BIST Implementation Access through A Reconfigurable Network

YAOJIE LU

MASTER´S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

BIST Implementation Access through A Reconfigurable Network

Yaojie Lu
ya0117lu-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor:
Erik Larsson
erik.larsson@eit.lth.se
Hemanth Prabhu
hemanth@xenergic.com

Examiner:
Pietro Andreani
pietro.andreani@eit.lth.se

September 3, 2019

Acknowledgements

First, I would like to thank my supervisor Prof. Erik Larsson(Lund University) and Hemanth Prabhu (Xenergic AB) for their continuous help and precious support through this project. I would like to express my gratitude to Xenergic for offering this great opportunity to me and also to my colleagues at Xenerigc for sharing their experience with me. At last but not least, I want to thank my family and friends. Without their love and support I am not able to overcome all the difficulties.

Abstract

SRAM is becoming one of the most dominant contributors to the power and area of nowadays SoC. Recent surveys show that on average around 70% of area budget of nowadays SoCs are occupied by SRAMs, with the size of these SRAMs ranging from a few kilo-bits to 10s of mega-bits [1]. One critical issue is to check for functionality of the memories in silicon both during the manufacturing (tester) stage and in-the-field (run time) stage. A well-known approach is to use a dedicated test logic to validate memory functionality, known as BIST.

The Memory Built-in Self-test method has become a mainstream test method with its unique advantages: good test defect coverage; strong operability and low dependence on testing equipment. Memory testing is often not done individually. It is also difficult to measure multiple different sizes of memories simultaneously using one BIST. The existing test interface usually lacks flexibility and scalability. Connecting multiple BISTs requires a highly flexible network which can easily access different BIST processors.

This project introduces two embedded memory BIST implementations: one programmable BIST and one hardcoded BIST. In addition, An IEEE Std.1687 IJTAG based network has been integrated and modified, in which TAP and the associated controller has been replaced by a UART port interface. This interface is a functional port to program BIST and to read error information out from the BIST. The system simply needs two pins to read and write data. Besides, it is flexible and scalable for scheduling access to multiple BISTs. In this project, RTL files are written in VHDL, whereas an instruction converting tool and a re-targeting tool are designed in Python.

Popular Science Summary

With the development and advancement of technology, people's dependence on electronic products continuously grow. Embedded memories play a crucial role in electronic devices. According to Moore's Law, the number of transistors on a chip roughly doubles every two years, which will lead to decreasing of the memory cell area, increasing of density [2]. However, as the density of memory cell increases, faults in memory will also increasingly sensitive and failures of chips will be more complex. The consequence is increasing cost and time of the test to detect these failures. Due to these existing factors, test cost cannot decrease with declining in the price of memory. The way to deduct the test cost on premises but ensuring test efficiency is the main point during the development of a new-generation integrated circuit. For memory tests, three main test methods commonly used are direct access test, embedded CPU test, and Memory Built-in Self-test. Built-in self-test can more effectively tackle the SoC test problem.

This master thesis introduces two BIST implementations: a hardcoded BIST and a programmable BIST which can handle common types of memory faults. Theoretically, one BIST can only test one size of memory. One SoC could contain many different sizes of memories, meaning that a number of BISTs is needed. This is problematic because the SoC will need a lot of pins. One solution is using an network to connect theses BISTs which saves pins and configures multiple BISTs. This thesis also introduces an IEEE Std.1687 IJTAG based network which has flexibility and scalability that allows configurations to multiple BISTs.

Arconyms

BIST	Built-in self-test
SRAM	Static Random Access Memory
SoC	System on a chip
MBIST	Memory Built-in self-test
DFT	Design for testing
RAM	Random Access Memory
DRAM	Dynamic Random Access Memory
AF	Address-Decoder Fault
SAF	Stuck-At Fault
TF	Transition Fault
CF	Coupling Fault
ATE	Automated Test Equipment
FSM	Finite State Machine
SIB	Segment Insection Bit
MSB	Most Significant Bit
LSB	Least Significant Bit
SPI	Serial Peripheral Interface
JTAG	Joint Test Action Group
UART	Universal Asynchronous Transmitter Receiver
TAP	Test Access Port
HIP	Hierarchical Interface Interface
OAT	Overall Access Time

