	ReadModifyWrite	Read zero from low address, then write one to the same address.
2	3	-	-	min address for every bank starting from lowest	R1	Read	Read one from low address.
2	4	-	-	max address for every bank starting from lowest	R0W1	ReadModifyWrite	Read zero from high address and write one.
3	5	Repeat from instruction 2	-	max address for every bank starting from lowest	R1	Read	Read one from the max address for every bank. Repeat from instruction 2 with the inverse of data .
4	6	-	-	max address for every bank starting from highest	R0W1	ReadModifyWrite	Read zero from high address, then write one to the same address.
4	7	-	-	max address for every bank starting from highest	R1	Read	Read one from high address.

Table C-26. Description of LVDataInterconnect Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
4	8	-	-	min address for every bank starting from highest	R0W1	ReadModifyWrite	Read zero from low address, then write one to the same address.
5	9	-	Repeat from instruction 6	min address for every bank starting from highest	R1	Read	Read one from low address. Repeat from instruction 6 with the inverse of data .
6	10	-	-	min address	W _D R _D	WriteRead	Write D-data, then read D-data the low address. Walk the one across the whole data register and repeat the write read operation.
6	11	-	-	max address	W _D R _D	WriteRead	Write D-data, then read D-data to the high address. Walk the one across the whole data bus and repeat the write read operations.
6	12	-	-	Increment the bank address	-	NoOperation	Increment the bank address and repeat from instruction 10.
7	13	-	-	min address	W _D R _D	WriteRead	Write \bar{D} -data register, then read \bar{D} -data register from the low address. Walk the zero across the whole data bus and repeat the write read operation.

Table C-26. Description of LVDataInterconnect Algorithm (cont.)

Phase	Instruction #	Repeat LoopA	Repeat LoopB	Address Sequence	Sequence	Operation	Description
7	14	-	-	max address	$W_D R_D$	WriteRead	Write \bar{D} -data register, then read \bar{D} -data register from the high address. Walk the zero across the whole data bus and repeat the write read operation.
7	15	-	-	Increment the bank address	-	NoOperation	Increment the bank address and repeat from instruction 13.

Example Algorithm Wrapper

Figure C-15 illustrates the *Algorithm* wrapper for the example memory.

Figure C-15. LVDataInterconnect Example Algorithm Wrapper

```
Algorithm (LVDATAINTERCONNECT) {
    TestRegisterSetup {
        AddressGenerator {
            load_bank_address_min : 2'b00;
            load_row_address_min : 3'b000;
            load_column_address_min : 2'b00;
            load_row_address_max : 3'b111;
            load_column_address_max : 2'b11;
            AddressRegisterA {
                load_bank_address : min_bank;
                load_row_address : min_row ;
                load_column_address : min_column;
                number_x0_bits : 0;
                number_y0_bits : 0;
                z_carry_in : NONE;
                x1_carry_in : NONE;
                y1_carry_in : NONE;
            }
            AddressRegisterB {
                load_bank_address : min_bank;
                load_row_address : max_row ;
                load_column_address : max_column;
                number_x0_bits : 0;
                number_y0_bits : 0;
                z_carry_in : NONE;
                x1_carry_in : NONE;
                y1_carry_in : NONE;
            }
        } // End of AddressGenerator wrapper
        DataGenerator {
            load_write_data : 8'b00000001;
            load_expect_data : 8'b00000001;
        }
        LoadCounterA_EndCount : 7;
        operation_set_select : SYNC;
    } // End of TestRegisterSetup wrapper
    MicroProgram {
        Instruction (PHASE_2_LADDR_WRITE) {
            AddressCommands {
                address_select : select_a;
            }
            DataCommands {
                expect_data : zero;
            }
            operation_select : WRITE;
            DataCommands {
                write_data : zero;
            }
            NextConditions {
            }
        }
        Instruction (PHASE_2_HADDR_WRITE) {
            AddressCommands {
                address_select : A_Xor_B;
            }
        }
    }
}
```

```

branch_to_instruction : PHASE_2_LADDR_WRITE;
DataCommands {
    expect_data : zero;
operation_select : WRITE;
DataCommands {
    write_data : zero;
}
AddressCommands {
    z_address : increment;
}
NextConditions {
    z_end_count : on;
}
}
Instruction (PHASE_3_4_LADDR_RMW) {
    AddressCommands {
        address_select : select_a;
    }
    DataCommands {
        expect_data : zero;
    }
    operation_select : READMODIFYWRITE;
    DataCommands {
        write_data : one;
    }
    NextConditions {
    }
}
Instruction (PHASE_3_4_LADDR_R) {
    AddressCommands {
        address_select : select_a;
    }
    DataCommands {
        expect_data : one;
    }
    operation_select : READ;
    DataCommands {
        write_data : zero;
    }
    NextConditions {
    }
}
Instruction (PHASE_3_4_HADDR_RMW) {
    AddressCommands {
        address_select : A_Xor_B;
    }
    DataCommands {
        expect_data : zero;
    }
    operation_select : READMODIFYWRITE;
    DataCommands {
        write_data : one;
    }
    NextConditions {
    }
}
Instruction (PHASE_3_4_HADDR_R) {
    AddressCommands {

```

```
        address_select : A_Xor_B;
    }
    branch_to_instruction : PHASE_3_4_LADDR_RMW;
    DataCommands {
        expect_data : one;
    }
    operation_select : READ;
    DataCommands {
        write_data : zero;
    }
    z_address : increment;
    NextConditions {
        z_end_count : on;
        RepeatLoopA {
            branch_to_instruction : PHASE_3_4_LADDR_RMW;
            Repeat {
                address_sequence : NoChange;
                write_data_sequence : inverse;
                expect_data_sequence : inverse;
                inhibit_last_address_count : on;
            }
        }
    }
}
Instruction (PHASE_5_6_HADDR_RMW) {
    AddressCommands {
        address_select : A_Xor_B;
    }
    DataCommands {
        expect_data : zero;
    }
    operation_select : READMODIFYWRITE;
    write_data : one;
}
NextConditions {
}
}
Instruction (PHASE_5_6_HADDR_R) {
    AddressCommands {
        address_select : A_Xor_B;
    }
    DataCommands {
        expect_data : one;
    }
    operation_select : READ;
    DataCommands {
        write_data : zero;
    }
    NextConditions {
    }
}
Instruction (PHASE_5_6_LADDR_RMW) {
    AddressCommands {
        address_select : select_a;
    }
    DataCommands {
        expect_data : zero;
    }
}
```

```

        operation_select : READMODIFYWRITE;
        DataCommands {
            write_data : one;
        }
        NextConditions {
        }
    }
    Instruction (PHASE_5_6_LADDR_R) {
        AddressCommands {
            address_select : select_a;
        }
        branch_to_instruction : PHASE_5_6_HADDR_RMW;
        DataCommands {
            expect_data : one;
        }
        operation_select : READ;
        DataCommands {
            write_data : zero;
        }
        AddressCommands {
            z_address : decrement;
        }
        NextConditions {
            z_end_count : on;
            RepeatLoopB {
                branch_to_instruction : PHASE_5_6_HADDR_RMW;
                Repeat {
                    address_sequence : NoChange;
                    write_data_sequence : inverse;
                    expect_data_sequence : inverse;
                    inhibit_last_address_count : on;
                }
            }
        }
    }
    Instruction (PHASE_7_LADDR_WR) {
        AddressCommands {

            address_select : select_a;
        }
        CounterCommands {
            counter_a : increment;
        }
        DataCommands {
            expect_data : DataReg_Rotate;
        }
        operation_select : WRITEREAD;
        DataCommands {
            write_data : DataReg_Rotate;
        }
        NextConditions {
            counter_a_end_count : on;
        }
    }
    Instruction (PHASE_7_HADDR_WR) {
        AddressCommands {
            address_select : A_Xor_B;
        }
    }

```

```
    CounterCommands {
        counter_a : increment;
    }
    DataCommands {
        expect_data : DataReg_Rotate;
    }
    operation_select : WRITEREAD;
    DataCommands {
        write_data : DataReg_Rotate;
    }
    NextConditions {
        counter_a_end_count : on;
    }
}
Instruction (PHASE_7_INC_BANK) {
    AddressCommands {
        address_select : select_a;
    }
    branch_to_instruction : PHASE_7_LADDR_WR;
    DataCommands {
        expect_data : zero;
        inhibit_data_compare : on;
    }
    operation_select : NOOPERATION;
    DataCommands {
        write_data : zero;
    }
    AddressCommands {
        z_address : increment;
    }
    NextConditions {
        z_end_count : on;
    }
}
Instruction (PHASE_8_LADDR_WR) {
    AddressCommands {
        address_select : select_a;
    }
    CounterCommands {
        counter_a : increment;
    }
    DataCommands {
        expect_data : InverseDataReg_Rotate;
    }
    operation_select : WRITEREAD;
    DataCommands {
        write_data : InverseDataReg_Rotate;
    }
    NextConditions {
        counter_a_end_count : on;
    }
}
Instruction (PHASE_8_HADDR_WR) {
    AddressCommands {
        address_select : A_Xor_B;
    }
    CounterCommands {
```

```

        counter_a : increment;
    }
    DataCommands {
        expect_data : InverseDataReg_Rotate;
    }
    operation_select : WRITEREAD;
    DataCommands {
        write_data : InverseDataReg_Rotate;
    }
    NextConditions {
        counter_a_end_count : on;
    }
}
Instruction (PHASE_8_INC_BANK) {
    AddressCommands {
        address_select : select_a;
    }
    branch_to_instruction : PHASE_8_LADDR_WR;
    DataCommands {
        expect_data : zero;
    }
    operation_select : NOOPERATION;
    DataCommands {
        write_data : zero;
    }
    AddressCommands {
        z_address : increment;
    }
    NextConditions {
        z_end_count : on;
    }
}
} // End of MicroProgram wrapper
} // End of Algorithm wrapper

```

Fault Coverage

The faults detected by the LVDataInterconnect algorithm are identified in [Table C-27](#).

