AppNote 4202

A P P N O T E S[SM]

Creating Artisan's Recommended Memory BIST Hardware with MBISTArchitect[TM]

By Scott Cook, Durga Prasad
Last Modified: November 4, 2004

©Copyright Mentor Graphics Corporation 1995-2004. 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.

Trademarks that appear in Mentor Graphics product publications that are not owned by Mentor Graphics are trademarks of their respective owners.

Abstract
Introduction
Artisan's Recommendations
Creating MBIST Hardware for Recommended Algorithms
- Memory Model Creation
- March-LR
- March-LR2
- Checkerboard Retention Test
- Address Decoder Open Test
- Write Mask Test
- Two Port Isolation Test

Creating BISA Hardware for Artisan Memories

Abstract

The application notes outlines how to create Memory Built-in-self-test (BIST) hardware that applies the six algorithms as recommended by Artisan and Built-in-self-repair (BISA) hardware that applies to Artisan redundant rows and column memories. To understand how BISA and repairable memories operate together, please refer to "BISA Repair using MBISTArchitect". MBISTArchitect supports the five highest priority algorithms required by Artisan. These memory-testing algorithms are the March-LR, March-LR2, Checkerboard Retention Test, Address Decoder Open Test and Write Mask Test. This application note provides examples of user defined algorithm files for the March-LR and March-LR2 algorithms. It discusses address scrambling and address jumping for fast row and column

addressing. Checkerboard Retention Testing and Address Decoder Open Testing are standard algorithms supported by MBISTArchitect and do not require UDA files.

Described below in Tabular format the Artisan recommended algorithms and MBISTArchitect support.

Sl.No	Artisan Recommended Algorithms and Redundant Rows and Column Memories	MBISTArchitect Support	License Requirement	Available Version
1	March-LR	Supported	MBISTA + MBIST Flex™	V8.2004_4
2	March-LR2	Supported	MBISTA +MBIST Flex™	V8.2004_4.
3	Checkerboard Retention Test	Supported	MBISTA +MBIST Flex™	V8.2004_4
4	Address Decoder Open Test	Supported	MBISTArchitect	V8.2004_4
5	Write Mask Test	Supported	MBISTA + MBIST Flex™	BETA (v8.2004_4)
6	Two Port Isolation Test	Not Supported	TBD	TBD
7	Redundant Column Repair	Supported	MBISTArchitect	V8.2004_4
8	Redundant Row Repair	Not Supported	TBD	TBD
9	Redundant Column and Row Repair	Not Supported	TBD	TBD

Introduction

The Artisan recommended algorithms are:

- March-LR
- March-LR2
- Checkerboard Retention Test
- Address Decoder Open Test
- Write-mask Test
- Two Port Isolation Test

These Algorithms are applicable for testing single and dual port SRAM as well as register files with MBISTArchitect.

This application note does not cover the general use of MBISTArchitect. General information regarding the operation of MBISTArchitect can be found in the MBISTArchitect Process Guide, which is part of the MBISTArchitect standard documentation. The purpose of this application note is to outline how to create hardware that applies the Artisan specific memory testing algorithms listed above and repair.

While this application note outlines how to create memory BIST hardware for six specific algorithms, there may be other methods to effectively test Artisan memories. Depending on the

application, size of the device and specific memory type, additional or different testing may be required.

Artisan's Recommendations

This document is based on information compiled from a series of Artisan's application notes. These application notes outline specific algorithms as well as a prioritized application sequence for testing single and dual port SRAMs as well as register files. Artisan makes the following recommendations.

Single Port SRAM Testing Algorithms & Prioritized Testing Sequence:

1. March-LR
2. Checkerboard Retention Test
3. Address Decoder Open Test
4. Write-mask Test

Dual Port SRAM Testing Algorithms & Prioritized Testing Sequence:

1. March-LR2
2. Two Port Isolation Test
3. Checkerboard Retention Test
4. Address Decoder Open Test
5. Write-mask Test

Single Port Register File Testing Algorithms & Prioritized Testing Sequence:

1. March-LR
2. Checkerboard Retention Test
3. Address Decoder Open Test
4. Write-mask Test

Dual Port Register File Testing Algorithms & Prioritized Testing Sequence:

1. March-LR
2. Two Port Isolation Test
3. Checkerboard Retention Test
4. Address Decoder Open Test
5. Write-mask Test

For each of the memory-types listed above the algorithms are listed in priority order. This means the first algorithm in each list is most effective, followed by the second, etc.