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In nowadays system on chip (SoC) designs, it is widespread to embed a large number of static memories into the chip. According to statistics, the area of memories accounts for over 50% to 60% of the total area of the chip [3]. The size of on-chip memory can be in the range of few kilobits to megabits (e.g. L2 cache). Large memory sizes with increasing density of memory cells (e.g. pushed rules in bit cells), can affect failure rate or yield severely. This makes failure detection, testing, and repair of SRAMs very critical. The testing process is time-consuming, which means it is essential to minimize test application time. However, costs as silicon area, equipment and interfaces need to be co-optimized with test application time [4]. There are three main methods for memory testing [5]: direct access test, embedded CPU test, and memory BIST. Among the three methods, MBIST is becoming a prevalent method to perform SRAM testing. The basic idea is to generate the test vector by the BIST instead of using specified external test circuit, and it relies on its own logic to determine if the test results obtained are correct, thus significantly reducing the requirement for test equipment, MBIST also lowers the number of pins and allow for high speed. Due to these advantages, the MBIST method is widely used [6].

1.1 Motivation

MBIST can be broadly classified into two types, one is hardcoded BIST, which generally hardwired with pre-defined algorithms. The advantage of this BIST is evident that it saves area. And it also need less test time since the test algorithms are in-built. However, it also has a bottleneck that lacks flexibility. Another MBIST is programmable BIST, which can encode basic memory operations provides more flexibility but with a higher area cost.

In this thesis, we attempt to design two different MBIST and find an interface with fewer pins and highly dynamic re-configurable.

1.2 Thesis Specification

The aim of the thesis is to design a system which is shown in Figure 1.1. The idea is to select and send instructions through a functional port interface to different BIST processor. Multiple instances of BIST processor which can be either hardcoded or programmable. A memory wrapper which contains multiple memories under test will connect with BIST, after writing and reading from memories, the BIST will send error data out from the system.

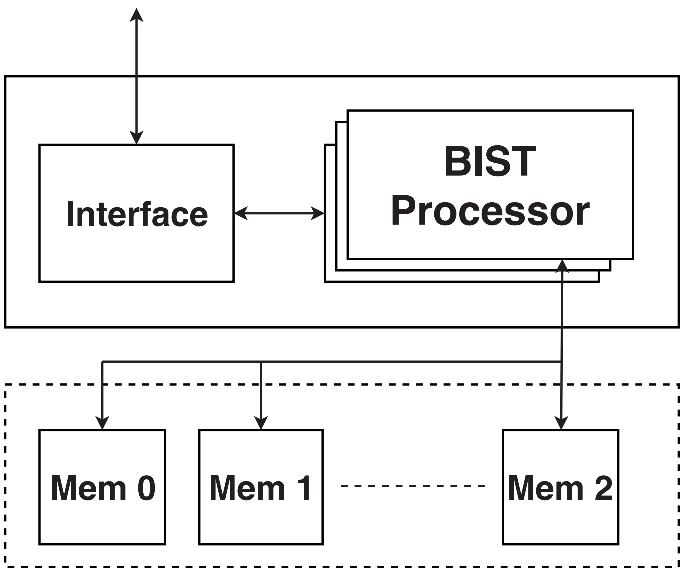

Figure 1.1: System level view of BIST processor.

1.3 Thesis organization

The thesis is organized in following manner:

- ***Chapter 2:*** Background for SRAM and basic concept for BIST with relevant March based algorithms
- ***Chapter 3:*** Implementation of a programmable BIST with instruction sets and a hardcoded BIST with 8 pre-defined algorithms
- ***Chapter 4:*** Introduction for existing protocol for DFT and interface be used in this project

- ***Chapter 5:*** Verification of the system and comparison of programmable BIST and hardcoded BIST
- ***Chapter 6:*** Conclusions of the report
- ***Chapter 7:*** Future work for optimizing the whole system and integrating programmable BIST into the interface

Chapter 2

Thesis Background

2.1 Static Random-Access Memory

SRAM is widely used as embedded memory (e.g. L1 and L2 caches in processors and data buffers in various DSP chips) since RAM can stores data bits in its memory when power is being supplied. Unlike DRAM, which stores bits in cells consisting of a capacitor and a transistor, SRAM does not have to be periodically refreshed [7].