Table C-27. LVDataInterconnect Algorithm Fault Coverage

Fault Type	Fault Coverage
Stuck-At Fault on a single data line at either a logic 1 or logic 0.	Yes
Stuck-Open Faults on data line. A stuck-open fault is a single data line that contains an open (break).	Yes
Bridging Faults. A bridging fault is two or more data lines shorted together. The short can result in either an ANDing or ORing of the values on the affected lines.	Yes

Specification

To test SRAMs using the LVDataInterconnect algorithm, specify **Algorithm** LVDataInterconnect 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 LVDataInterconnect algorithm:

- A binary complement values for the min and max should be assigned to CountRange(BankAddress), CountRange(RowAddress), and/or CountRange(ColumnAddress) in the memory TCD file if coverage of the Wired-OR fault model is desired for address-line shorts.
- All the physical address lines cannot be mapped to bank address only. The memory configuration should have at least two logical addresses.

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	630
Bit/Group/Global Write Enable Faults	630
Access Transistor Current Leakage Faults	630
Data Retention Faults	631
DataPath Shorts	631
Destructive Read Faults	632
Dynamic Coupling Faults	633
Idempotent Coupling Faults	634
Inversion Coupling Faults	635
Memory Select Faults	635
Multi-Port Interference Faults	636
Multiport Synchronous Bitline Coupling Faults	637
Neighborhood Pattern Sensitive Faults	637
Parametric Faults	637
Read Disturb Faults	638
Read Enable Faults	638
Single Port Bitline Coupling Faults	639
Stuck-At Faults	639
Stuck-Open Faults	640
Transition Faults	640
Write Recovery Faults	641

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 will 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 which 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, pause 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 in order to facilitate the application on the tester. It is recommended to 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](#).

DataPath Shorts

Datapath shorts occur between even-numbered and odd-numbered elements of the data path, which 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.

Algorithm

The DataPath 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. Mentor 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 in order to avoid slowing down the diagnosis process due to multiple identical failures being reported. In fact, the last strobe can be omitted in all SMArch* algorithms as the fault will be 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 since the N-channel needs to transfer more charge from the bit lines.

Since 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 Mentor 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 Mentor 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 and/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*).

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 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](#).

Appendix E

Parallel Static Retention Testing

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	643
Handling Multiple Controllers	645
Sequence for Test Sub-Phases	645
Sample PatternsSpecification Syntax for Test Sub-Phases	646
Handling Memories Tested Sequentially.....	647
Testing Controllers in Groups	648
Sequence for Test Sub-Phases	648
Sample PatternsSpecification Syntax for Test Sub-Phases	649

Parallel Static Retention Testing Limitations

Running PSRT has some limitations.

- These library algorithms support PSRT:
 - SMarch
 - SMarchCHKB
 - SMarchCHKBci
 - SMarchCHKBcil
 - SMarchCHKBvcd

- PSRT can be executed only in HWDefault mode.

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	645
Sample PatternsSpecification Syntax for Test Sub-Phases	646