Creating MBIST Hardware for Recommended Algorithms

Two tasks must be done to create MBIST Hardware for Artisan specific memory testing algorithms listed in the previous section.

1. A proper memory model must be created. This memory model may require address scrambling statements as well as an address increment statement. The need for these statements will depend on the type of memory that is being tested

2. For non-standard algorithms, a User Defined Algorithm (UDA) file must be created. Examples of these files for Artisan specific algorithms are shown in this application note.

Memory Model Creation

Memory model creation is a very important step in the creation of memory BIST with MBISTArchitect. The memory model provides necessary information to the MBISTArchitect tool that is used to create the BIST Controller. MBISTArchitect memory models are usually provided by the memory vendor, however if the vendor does not provide these models it can be created easily by following the instructions in the MBISTArchitect documentation.

This application note will only focus on aspects of the MBISTArchitect memory model that are important for Artisan memory testing. The application note provides an overview of

- Address scrambling as it is used for fast column addressing.

- Address increment statement as it is used for fast row addressing. Fast row and column addressing is important for the March-LR and March-LR2 algorithms and

- Repair strategy to be used

The Checkerboard Retention test will also require a memory model that contains the top_column and top_word statements. To identify the number of words in each row the bist definition in the memory model should include the ***top_column*** statement. For example if the memory has 16 words per row, then the top_column statement would be "top_column = 16."

To make sure the memory is actually filled with a checkerboard. The multiplexing scheme should also be described in the memory model. This can be done by adding the ***top_word*** statement to the bist definition, which is in the memory model. The default is for top_word to equal 0. This means that the 010101…. Vector is put onto the data lines. If the memory however is bit-wise multiplexed together then the 00000… and the 111111… vector is put onto the data lines. In both situations, the memory is filled with an alternating pattern of 1 and 0 so that a checkerboard is created.

To generate the BISA hardware, user need to define the repair strategy attribute in the memory model and "add bisa hardware" command in the dofile. The syntax for the same is defined below.

Library Syntax:
repair *logical_column [1| spare_columns] [bits| index]*

logical_column - Generated logic will OR each of the failing bits into a holding register that has the same width as the memory's data word.

spare_columns - Optional, defaults to 1. At end of BIST process, the number of 1-bits in the holding register must be less than or equal to spare_columns or the memory is considered non-repairable.

Bits| index – Defines how the result data will be clocked out.

- If bits, then the holding register will be clocked out bit-by-bit, MSB as first bit.
- If index, then the logical index of each 1 bit in the holding register will be clocked out with the index for the most significant 1 bit first.

NOTE: Currently MBISTArchitect supports only the logical column strategy

Address Scambling (Fast Column Addressing)

The March-LR and March-LR2 algorithms require fast column addressing. MBISTArchitect supports fast column addressing through the address scrambling statements inside the memory model. Vendor provided memory models will contain correct address scrambling statements.

Address scrambling is supported through the address and data scrambling definition portion of the bist definition inside the memory bist model. Figure 1 below shows the syntax for this statement.

```
model model_name (list_of_pins) (
  bist_definition (
    ...
    descrambling_definition (
      address (
            <descrambled_LSB_name> = <boolean_statement>;
            ...
            <descrambled_MSB_name> = <boolean_statement>;
      ) // end address bus section
      data_in (
            <descrambled_LSB_name> = <boolean_statement>;
            ...
            <descrambled_MSB_name> = <boolean_statement>;
      ) // end data input bus section
    ) // end descrambling definition
    ...
  ) // end BIST definition
) // end model description
```

Figure 1. Address Scrambling Definition Syntax

This syntax will allow the user to specify the addressing order for writing and reading during the Memory BIST test.