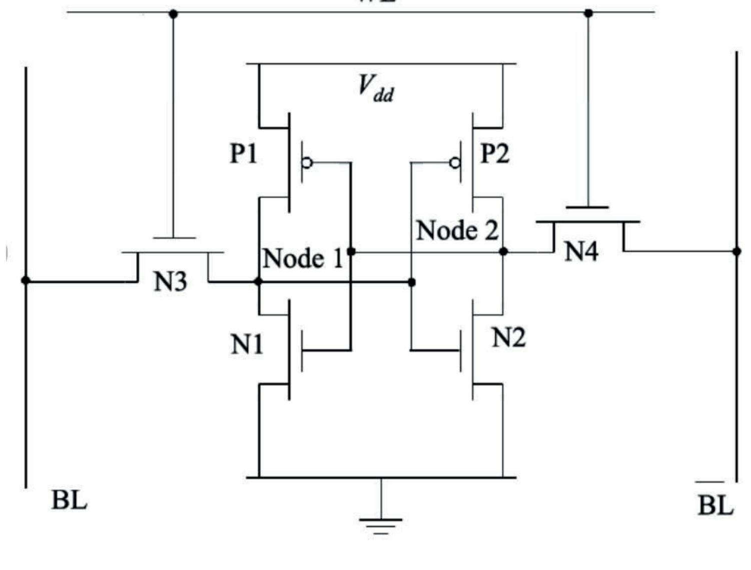

Figure 2.1: A conventional 6T single port SRAM [8].

Figure 2.1 shows a 6T SRAM cell consists of two cross-coupled CMOS inverters and two transistors. P1, P2, N1, N2, which are responsible for storing the data, as long as the memory is powered. The other 2 transistors, Q5 and Q6 serve as gateways for bidirectional access between stored data and the Bit Lines. These access transistors are controlled with the word line.

A basic SRAM memory structure is shown in Figure 2.2, each one of the memory cells have a unique memory address; these show where the cell is. The total memory address is the sum of the number of bits of the row address and the number of bits of the column address. These addresses must be sent to the row and column decoders for selecting a memory cell. However, the decoders will not be the focus of this post because each row and column will be accessed individually, for testing purposes [9].

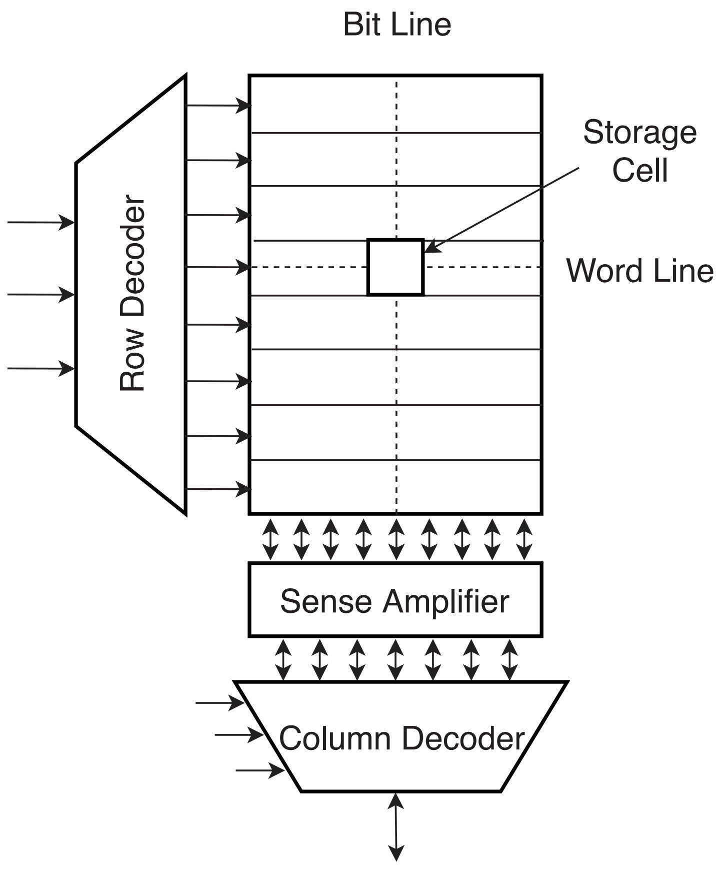

Figure 2.2: Basic SRAM Architecture [8].

2.2 Memory Faults

With the increasing density of the memory cell, the memory becomes more sensitive. Meanwhile, the risk of memory fault increase. In this section, some major functional fault will be introduced [10].

- ***Address-Decoder Fault (AF):*** Any fault that affects address decoder can be considered as AF(e.g.With a certain address, no cell will be accessed)

- ***Stuck-At Fault (SAF):*** The SAF considers that the logic value of a cell or line is always 0(stuck-at 0 or SA0) or always 1(stuck-at 1 or SA1)

- ***Transition Fault (TF):***The TF is a special case of the SAF. A cell or line that fails to undergo a 0 to 1 transition after a write operation is said to contain an up transition fault. Similarly, a down transition fault indicates the failure of making a 1 to 0 transition.

- ***Coupling Fault (CF):***A write operation to one cell changes the content of a second cell.