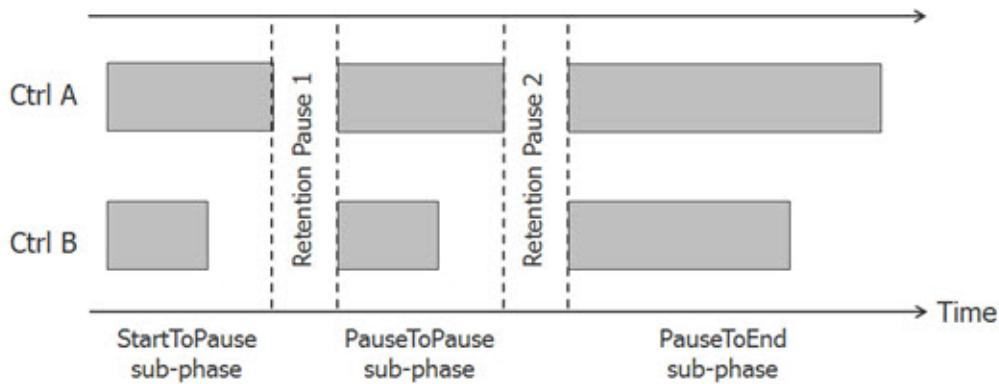
Sequence for Test Sub-Phases

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 execute 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 execute 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 execute 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 test bench 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) {
  : 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 (0) {
: parallel_retention_time: 10ms;
: AdvancedOptions (A) {
:   retention_test_phase: start_to_pause;
: }
: AdvancedOptions (B) {
:   retention_test_phase: start_to_pause;
: }
: }
: TestStep (1) {
: parallel_retention_time: 10ms;
: Controller (A) {
:   retention_test_phase: pause_to_pause;
: }
: AdvancedOptions (B) {
:   retention_test_phase: pause_to_pause;
: }
: }
: TestStep (2) {
: parallel_retention_time: 10ms;
: AdvancedOptions (A) {
:   retention_test_phase: pause_to_end;
: }
: Controller (B) {
:   retention_test_phase: pause_to_end;
: }
: }
: }
```

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 *Cntrl 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 executed 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	648
Sample PatternsSpecification Syntax for Test Sub-Phases	649

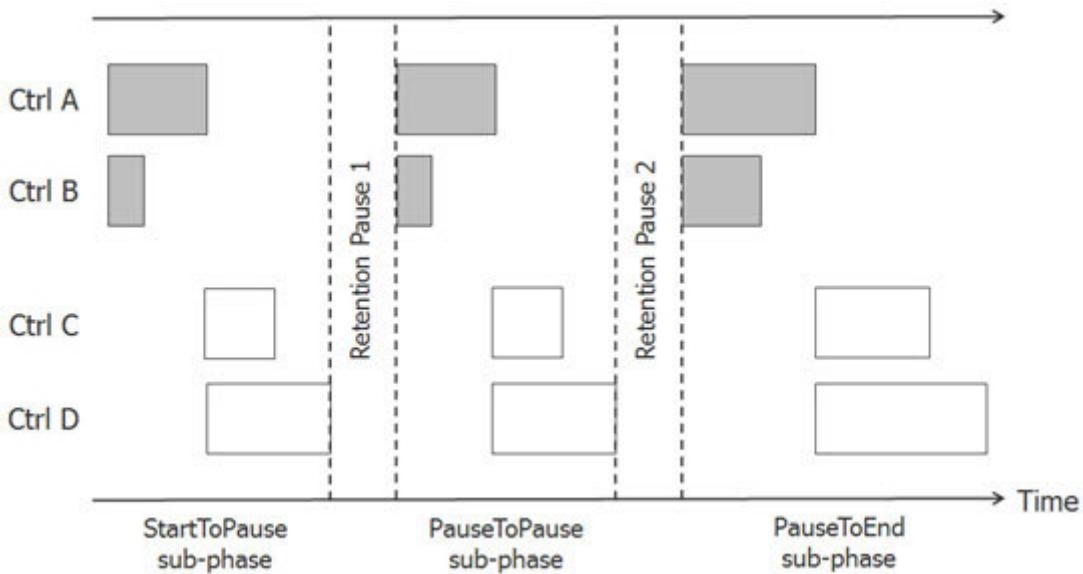
Sequence for Test Sub-Phases

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 *Cntrl A* and *Cntrl B* in parallel to execute the *start_to_pause* sub-phase.
2. Enable *Cntrl C* and *Cntrl D* in parallel to execute the *start_to_pause* sub-phase.
3. When *Cntrl C* and *Cntrl D* complete Step 2, apply the first static retention test pause.
4. Enable *Cntrl A* and *Cntrl B* in parallel to execute the *pause_to_pause* sub-phase.
5. Enable *Cntrl C* and *Cntrl D* in parallel to execute the *pause_to_pause* sub-phase.
6. When *Cntrl C* and *Cntrl D* complete Step 5, apply the second static retention test pause.
7. Enable *Cntrl A* and *Cntrl B* in parallel to execute the *pause_to_end* sub-phase.
8. Enable *Cntrl C* and *Cntrl D* in parallel to execute 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 test bench 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) {
    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 test bench 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	652
Example 1	652
Logical and Physical Mapping	655
Example 2	657
Example 3	658
Example 4	660
Example 5	662

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.

$$w \times b = nrow \times ncol$$

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 0 th 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:

$$2nadd \geq w > 2(nadd-1)$$

The address inputs are divided into nra row address inputs and nca column address inputs where:

$$2nra \geq nrow > 2(nra-1),$$

$$2nca \geq ncolmux > 2(nca-1).$$

Example 1 652

Example 1

Consider a $1k \times 32$, 1-port memory, with 4:1 column multiplexing. This memory has the following:

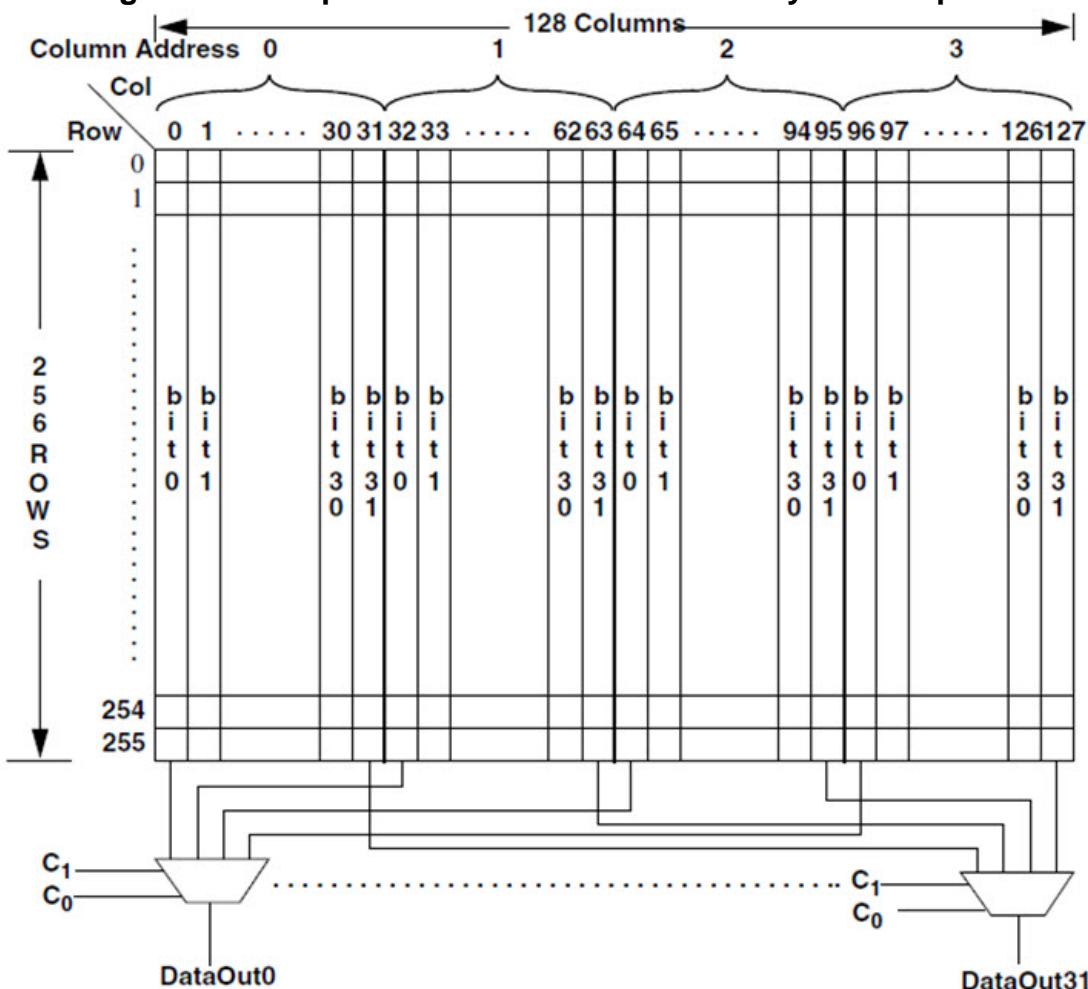
$$w = 1024, \ b = 32, \ ncolmux = 4$$

Therefore,

$$\begin{aligned} nrow &= w / ncolmux = 1024 / 4 = 256, \\ ncol &= b \times ncolmux = 32 \times 4 = 128, \\ nadd &= \log_2 w = \log_2 1024 = 10, \\ nra &= \log_2 nrow = \log_2 256 = 8, \text{ and} \\ nca &= \log_2 ncolmux = \log_2 4 = 2. \end{aligned}$$

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 C0 and C1, 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,

Example 1

- nadd = number of addresses $\leq \log_2 w$,
- nra = number of row addresses $\leq \log_2 n_{row}$, and
- nca = number of column addresses $\leq \log_2 n_{colmux}$.

Note

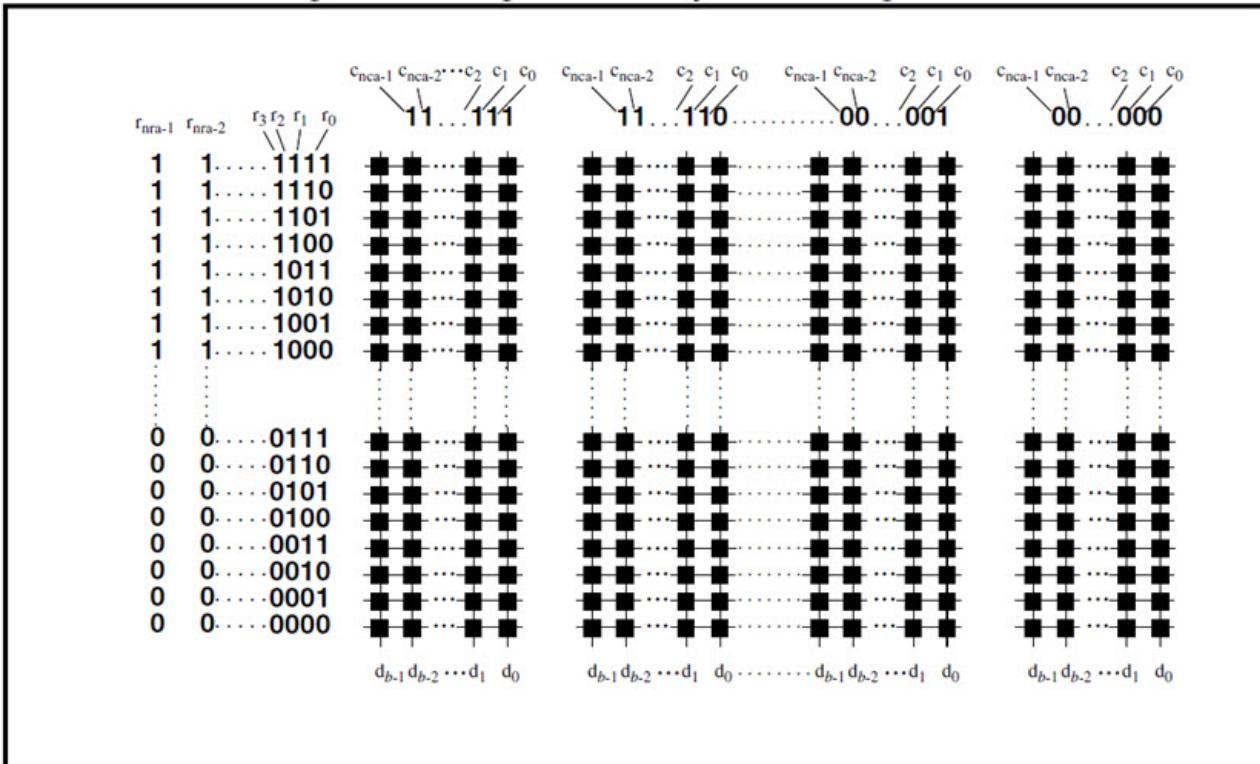
 For $\log_2 w$, $\log_2 n_{row}$, and $\log_2 n_{colmux}$, when the value is not a whole number, round the value *up* to the next whole number.

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:

- In order 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.
- In order to balance the load on different address lines or (pre)decoded lines, memory designers sometimes scatter the wordlines or bitlines.
- In order 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.
- In order to increase the yield for larger memories, spare (redundant) rows and/or columns are often implemented, which typically disrupt the physical address sequence.

To be as general and straight-forward as possible, the Mentor Graphics memory BIST controller assumes the logical arrangement of cells when generating patterns, particularly the checkerboard patterns. In order 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 due to a *scrambling* of the rows and columns and/or 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 n th logical row and m th 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 n th physical row and m th 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 **657**

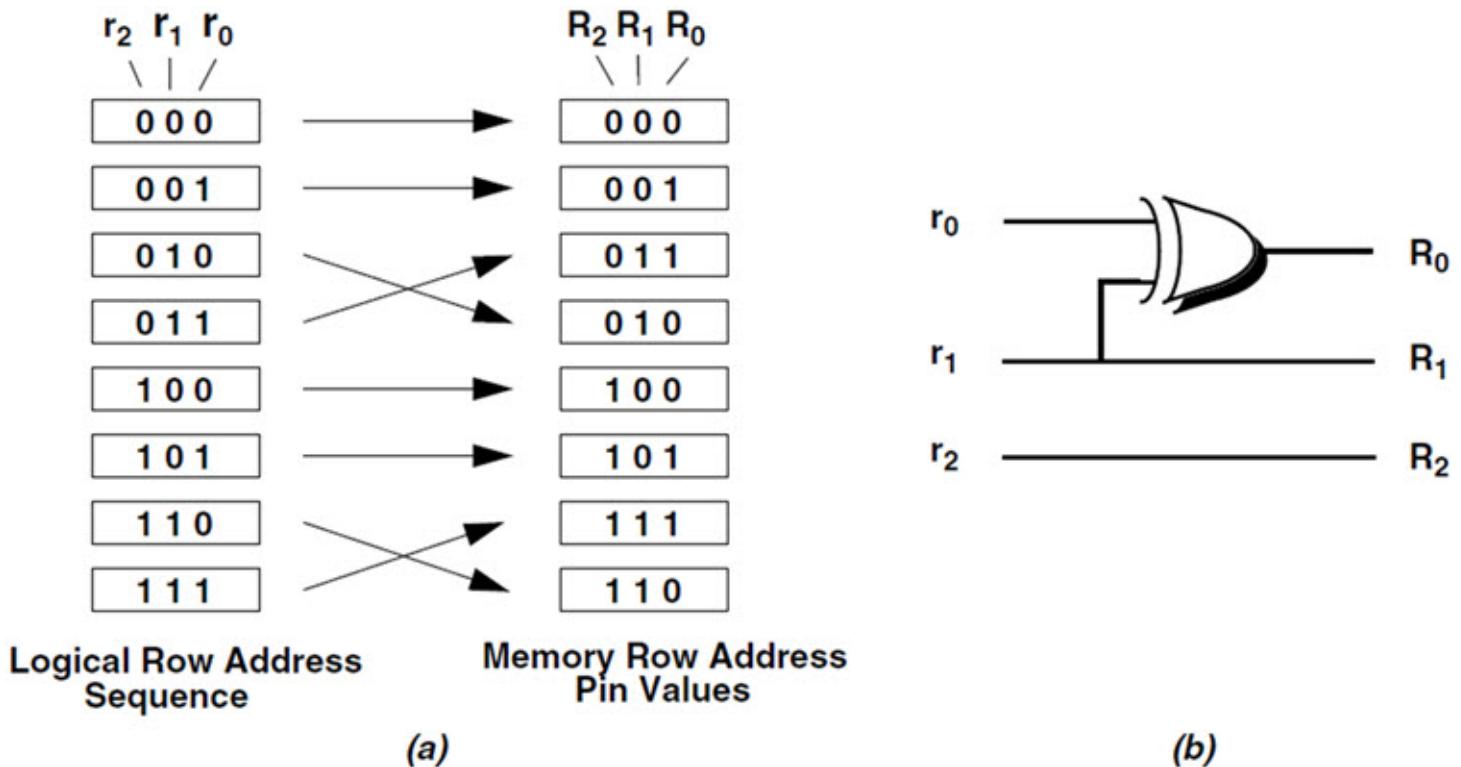
Example 3 **658**

Example 4	660
Example 5	662

Example 2

Consider the row address mapping example shown in Figure E-3.

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, in order to achieve the desired sequence, 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
```