As an example, if the required address order is to be 0x000. 0x001,0x003, 0x002, 0x004,… Then the address scrambling definition shown in figure 2 would be required.

NOTE: *Not all memories require address or data scrambling. It is suggested that you contact Artisan to identify which memories require data scrambling in the memory model.*

Figure 2. Describing Example for Address Order of 0,1,3,2,…

```
Descrambling_definition{
        Address {
                Dsc_a0 = a<1> xor a<0>;
                Dsc_a1 = a<1>;
                Dsc_a2 = a<2>;
                Dsc_a3 = a<3>;
        }
        Data_in{
                Dsc_d0 = d<0>;
                Dsc_d1 = d<1>;
        }}
```

NOTE: *More information on address scrambling can be found in the MGC MBIST documentation.*

Address Incrementation (Fast Row Addressing)

When fast row addressing is required, the addr_inc statement can be added to the bist definition portion of the MBISTArhcitect memory model. This number associated with the addr_inc statement is utilized to calculate what the next address should be every time a "jump" statement is used inside the user defined algorithm file. The user defined algorithm file will be discussed later in this application note. Please see the MBISTArchitect documentation for more information on the syntax of the bist definition statement.

The March-LR and March-LR2 algorithms require fast row addressing. Addr_inc statements are included in the memory model provided by the vendor.

March-LR

MBISTArchitect supports the Artisan specific March-LR algorithm through the User Defined Algorithms (UDA) MBIST Flex™ option. The March-LR algorithm requires both fast column and fast row addressing. The reader should check with Artisan to understand if their specific memory requires address scrambling and address increment statements. If the memory was obtained from Artisan, correct address scrambling and addr_inc statements will be included in the memory model. The example shown below uses the jump statement that requires the memory model to have an address increment statement.

Figure 3 illustrates how to add the March-LR UDA file before adding the March-LR algorithm to the controller. Figure 4 is the UDA file that can be used to create the March-LR Hardware. This file should be read into MBISTArchitect before adding the March-LR algorithm to the controller

```
add memory model 1portSRAM

load algorithms ./marchlr.uda

setup retention cycle 10

setup mbist algorithms march-lr retentioncb

set bist insertion -on
set bsda -on
setup mem clock –control

set controller naming -module 1portSRAM_bist
set file naming -bist_model 1portSRAM_bist.v
set file naming -connected_model 1portSRAM_bist_con.v
set file naming -testbench 1portSRAM_bist_tb.v
set file naming -script 1portSRAM_bist.v_dcscript
set file naming -ctdl 1portSRAM_bist.v.ctdf
set file naming -wgl 1portSRAM_bist.wgl
run
save bist -verilog -script -replace
exit –d
```

Figure 3. Loading the March-LR UDA File

```
// Algorithm:
//          March-LR //
// Summary:
//          two pass test
//          Address order is dependent on the address scambling portion of memory model
//          "jump" is equal to the addr_inc value in the memory model
//
// Pass 1: all 0 and all 1
//     write 0 - up
//     read 0, write 1 - down
//     read 1, write 0, Read 0, Read 0, write 1 - up
//     read 1, write 0 - down
//     read 0, write 1, read 1, read1, write 0 - up
//     read 0 - down
//
// Pass 2: "5" and "A" (0101 and 1010)
//     write 5 - up
//     read 5, write A - down
//     read A, write 5, Read 5, Read 5, write A - up
//     read A, write 5 - down
//     read 5, write A, read A, read A, write 5 - up
//     read 5 - down

step step1_pass1;
  addr min, max, up, 1;
  data seed;
  operation w;

step step2_pass1;
  addr min, max, down, 1;
  data invSeed;
  operation rw;

step step3_pass1;
  addr min, max, up, 1;
  data seed;
  operation rwrrw;

step step4_pass1;
  addr min, max, down, 1;
  data invSeed;
  operation rw;

step step5_pass1;
  addr min, max, up, 1;
  data seed;
  operation rwrrw;

step step6_pass1;
  addr min, max, down, 1;
  data seed;
  operation r;

step step1_pass2;
  addr min, max, up, jump;
  data seed;
  operation w;

step step2_pass2;
  addr min, max, down, jump;
  data invSeed;
  operation rw;

step step3_pass2;
  addr min, max, up, jump;
  data seed;
  operation rwrrw;

step step4_pass2;
  addr min, max, down, jump;
  data invSeed;
  operation rw;

step step5_pass2;
  addr min, max, up, jump;
  data seed;
  operation rwrrw;

step step6_pass2;
  addr min, max, down, jump;
  data seed;
  operation r;

repetition pass_1;
  seed 0000;
begin
step step1_pass1;
step step2_pass1;
step step3_pass1;
step step4_pass1;
step step5_pass1;
step step6_pass1;
end

repetition pass_2;
  seed 0101;
begin
step step1_pass2;
step step2_pass2;
step step3_pass2;
step step4_pass2;
step step5_pass2;
step step6_pass2;
end

test march-lr;
begin
  repetition pass_1;
  repetition pass_2;
end
```