2.3 Built-In Self-Test

BIST is a technique which integrated additional functional block into the circuit to allow them to perform self-testing, so it can reduce dependence on an external ATE. As shown Figure 2.3, a general BIST architecture usually consists of four logic blocks. Test controller responsible for controlling the BIST system. Test generator which generates address sequences and writes a pattern to the CUT. Response verification works as a comparator which read the data out from CUT and makes comparisons with expected data. CUT, in this case, is mainly the memory under test.

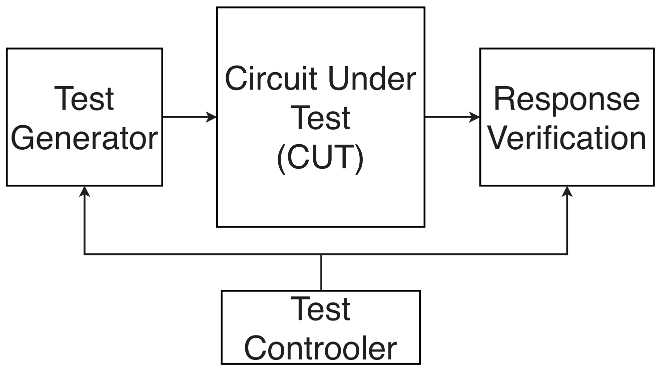

Figure 2.3: General BIST Architecture.

2.4 March Test Algorithms

March based test algorithms are the most commonly used memory testing methods because they are all simple and possess good fault coverage. These algorithms consist of finite operation sequences which given read and write an order to the memory. For example:

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

The sequence shown above is a typical March C- algorithm. w0 represent write a 0 to the memory location, r0 represent read a 0 from the memory location, w1 represents write a 1 to the memory location, r1 represent read a 1 from the memory location. (↑) Means increasing memory address order, (↓) means decreasing memory address order. This process of reading and write operations for the testing of memory was developed by Suk and Reddy [11]. Each March element will be applied to each cell in memory before proceeding to the next March element, which means if a w0 is applied to one cell, then it must be applied to all cells.

Table 2.1 list some common March-based algorithms [12]. Each algorithm consists of a specific read or writes operations that go through all memory locations in either an increasing or decreasing address order.

Table 2.1: March-based Algorithms.

Name	March Elements
MATS	{$\updownarrow$(w0); $\updownarrow$(r0, w1); $\updownarrow$(r1)}
MATS++	{$\updownarrow$(w0); $\uparrow$(r0, w1); $\downarrow$(r1, w0, r0)}
March X	{$\updownarrow$(w0); $\uparrow$(r0, w1); $\downarrow$(r1, w0); $\updownarrow$(r0)}
March C-	{$\updownarrow$(w0); $\uparrow$(r0, w1); $\uparrow$(r1, w0); $\downarrow$(r0, w1); $\downarrow$(r1, w0); $\updownarrow$(r0)}
March A	{$\updownarrow$(w0); $\uparrow$(r0, w1, w0, w1); $\uparrow$(r1, w0, w1); $\downarrow$(r1, w0, w1, w0); $\downarrow$(r0,w1, w0)}
March B	{$\updownarrow$(w0); $\uparrow$(r0, w1, r1); $\downarrow$(r1, w0, r0); $\updownarrow$(r0)}
March LR	{$\updownarrow$(w0); $\uparrow$(r0, w1); $\uparrow$(r1, w0, r0, w1); $\uparrow$(r1, w0); $\uparrow$(r0, w1, r1, w0); $\updownarrow$(r0)}
March Y	{$\updownarrow$(w0); $\uparrow$(r0, w1, r1, w0, r0, w1); $\uparrow$(r1, w0, w1); $\downarrow$(r1, w0, w1, w0);$\downarrow$(r0, w1, w0)}

Different combination of March elements will lead to different fault coverage. It is obvious that more complex algorithm gives better fault coverage. Table 2.2 shows the fault coverage of listing algorithms in Table 2.1.

Table 2.2: Fault coverage and operation count of different March-based algorithms.

Fault Coverage					Operation Count
	SAF	**AF**	**TF**	**CF**	
MATS	ALL	SOME			4.n
MATS++	ALL	ALL			6.n
March X	ALL	ALL	ALL	SOME	6.n
March C-	ALL	ALL	ALL	ALL	10.n
March A	ALL	ALL	ALL	SOME	15.n
March B	ALL	ALL	ALL	SOME	8.n
March LR	ALL	ALL	ALL	SOME	14.n
March Y	ALL	ALL	ALL	SOME	17.n

Chapter 3

BIST Implementation

This chapter will introduce two BIST implementation. Programmable BIST is a microcode-based BIST which can program different algorithms and select memory address location with predefined instruction sets. Hardcoded BIST is an FSM-based memory BIST which can run a selected memory test algorithms. In this design, the user has eight different algorithms to choose from.