The corresponding memory library file syntax is as follows:

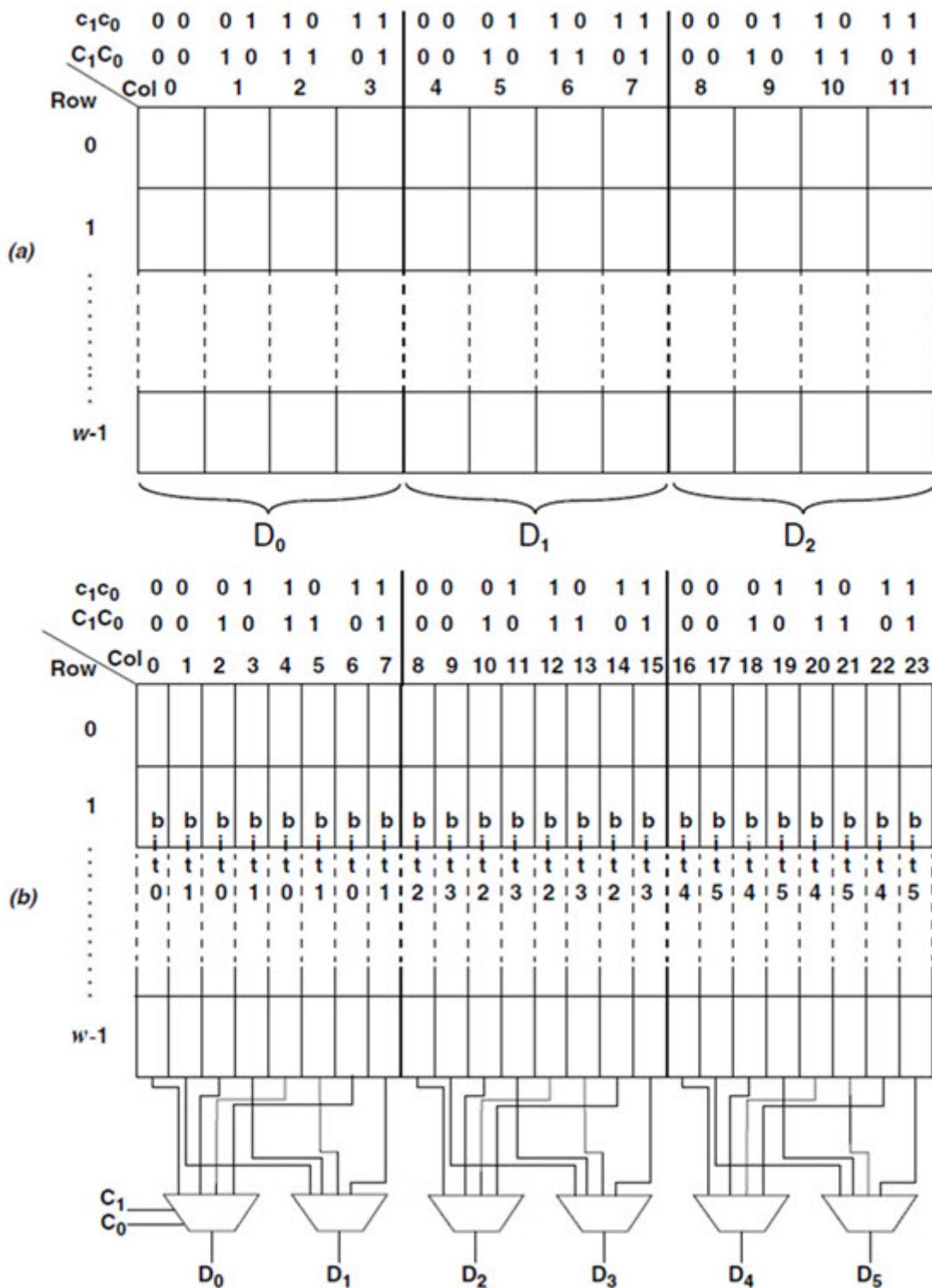
```
PhysicalAddressMap {  
    RowAddress[0] : r[0] xor r[1];  
    RowAddress[1] : r[1];  
    RowAddress[2] : r[2];  
}
```

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 c1c0 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 desired pattern:

$$\begin{aligned} C_0 &= c_1 \\ C_1 &= c_0 \text{ xor } c_1 \end{aligned}$$

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;
```

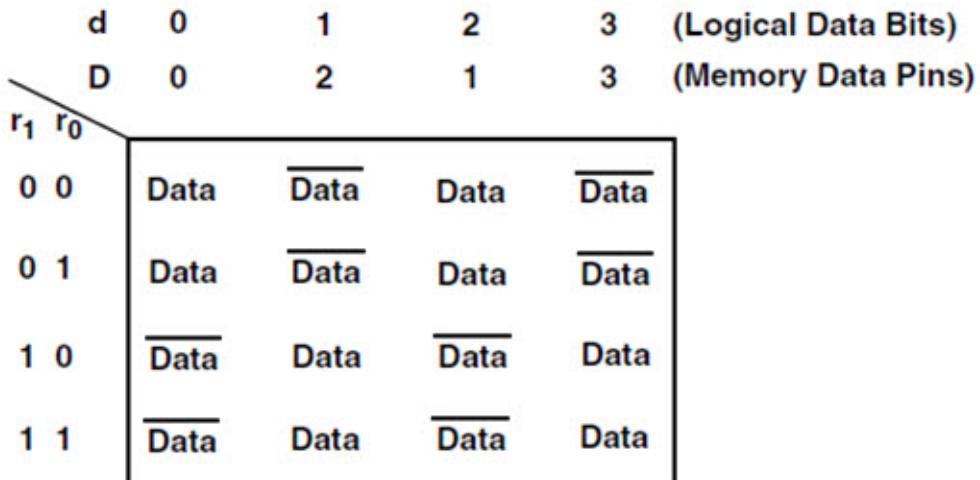
In [Figure F-4](#), the multiplexers before the data output pins are drawn to better illustrate the column multiplexing.

Example 4

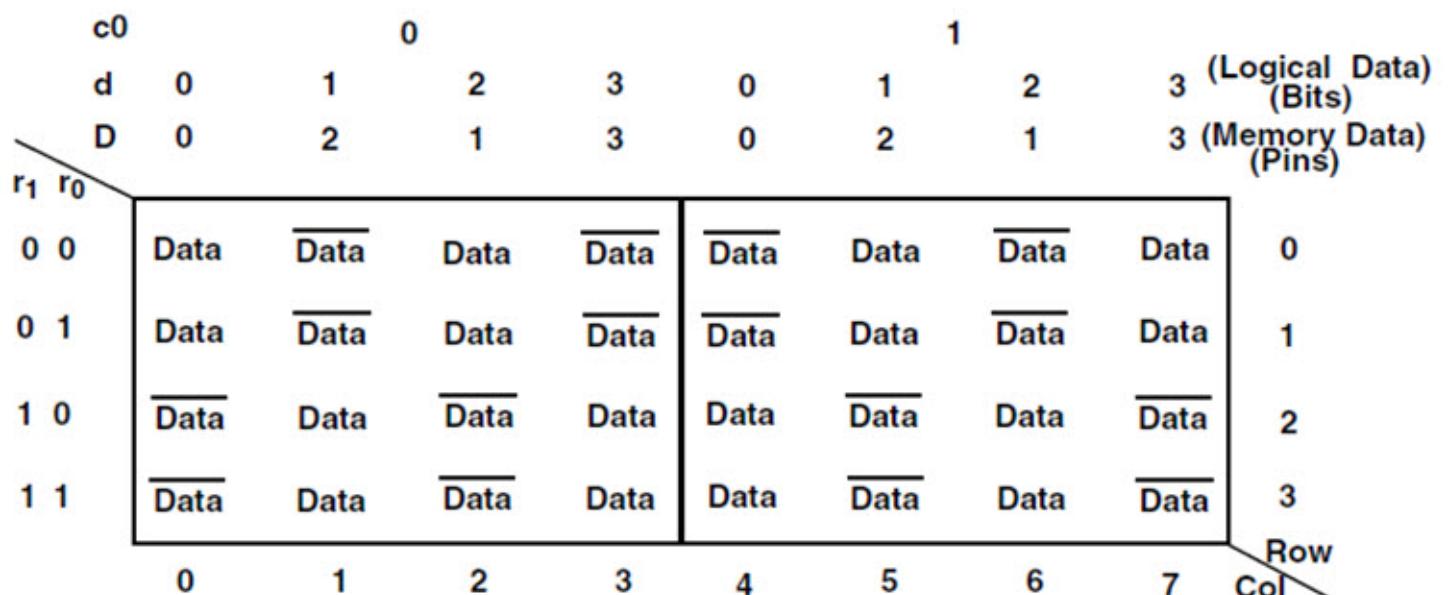
Consider a memory that has w words, each word 4 bits wide.

[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

Example 5

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 (D0), note that the logical data bit 0 (d0) can be mapped to D0 with one difference—when r1 is 1, the data bit value is flipped. Furthermore, D2, which is physically placed as the second column of the memory array, needs logical d1, but here data is flipped when r1 is 0. That is why the NOT of r1 XORed with d1 is used. A similar explanation is in place for D1 and D3.

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-5](#), while columns 4-7 are the mirror image of columns 0-3 with respect to the vertical line. When the column address c0 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 [Figure E-6](#).

This example illustrates portions of four columns of a memory array with a *BitGrouping* of 1.

As you can observe, going from one column to the other, data inverts when c0 is 1. Furthermore, the data pattern on each row is repeated while r6 is 0. From row 64 onwards, however, the data pattern is inverted for the columns with c0 equal to 0 due to bitline twist. Therefore, you can write the data mapping for D0 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

		column	0	1	2	3
		$c_1 c_0$	00	01	10	11
row		$r_6 r_5 \dots r_1 r_0$				
0	0	0 0 0 0	Data	<u>Data</u>	Data	<u>Data</u>
1	0	0 0 0 1	Data	<u>Data</u>	Data	<u>Data</u>
.
.
.
63	0	1 1 1	Data	<u>Data</u>	Data	<u>Data</u>
64	1	0 0 0	<u>Data</u>	<u>Data</u>	<u>Data</u>	<u>Data</u>
65	1	0 0 1	<u>Data</u>	<u>Data</u>	<u>Data</u>	<u>Data</u>
.
.
.
127	1	1 1 1	<u>Data</u>	<u>Data</u>	<u>Data</u>	<u>Data</u>

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.

In order 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](#)

Appendix G

Complete Examples for Multi-Segment Memory Repair

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.

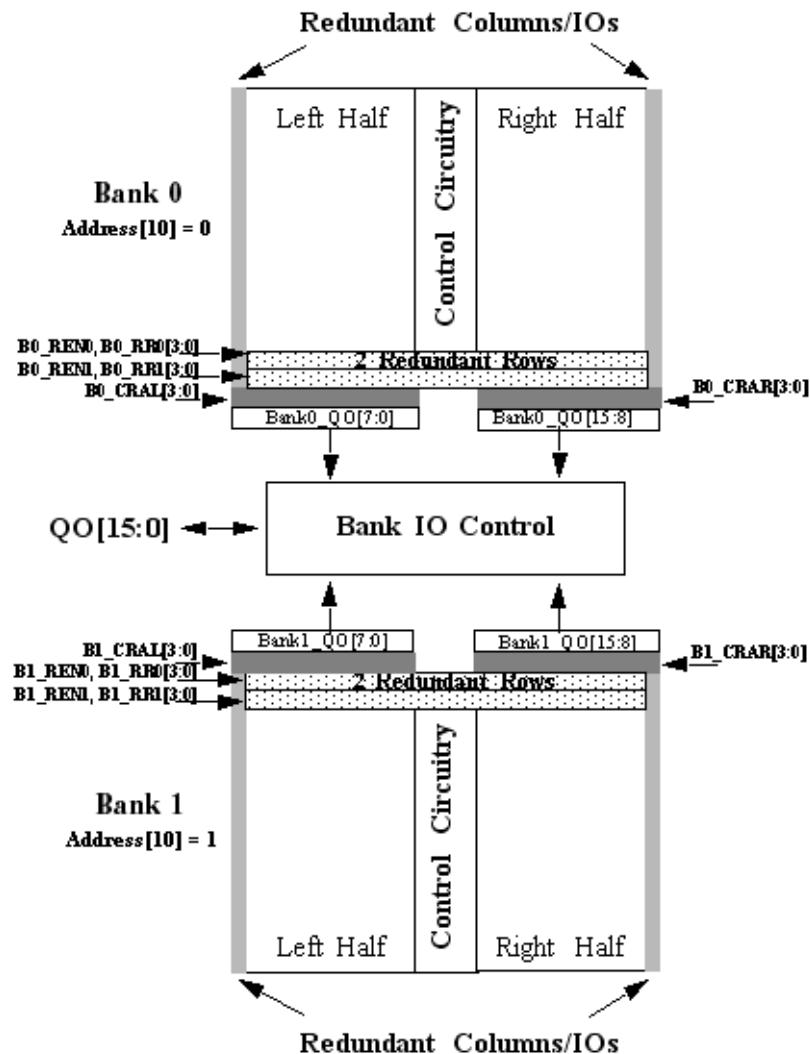
This appendix covers the following topics:

Parallel BISR Interface Example	667
Serial BISR Interface Example	673

Parallel BISR Interface Example

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;
}
```

Complete Examples for Multi-Segment Memory Repair

Parallel BISR Interface Example

```
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_RENO;
                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_RENO;
                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;
        }
    }
}

```

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:
 - Two ColumnSegment wrappers for *Bank0*
 - Two ColumnSegment wrappers for *Bank1*

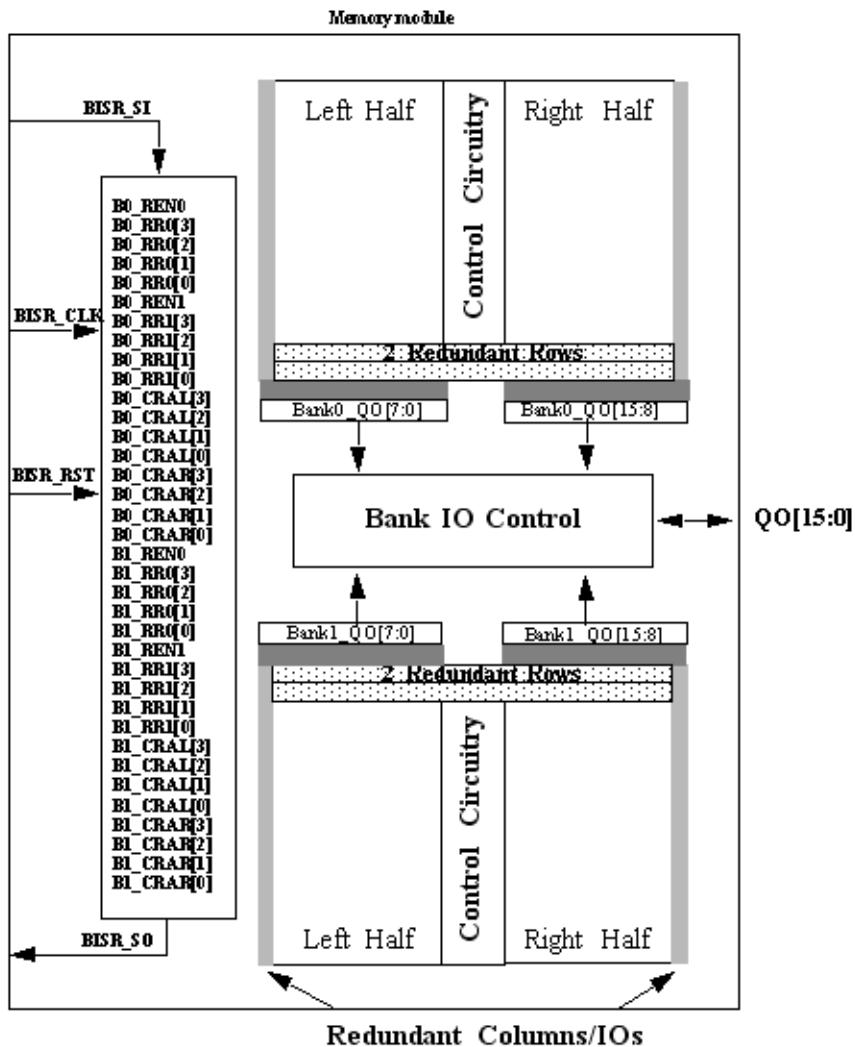
Each `ColumnSegment` has a `RowSegmentCountRange` property that assigns it to the appropriate range.