Figure 4. UDA File for March-LR Algorithm

March_LR2

In general, complex multi-port algorithms are not supported by the UDA inside MBISTArchitect. However, by splitting the March_LR2 algorithm into two parts and applying each part to a different port, the UDA will support March_LR2.

To use a UDA file for March-LR2 the following steps should be followed

STEP 1: Divide the March-LR2 algorithm into two parts. One part will contain the fast column addressing with data background 0 and 1. The other part will contain the fast row addressing with data backgrounds 5 and A.

STEP 2: use the ***"add algorithm"*** command and specify the first part for port A and then specify the second part for port B. This will make sure the correct data background is written to the correct port.

STEP 3: Make sure to use the ***"setup memory access –simultaneous"*** command. This will make sure the controller reads from both ports.

NOTE: The **"–simultaneous"** switch is the default operation of MBISTArchitect. If the memory doesn't allow read from more than one port to the same address, then the set memory access command should be used to disallow simultaneous reading.

A sample dofile is shown in figure 5 below that outlines the steps shown above.

```
add memory model 2port_RAM

load algorithms ./marchlr2.uda

setup memory access -simultaneous

setup mbist algorithms  1 march-lr-A 2 march-lr-B
set bist insertion -on
set bsda -on
setup mem clock -control

set controller naming -module 2port_RAM_bist
set file naming -bist_model 2port_RAM_bist.v
set file naming -connected_model 2port_RAM_bist_con.v
set file naming -testbench 2port_RAM_bist_tb.v
set file naming -script 2port_RAM_bist.v_dcscript
set file naming -ctdl 2port_RAM_bist.v.ctdf
set file naming -wgl 2port_RAM_bist.wgl
run
save bist -verilog -script -replace
exit -d
```

Figure 5. Dofile for March-LR2 Algorithm

Shown below is the UDA file that can be used to create the March-LR2 Hardware. The MBISTArchitect commands used to load this file is the same as the commands used to load the March-LR algorithm from the previous section.

```
// Algorithm:
//          March-LR2-A
//
// Summary:
//
// Pass 1: all 0 and all 1
//      write 0 - up A
//      read 0 A/B, write 1 - down A
//      read 1 A/B, write 0, Read 0 A/B, Read 0 A/B, write 1 - upA
//      read 1 A/B, write 0 - down A
//      read 0 A/B, write 1, read 1 A/B, read 1 A/B, write 0 - upA
//      read 0 A/B - down A
step step1_pass1;
  addr min, max, up, 1;
  data seed;
  operation w;

step step2_pass1;
  addr min, max, down, 1;
  data invSeed;
  operation rw;

step step3_pass1;
  addr min, max, up, 1;
  data seed;
  operation rwrrw;

step step4_pass1;
  addr min, max, down, 1;
  data invSeed;
  operation rw;

step step5_pass1;
  addr min, max, up, 1;
  data seed;
  operation rwrrw;

step step6_pass1;
  addr min, max, down, 1;
  data seed;
  operation r;

repetition pass_1;
  seed 0000;
begin
step step1_pass1;
step step2_pass1;
step step3_pass1;
step step4_pass1;
step step5_pass1;
step step6_pass1;
end

test march-lr-A;
begin
  repetition pass_1;
end

// Algorithm:
//          March-LR2-B
//
// Summary:
//          part 2 of the march-LR2 test
//
// Pass 2: "5" and "A" (0101 and 1010)
//      write 5 - upB
//      read 5 A/B, write A - down B
//      read A A/B, write 5, Read 5 A/B, Read 5 A/B, write A - upB
//      read A A/B, write 5 - down B
//      read 5, write A, read A A/B, read A A/B, write 5 - up B
//      read 5 - down B

step step1_pass2;
  addr min, max, up, 1;
  data seed;
  operation w;

step step2_pass2;
  addr min, max, down, 1;
  data invSeed;
  operation rw;

step step3_pass2;
  addr min, max, up, 1;
  data seed;
  operation rwrrw;

step step4_pass2;
  addr min, max, down, 1;
  data invSeed;
  operation rw;

step step5_pass2;
  addr min, max, up, 1;
  data seed;
  operation rwrrw;

step step6_pass2;
  addr min, max, down, 1;
  data seed;
  operation r;

repetition pass_2;
  seed 0101;
begin
step step1_pass2;
step step2_pass2;
step step3_pass2;
step step4_pass2;
step step5_pass2;
step step6_pass2;
end

test march-lr2-b;
begin
  repetition pass_2;
end
```

Figure 6. UDA File for March-LR2 Algorithm

Checkerboard Retention Test

MBISTArchitect supports Checkerboard Retention Testing as a standard algorithm. A UDA file is not required for this algorithm. However, this algorithm does depend on the structure of the memory. The memory model must contain information about the number of words in each row as well as how the words are multiplexed together to form columns.

To identify the number of words in each row the bist definition in the memory model should include the ***top_column*** statement. For example if the memory has 16 words per row, then the top_column statement would be "top_colum = 16."

To make sure the memory is actually filled with a checkerboard. The multiplexing scheme should also be described in the memory model. This can be done by adding the ***top_word*** statement to the bist definition, which is in the memory model. The default is for top_word to equal 0. This means that the 010101…. Vector is put onto the data lines. If the memory however is bit-wise multiplexed together then the 00000… and the 111111… vector is put onto the data lines. In both situations, the memory is filled with an alternating pattern of 1 and 0 so that a checkerboard is created.

Once the memory model has these statements added to the BIST definitions, then the "***add mbist algorithm"*** command can be used to add the retention test to the memory bist hardware.

To specify the length of the retention period, the "***setup retention cycle"*** command should be used.

Address Decoder Open Test

MBISTArchitect support Address Decoder Open Testing as a standard algorithm. This means that a UDA test algorithm is not required in order to incorporate this test into your memory BIST controller.

To add the Address Decoder Open Test, the user should simply use the *add mbist algorithm* command to add this algorithm to the MBIST controller. The user can also use the *setup mbist algorithm* command with this algorithm.

The following is an example of how to add this algorithm to a controller.

```
MBISTA> add mbist algorithm 1 addressdecoder
```

For more information regarding this type of algorithm, the user is directed to read the Mentor Graphics MBISTArchitect documentation.

Write Mask Test

MBISTArchitect does support Write Mask Algorithm; currently it is available as BETA release. The production release is schedule for v8.2004_6.

Please contact DFT Customer Support for more information on this feature.

Two Port Isolation Test

MBISTArchitect does not support Two Port Isolation Testing. Support for this test is expected in 2004.

Creating BISA Hardware for Artisan Memories

Two tasks must be done to create BISA Hardware for Artisan Redundant rows and column memories.

- A proper memory model must be created. This model should have statement ***repair logical_column.*** This statement should also include number of redundant columns in the memory and reporting detail information bits or indexes. For more information on how to define this statement in the memory model, please refer "memory model creation" section in this AppNote.
- Define "add bisa hardware" command to instruct the tool to generate the BISA hardware. Details are explained in below section.

NOTE: Currently MBISTArchitect Supports only Column Strategy.

MBIST Command to generate BISA Hardware

The command "*add bisa hardware*" instruct the tool to generate the BISA hardware for only those memories defined in this command.