- 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;
}
```

Complete Examples for Multi-Segment Memory Repair

Serial BISR Interface Example

```
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 many memories with destructive scan-out.
- Read a single location of a single memory with a nondestructive scan-out that allows many algorithm executions with single scan-in.
- 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.

This appendix covers the following topics:

Feature Description	680
Algorithm Description	680
Verification of Read and Write Access Functions	684
Requirements, Assumptions and Limitations	685
Implementation	686

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 `functional_debug_mode` property in the `DftSpecification/MemoryBist/Controller/AdvancedOptions` wrapper.

Note

 When `functional_debug_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	680
Verification of Read and Write Access Functions	684
Requirements, Assumptions and Limitations	685

Algorithm Description

Custom algorithms that access a single memory location and from a range of memory locations will be described in this section.

The algorithm may be included during the [Design Loading](#) step so that it will be 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](#). Address A 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

```

Algorithm (MemReadAlgo) {
    TestRegisterSetup {
        OperationSetSelect: SYNC;
        AddressGenerator {
            AddressRegisterA {
                load_row_address: 4'h0;//actual row address loaded at run time
                load_column_address: 4'h0;// actual column address also
                load_bank_address: 4'h0;// 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: 2'b00; // 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 execute 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 execute the algorithm and is a preparation step for the subsequent Exec TestSteps. Therefore, the Init TestStep runtime is very short.

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 {
                    functional_debug_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 executes 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 allows 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 {
                    functional_debug_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

```
Algorithm (MemWriteAlgo) {
    TestRegisterSetup {
        OperationSetSelect: SYNC;
        AddressGenerator {
            AddressRegisterA { // Address A is used in this example
                load_row_address: 4'h0; // Default initial row address
                load_column_address: 4'h0; // Default initial column address
                x1_carry_in: y1_carry_out;
                y1_carry_in: none;
            }
        }
        DataGenerator {
            load_write_data: 2'b00; // 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. To

limit simulation time, only 16 locations are accessed. All locations are initialized to 0s (h00). Next, two arbitrary locations are changed to 1s (hFF). Lastly, the final content of all 16 locations is read back. The following example uses an 8-bit address register.

Figure H-3. Test Steps Used for Verification of Debug Capability

```

Patterns(verify_func_debug_mode) {
    // Initialize the first 16 locations to 8'h00
    TestStep(mem_write_zeros_init) {
    }
    TestStep(mem_write_zeros_exec) {
    }

    // Write hFF to first arbitrary location 8'h01
    TestStep(mem_write_init_1) {
    }
    TestStep(mem_write_exec_1) {
    }
    // Write hFF to second arbitrary location 8'h04
    TestStep(mem_write_init_2) {
    }
    TestStep(mem_write_exec_2) {
    }

    // Read first 16 locations and compare content to h00
    // except locations 8'h01 and 8'h04
    TestStep(mem_read_init) {
    }
    TestStep(mem_read_exec_1) {
    }
    TestStep(mem_read_exec_2) {
    }
    ...
    TestStep(mem_read_exec_16) {
    }
}

```

Requirements, Assumptions and Limitations

The following are the requirements, assumptions and limitations in utilizing functional debug memory access in Tesson 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 which 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 will delay 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 will be 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. 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. So, if an algorithm requires execution of X number of operations, and CounterA is used to keep track of 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 SVF, the `GO_ID` registers will remain 0. The memory is disabled when the address from the controller goes out of range.

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 functional_debug_mode property must be specified for the MemoryBIST controller when configuring the DFT Specification:

```
DftSpecification {
    MemoryBist {
        Controller(id) {
            AdvancedOptions {
                .
                .
                .
                functional_debug_mode: on ;
                .
                .
            }
        }
    }
}
```

When functional_debug_mode is set to on, Tessent MemoryBIST will generate the necessary setup chain muxing and controller ports to allow 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/
functional_debug_mode on
```

PatternsSpecification

Functional Debug Memory Access mode requires preparing and applying an initialization (Init) TestStep and a series of execute (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 executes 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 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 will be 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 will be inverted when the column and/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;  
                Controller(c1_inst) {  
                    MemoryInterface(m1) {  
                        comparator_id_select : all;  
                    }  
                    AdvancedOptions {  
                        functional_debug_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;  
            Controller(c1_inst) {  
                AdvancedOptions {  
                    functional_debug_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 will be 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 3 subsequent Exec TestSteps executes 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;
            Controller(c1_inst) {
                MemoryInterface(m1) {
                    comparator_id_select : all;
                }
                AdvancedOptions {
                    functional_debug_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 {
                    }
                }
            }
        }
    }
}}
```

Functional Debug Memory Access Implementation

```
TestStep(Functional_debug3_exec1) {
    MemoryBist {
        run_mode : hw_default;
        Controller(c1_inst) {
            AdvancedOptions {
                functional_debug_mode : on;
                test_execution_cycles : 500;
            }
            DiagnosisOptions {
                compare_go_id : on;
            }
        }
    }
}
TestStep(Functional_debug3_exec2) {
    MemoryBist {
        run_mode : hw_default;
        Controller(c1_inst) {
            AdvancedOptions {
                functional_debug_mode : on;
                test_execution_cycles : 500;
            }
            DiagnosisOptions {
                compare_go_id : on;
            }
        }
    }
}
TestStep(Functional_debug3_exec3) {
    MemoryBist {
        run_mode : hw_default;
        Controller(c1_inst) {
            AdvancedOptions {
                functional_debug_mode : on;
                test_execution_cycles : 500;
            }
            DiagnosisOptions {
                compare_go_id : on;
            }
        }
    }
}
} // }}
```

Appendix I

BAP Parallel Memory Access

The BAP parallel memory access feature 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, execute Go/NoGo tests and monitor the pass/fail status.

The primary application of BAP parallel memory access is in-system testing of memories. For in-system test, memory BIST may be applied during the power-on/off phase of the device operation, or periodically while the device is operating. The portion of the BAP parallel memory access feature that interacts with the MemoryBIST controllers is configurable through the IJTAG protocol, and this controller configuration may be used for in-system test through the Tesson MissionMode controller as well as for manufacturing test within the ATE environment. The advanced BAP allows certain feature overrides in the hw_default operating mode of the memory BIST controller. Test time can be reduced by eliminating shift cycles to serially configure the controllers in the IJTAG environment.

This appendix describes the BAP parallel 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.

Feature Description	692
BAP Architecture	692
BAP Direct Access Interface Pins	694
System Logic Interaction and Timing	698
Requirements, Assumptions and Limitations	699
Implementation	701
Inserting the BAP Direct Access Interface	702
Verifying the BAP Direct Access Interface	707
Timing Closure Considerations	712

Feature Description

The BAP module is created for each MBIST DftSpecification insertion pass and services one or more MemoryBIST controllers. 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.

The sections listed below provide the system-level details of the BAP parallel memory access feature.

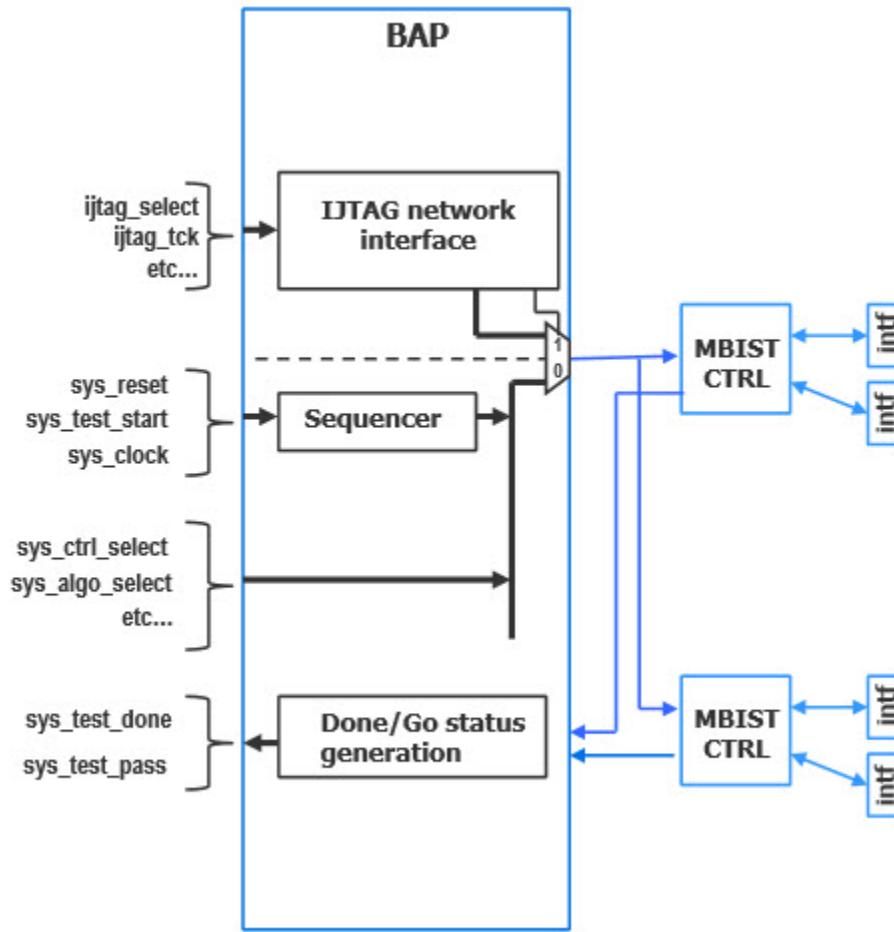
BAP Architecture.....	692
BAP Direct Access Interface Pins.....	694
System Logic Interaction and Timing	698

BAP Architecture

Enabling the BAP parallel memory access feature creates a direct access interface on the BAP. The direct access interface supports a low latency protocol to configure the MemoryBIST controller, execute Go/NoGo test and monitor the pass/fail status.

A block diagram for the BAP parallel memory access feature, together with the IJTAG network interface, is shown in [Figure I-1](#).

Figure I-1. BAP Parallel Memory Access Diagram



The parallel 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 allows 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.

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.

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 will typically be different, so the pass/fail and completion status of individual controllers can also be monitored.

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 will contain the ports necessary to support the controller features enabled within the Controller/DirectAccessOptions/`ExecutionSelections` wrapper. The ports related to controller selection will not be 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.

The direct access interface ports that are created on the BAP are described in [Table I-1](#).

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 BAP for the parallel memory access mode.
sys_clock[_<clock_domain>]	Input	System clock.
sys_test_init[_<clock_domain>]	Input	An active low signal that asynchronously resets the memory BIST controllers. Set it to 0 at the beginning of the test and keep it at 0 at the end of the test if you want to reset the controller. Set it to 1 at the end of the test if you don't want to reset the controller for the subsequent run. Tie it to 0 if you want to reset the controller for each run.
sys_test_start[_<clock_domain>]	Input	An active high signal that initiates the memory test.
sys_test_pass[_<clock_domain>]	Output	Memory test pass/fail status. Asserted to 0 when at least one controller detects a failure.

Table I-1. BAP Direct Access Interface Ports (cont.)

Port Name	Type	Description
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.
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 will be used. Otherwise, the default algorithm encoded at RTL generation will be executed.
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 will be used. Otherwise, the default operation set encoded at RTL generation will be executed.
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 will be used. Otherwise, all the steps will be executed.
sys_memory_select[msb:lsb]	Input	Memory selection code.
sys_memory_select_en	Input	Individual enable signal per memory within a controller step. Select the memory to run by setting the corresponding bit to 1.
Controller Auto-Connected Signals		

Table I-1. BAP Direct Access Interface Ports (cont.)

Port Name	Type	Description
BIST_ASYNC_RESET	Output	Asynchronous reset signal. Connects to the ASYNC_RESETN input of the MBIST controller.
bistEn[msb:lsb]	Output	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 SETUP input of the MBIST controller.
MBISTPG_GO[msb:lsb]	Input	Pass/fail status per controller. Connects to the DONE 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	Enable the algorithm selection. 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.
BIST_SELECT_COMMON_OPSET	Output	Enable the operation set selection. 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	Enable the controller step selection. Connects to the STEP_SELECT_EN input of the MBIST controller.
BIST_MEM_SEL[x:y]	Output	Individual enable signal per memory within a controller step. Connects to the MEM_SELECT input of the MBIST controller.
BIST_MEM_SELECT_EN	Output	Enable the memory selection. Connects to the MEM_SELECT_EN input of the MBIST controller.

Table I-1. BAP Direct Access Interface Ports (cont.)

Port Name	Type	Description
sel	Input	Reserved for future use. Tied to 0.

System Logic Interaction and Timing

The following sections provide details for system logic connections and the direct access interface timing:

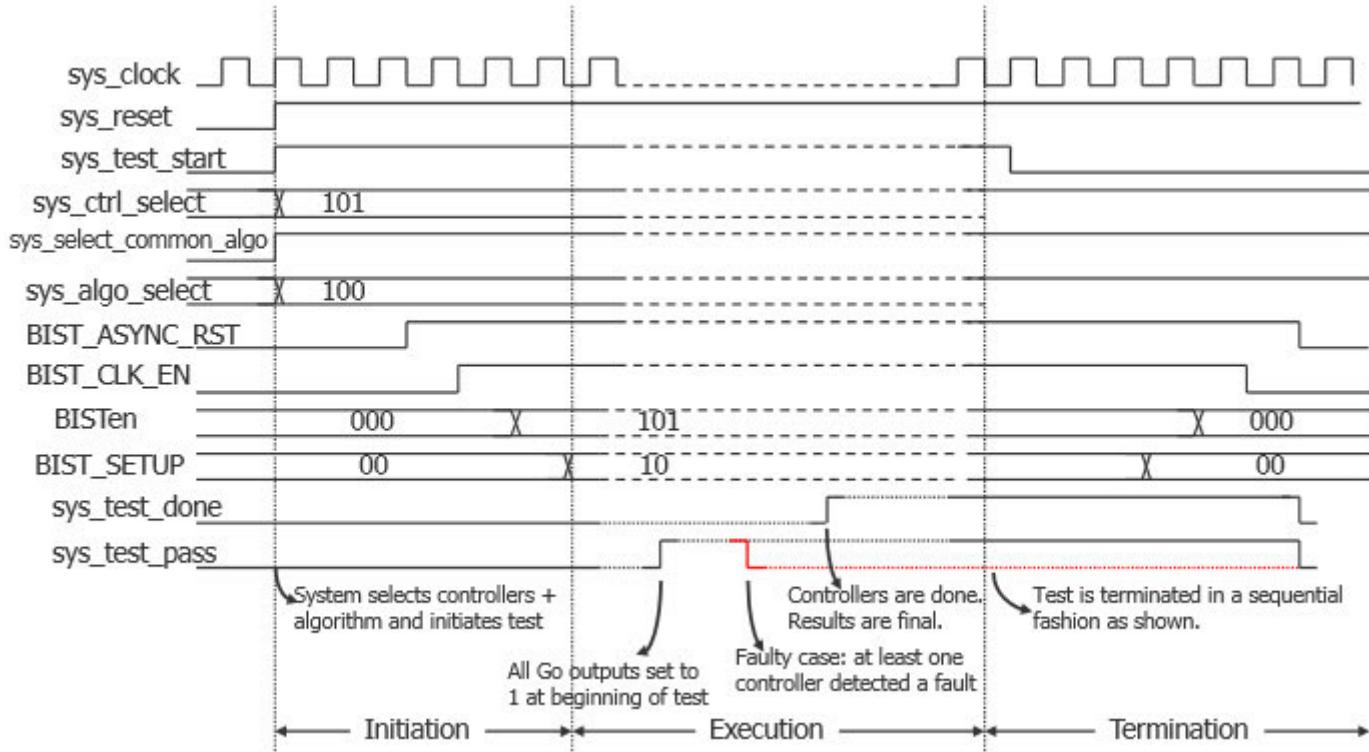
Timing Diagram	698
Clocking Schemes	698
Sampling Pass/Done Signals	699

Timing Diagram

The general access protocol for in-system testing using BAP parallel memory access is shown in the figure below. The timing diagram illustrates memory tests being initiated on memory controllers 0 and 2, and each will execute 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-2. BAP Parallel Memory Access Timing Protocol



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 will be implemented in the BAP to operate all controllers. The common clock must be slower than the lowest memory BIST clock frequency. This requirement allows time for the signals to propagate and stabilize, which avoids timing violations due to the cross domain paths from the BAP to the controller.

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 will be implemented in the BAP. Each sequencer will operate the controllers belonging to the same clock domain. In addition to the sequencer clock input, other signals such as reset, memory test pass/fail, and completion status, will be dedicated per clock domain as shown in [Table I-1](#).

Sampling Pass/Done Signals

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 combinatorial signals are sourced from memory BIST controller DONE and GO outputs, and can be from various memory BIST clock domains. Because of this, consideration on how the functional control logic captures these signals is necessary to avoid potential glitches in sampling.

There are two methods available for avoiding potential glitches when functional logic captures pass/done status from the BAP direct access interface:

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.
2. Synchronize the BAP direct access interface pass/done output signals with either a single or double flop, clocked by the appropriate functional clock. For single flop synchronizers, it is recommended to sample the result more than once to ensure stability of the output.

Utilizing a time-out function is good practice when sampling BAP direct access interface pass/done signals in case a controller does not reach the Done state.

Requirements, Assumptions and Limitations

The following are the requirements, assumptions and limitations for implementing and utilizing BAP parallel 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, will be 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 will be reported. Operating modes that require shifting the controller setup chain are not supported.

- Memories accessed through the shared bus interface will not be 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 which 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.

Implementation

The following sections provide details on implementing the BAP parallel 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	702
Verifying the BAP Direct Access Interface	707
Timing Closure Considerations	712

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	702
Clock Connections	704
System Connections	705

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 will be 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 will be performed using the default settings defined in the memory library and DFT specification. You will be able to test all memories at once, or a subset of memories based on controller assignment. A single sequencer will be 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.

```
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 will be equipped to select a group of memories tested concurrently or a single memory within controller c1. Additional IJTAG TDRs will be added into the BAP for control during manufacturing test. Additional input ports will be added on the BAP for control during in-system test.

```

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. Since IJTAG TDRs will be 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 desired. 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 will not be implemented, but will be equipped with IJTAG TDR to select the controller step and memory within controller c1.

```

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 will be able to perform in-system memory test on all memories at once, without the ability to select controller, algorithm or operation set. The BAP will be built with IJTAG TDR to program the algorithm or operation set.

```
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 which 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 since 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 since 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 system connections are configured within the [Connections](#) wrapper in the MemoryBist section of the DFT specification. In order to use the Connections wrapper, the BistAccessPort [DirectAccessOptions/direct_access](#) property must be set to on.