Command Syntax:

Add Bisa hardware *{-all |memory_number...}*

- *-All*
  An optional switch that directs the tool to generate BISA logic and necessary signals in the BIST module of each memory that has a repair strategy defined in the library. This is the default function of the command and is the same if no argument is supplied with the command.
- *memory_number…*
  An optional switch that selects defined memory for BISA generation. This switch is repeatable for defining other memories to be also selected for BISA generation. User should use this switch only when there are multiple memories and only some of them will be designed with repair capabilities.

MBIST Command to Report BISA Hardware

The command "*report bisa hardware*" reports BISA repair status for all memories.
The report list the following for each memory:

- Memory's number
- Flag Status: if repair function is active or not for memory
- Repair strategy for the memory
- Parameters of each strategy

Example:

```
// command: report bisa hardware
Built-In Self-Analysis Hardware Setup
Memory Active        Strategies/Parameters
   1          Y          logical_column / data_width=17, spares=1, report mode: bits
   2          Y          logical_column / data_width=17, spares=1, report mode: bits
```

Examples Dofile and Memory model Description for generating BISA Hardware

Dofile:

```
// Load MBIST models
load library memory.lib

// Add memory models
add memory model mem3

// Load User Define Algorithm (UDA)
load algorithm March_L_R

add mbist algorithm 1 March_L_R_1 March_L_R_2 retentionCB addressdecoder

Add data backgrounds 0101010101010101010101010101010101010101010101010

Set controller delay  1

set scan logic -addr_observe 8 -data_observe 43 –noscan

// Instructing the tool to add bisa hardware for all the memories described above
// using add memory model command
add bisa hardware –all

setup clock period 2

run

// reports the bisa hardware added to specified memories.
report bisa hardware

report algorithm steps

save bist –replace
```

Figure 7: Example dofile on how to generate BISA hardware

Memory model example:

```
  model mem3 (  QA, CLKA, CENA, AA, CLKB, CENB, AB,DB)
(
```

```
  bist_definition (
  address AA (array=7:0;);
  clock CLKA high;
  read_enable CENA low;
  data_out QA (array=50:0;);
  address AB (array=7:0;);
  data_in DB (array=50:0;);
  clock CLKB high;
  chip_enable CENB low;

  tech = fast;
  vendor = "Artisan Components Inc.";
  version = "1.0";
  message = " mem3 Synchronous Dual-Port Register File";
……
  address_size = 8;
  min_address = 0;
  max_address = 255;
  data_size = 51;

  repair logical_column  1 index;
…
read_port (
   read_cycle (
     change AA;
     assert CENA;
…
```

Figure 8: Example memory model showing the repair attribute definition

Memory BIST Controller with BISA Container:

The high level view of the BIST controller with BISA hardware is shown in below figure. To know more about BISA Hardware and its functionality, please refer "BISA Repair Using MBISTArchitect".

Simple BIST Controller Model:

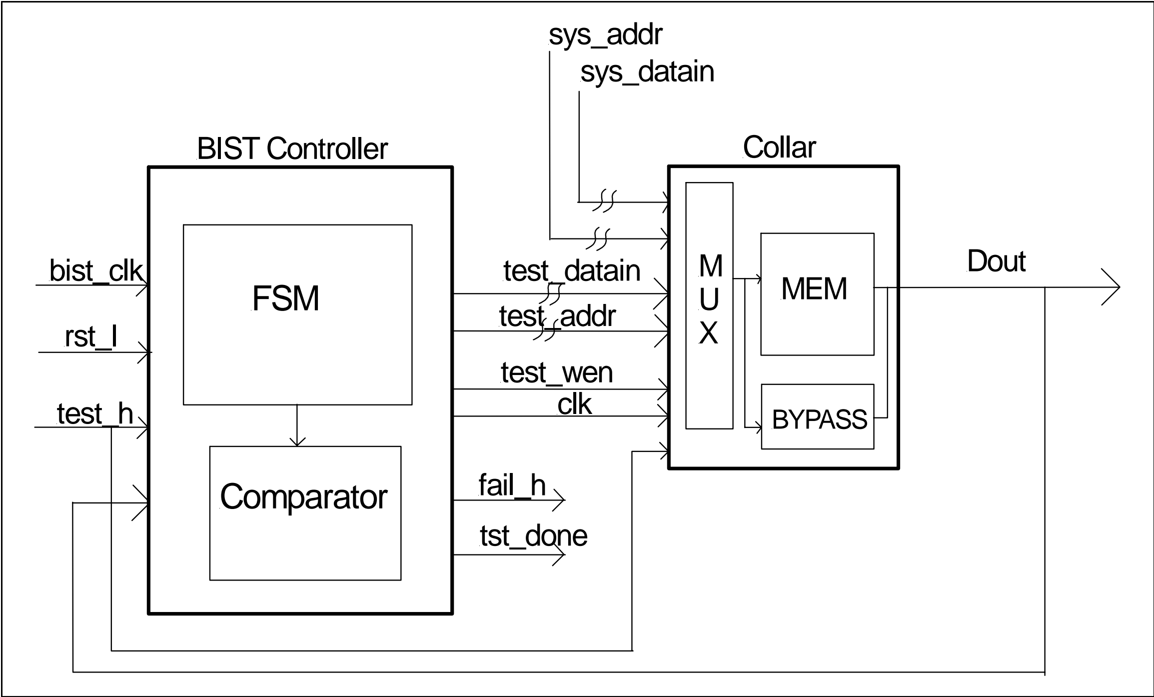

Fig 9: Simple BIST controller block

BIST Controller with BISA Hardware:

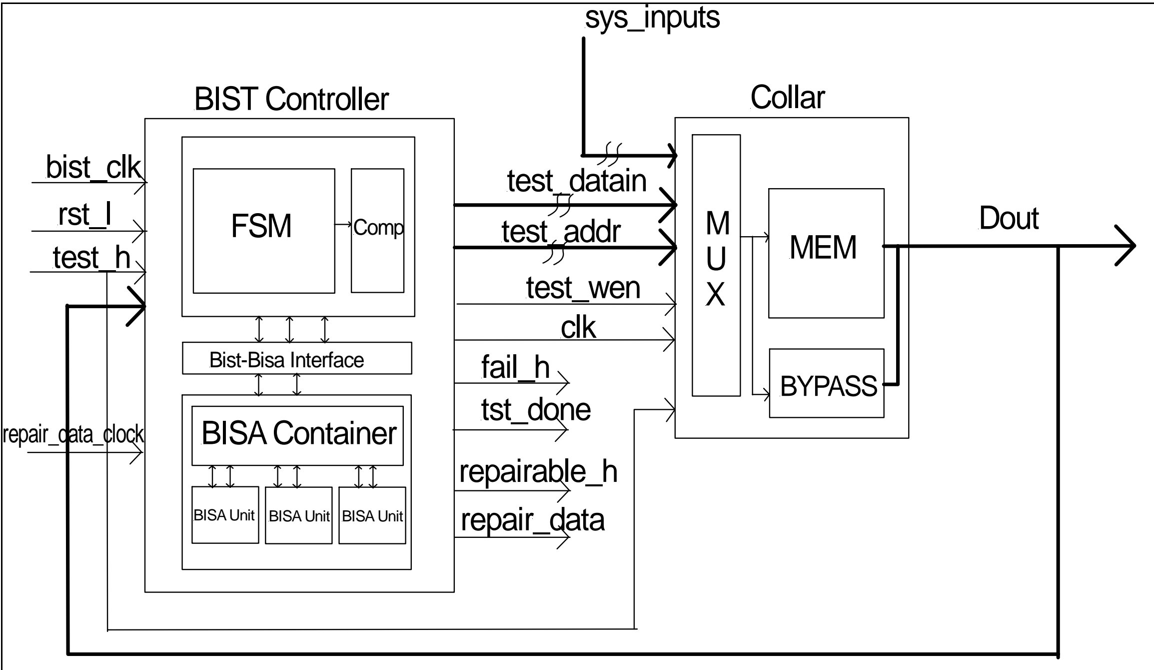

Fig 10: BIST Controller with BISA Hardware in it