By default, the BAP direct access interface will contain 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 will not be 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 will 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 will be connected to the control logic and the remaining control logic bits will be unused. An error will be issued if the system control logic port or pin is smaller in width than its counterpart on the BAP direct access interface

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	707
Selection Codes.....	710

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](#) which 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 to controllers c1 and c2 will be tested using the default settings. The MBIST controllers will be clocked at the functional period of 10ns. Assume the system control logic operates on a slow functional clock. One sequencer will be implemented in the BAP. The BAP direct access interface will be clocked at 40ns, which is four times slower than the lowest MBIST clock frequency.

```
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.

In the following example, the BAP direct access interface will only enable controller c1. Furthermore, a custom algorithm will be applied to one specific memory. The memory under test is activated in the second controller step of controller c1.

```

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 will be 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 will be 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 BAP parallel 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 ijttag, 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 will execute controllers c1 and c2. Furthermore, a custom algorithm will be applied to all memories associated to both controllers. Since run_mode is hw_default, the algorithm will be configured in the BAP only. The configuration chains of controllers c1 and c2 will not be accessed.

```
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;
                }
            }
        }
    }
}
```

Selection Codes

System logic will source 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 these selections.

The controller selection codes are one-hot encoded in the order listed. For this example, controller c1 will have the selection code 2'b01 applied to the direct access interface sys_ctrl_select input ports, and controller c2 will be selected with 2'b10. Both controllers will be selected with the code 2'b11.

Similarly, memory selection codes are also one-hot encoded. In this example, assume controller c1 (2'b01) 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 will select the memory identified with an index of “0”, and a code of 2'b10 will select the memory identified with an index of “1”. Both memories in the step will be selected with the code 2'b11.

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) {
        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) {
        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
        }
    }
}
```

If there is only one controller configured for BAP parallel memory access, the BAP direct access interface will not contain sys_ctrl_select input ports for controller selection.

Timing Closure Considerations

Insertion of the BAP parallel 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 which manages memory testing for all attached controllers. For this setting, the clock frequency must be at least four times slower than the lowest memory BIST clock frequency. 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 in order to apply the timing exception on the netlist for static timing analysis.
- In the v2018.2 release, the SDC file will declare a generated clock at the clock buffer output, and will include 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.

Appendix J

Getting Help

There are several ways to get help when setting up and using Tesson software tools. Depending on your need, help is available from documentation, online command help, and Mentor Graphics Support.

The Tesson Documentation System	713
Mentor Support Services.....	714

The Tesson 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 “[Documentation Options](#)” in the *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 Tesson tool with the -manual invocation switch.
- **File System** — Access the Tesson InfoHub or PDF bookcase directly from your file system, without invoking a Tesson tool. For example:

HTML:

```
firefox $MGC_DFT/docs/infohubs/index.html
```

PDF

```
acroread $MGC_DFT/docs/pdfdocs/_bk_tesson.pdf
```

- **Application Online Help** — You can get contextual online help within most Tesson 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.

Mentor Support Services

Mentor provides a range of industry-leading support services that keep design teams productive and up-to-date with Mentor products.

A Mentor support contract includes the following:

- **Software Updates** — Get the latest releases and product enhancements to keep your environment current.
- **Mentor Graphics Support Center** — Access our online knowledge base, personalized to your Mentor products.
- **Support Forums** — Learn, share, and connect with other Mentor users.
- **Technical Support** — Collaborate with Mentor support engineers to solve complex design challenges.
- **Regular Communications** — Receive the latest knowledge base articles and announcements for your Mentor products.
- **Mentor Ideas** — Share ideas and vote for your favorites to shape future products.

More information is available here:

<https://support.mentor.com>

If your site is under a current support contract, but you do not have a Support Center login, register today:

<https://support.mentor.com/register>

Third-Party Information

For information about third-party software included with this release of Tesson products, refer to the [*Third-Party Software for Tesson Products*](#).

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/eula

IMPORTANT INFORMATION

USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. ORDERS, FEES AND PAYMENT.

- 1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (each an "Order"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not those documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented in any electronic portal or automated order management system, whether or not required to be electronically accepted, will not be effective unless agreed in writing and physically signed by an authorized representative of Customer and Mentor Graphics.
- 1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice. Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.
- 1.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation, setup files and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Except for Software that is embeddable ("Embedded Software"), which is licensed pursuant to separate embedded software terms or an embedded software supplement, Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 4.2); (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer provides any feedback or requests any change or enhancement to Products, whether in the course of receiving support or consulting services, evaluating Products, performing beta testing or otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

3. BETA CODE.

- 3.1. Portions or all of certain Software may contain code for experimental testing and evaluation (which may be either alpha or beta, collectively “Beta Code”), which may not be used without Mentor Graphics’ explicit authorization. Upon Mentor Graphics’ authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. Mentor Graphics may choose, at its sole discretion, not to release Beta Code commercially in any form.
- 3.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer’s use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer’s evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.
- 3.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer’s feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 3.3 shall survive termination of this Agreement.

4. RESTRICTIONS ON USE.

- 4.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except for Embedded Software that has been embedded in executable code form in Customer’s product(s), Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer’s employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer becomes aware of such unauthorized disclosure or use. Customer acknowledges that Software provided hereunder may contain source code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such source code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, disassemble, reverse-compile, or reverse-engineer any Product, or in any way derive any source code from Software that is not provided to Customer in source code form. Log files, data files, rule files and script files generated by or for the Software (collectively “Files”), including without limitation files containing Standard Verification Rule Format (“SVRF”) and Tcl Verification Format (“TVF”) which are Mentor Graphics’ trade secret and proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Products or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.
 - 4.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use, or as permitted for Embedded Software under separate embedded software terms or an embedded software supplement. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer’s employees or on-site contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.
 - 4.3. Customer agrees that it will not subject any Product to any open source software (“OSS”) license that conflicts with this Agreement or that does not otherwise apply to such Product.
 - 4.4. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense, or otherwise transfer the Products, whether by operation of law or otherwise (“Attempted Transfer”), without Mentor Graphics’ prior written consent and payment of Mentor Graphics’ then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics’ prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics’ option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer’s permitted successors in interest and assigns.
 - 4.5. The provisions of this Section 4 shall survive the termination of this Agreement.
5. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer with updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics’ then current End-User Support Terms located at <http://supportnet.mentor.com/supportterms>.
 6. **OPEN SOURCE SOFTWARE.** Products may contain OSS or code distributed under a proprietary third party license agreement, to which additional rights or obligations (“Third Party Terms”) may apply. Please see the applicable Product documentation (including license files, header files, read-me files or source code) for details. In the event of conflict between the terms of this Agreement

(including any addenda) and the Third Party Terms, the Third Party Terms will control solely with respect to the OSS or third party code. The provisions of this Section 6 shall survive the termination of this Agreement.

7. LIMITED WARRANTY.

- 7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification, improper installation or Customer is not in compliance with this Agreement. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."
 - 7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.
8. **LIMITATION OF LIABILITY.** TO THE EXTENT PERMITTED UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

9. THIRD PARTY CLAIMS.

- 9.1. Customer acknowledges that Mentor Graphics has no control over the testing of Customer's products, or the specific applications and use of Products. Mentor Graphics and its licensors shall not be liable for any claim or demand made against Customer by any third party, except to the extent such claim is covered under Section 10.
- 9.2. In the event that a third party makes a claim against Mentor Graphics arising out of the use of Customer's products, Mentor Graphics will give Customer prompt notice of such claim. At Customer's option and expense, Customer may take sole control of the defense and any settlement of such claim. Customer WILL reimburse and hold harmless Mentor Graphics for any LIABILITY, damages, settlement amounts, costs and expenses, including reasonable attorney's fees, incurred by or awarded against Mentor Graphics or its licensors in connection with such claims.
- 9.3. The provisions of this Section 9 shall survive any expiration or termination of this Agreement.

10. INFRINGEMENT.

- 10.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to such action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.
- 10.2. If a claim is made under Subsection 10.1 Mentor Graphics may, at its option and expense: (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.
- 10.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) OSS, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such OSS; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.
- 10.4. THIS SECTION 10 IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, FOR DEFENSE,

SETTLEMENT AND DAMAGES, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

11. TERMINATION AND EFFECT OF TERMINATION.

- 11.1. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.
- 11.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination of this Agreement and/or any license granted under this Agreement, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.
12. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and European Union ("E.U.") and United States ("U.S.") government agencies, which prohibit export, re-export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export or re-export Products in any manner without first obtaining all necessary approval from appropriate local, E.U. and U.S. government agencies. If Customer wishes to disclose any information to Mentor Graphics that is subject to any E.U., U.S. or other applicable export restrictions, including without limitation the U.S. International Traffic in Arms Regulations (ITAR) or special controls under the Export Administration Regulations (EAR), Customer will notify Mentor Graphics personnel, in advance of each instance of disclosure, that such information is subject to such export restrictions.
13. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. The parties agree that all Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to U.S. FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. government or a U.S. government subcontractor is subject solely to the terms and conditions set forth in this Agreement, which shall supersede any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.
14. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.
15. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 15 shall survive the termination of this Agreement.
16. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the U.S. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, U.S., if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America or Japan, and the laws of Japan if Customer is located in Japan. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply, or the Tokyo District Court when the laws of Japan apply. Notwithstanding the foregoing, all disputes in Asia (excluding Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
17. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
18. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. Any translation of this Agreement is provided to comply with local legal requirements only. In the event of a dispute between the English and any non-English versions, the English version of this Agreement shall govern to the extent not prohibited by local law in the applicable jurisdiction. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.