3.1 Programmable BIST

3.1.1 Programmable BIST Architecture

The Programmable BIST processor has very minimalistic custom instructions and a mechanism for logging the faults in the memory. Based on the instruction set, end-users can also write their own algorithms for advanced debugging.

The processor comes with an assembler to translate to processor binary code. Also, during the translation, a linker is invoked to handle all the labeling, loops and address calculations. Figure 3.1 shows the top-level block diagram of the BIST processor. The BIST processor requires a very minimal interface connection to start the processor (fetch, start, etc.). Also, a standard interface is used to program the processor, i.e. SPI, JTAG or UART. The memory transactor pins are the ones which are connected to the memory under test. This can be either a direct connection to the memory or to a sub-system with corresponding address mapping.

Most of the modules in the BIST processor is typical to a standard RISC processor. However, memory transactor and error checker modules are very specific to testing. The memory transactor is responsible for initiating memory read/write for testing. The error checker module is used to compare the memory functionality. As of now the processor is designed only for a single clock read/write latency.

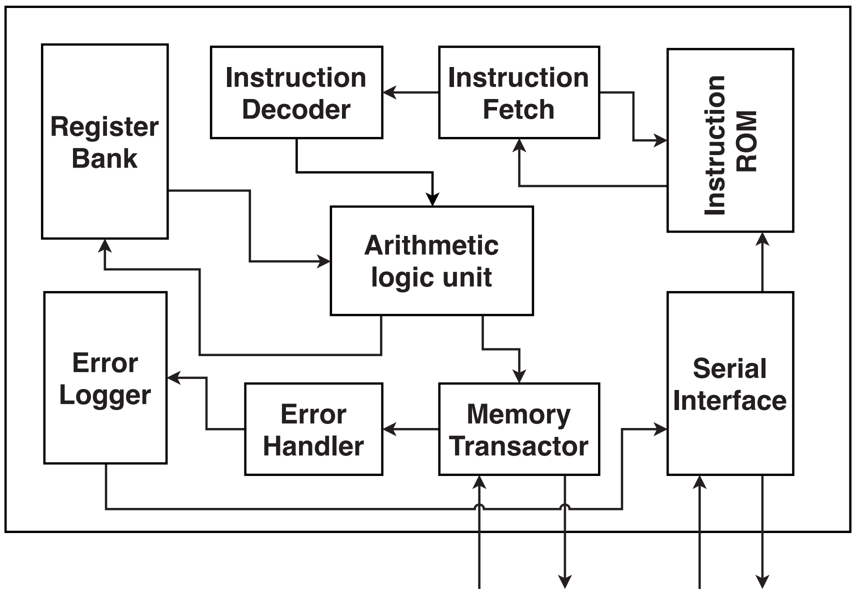

Figure 3.1: Programmable BIST Top Level Block Diagram.

3.1.2 Instruction Converting Tool

In this section, an instruction converting tool which transferred assembly to binary code will be explained in detail. A user can use this tool to generate a specific testing algorithm.

1. *mv ls, #start*
2. *mv le, #end*
3. *mv lc, #constant*
4. *mv wptr, all zero*
5. *mv wp, all zero*
6. *#start*
7. *wr++*
8. *#end*

Here we use incremental write 0 as an example: First, we need to move the loop start register to a pc location, then the loop end register moves to the pc where to the loop end. In step 3, the loop number can be assigned. After setting the register associated with a loop, we need to set the write pointer and write pattern value. Because the instruction is an incremental write 0, so the write pointer should start from address 0, and the write pattern value selects all 0. The start and end labels here indicate the starting and ending positions of the loop start, *wr++* is the incremental write operation which will be repeated until the number of times set in the loop count is reached. The instruction converting tool which is written in Python will convert these assembly commands into binary code and automatically generate a VHDL file.

3.1.3 Register View

This section provides details of the instruction set for the programmable BIST processor. The key idea with the processor instruction set is to make it as compact as possible to optimize for the program memory. Furthermore, to lower area cost, the processor is designed by avoiding expensive general-purpose register decoders. Instead, there are a bunch of dedicated registers. (e.g. loop registers, read/write pointers, etc).

Table 3.1: Register view of BIST processor.

Register Name	Symbol	Width	Description
Read Pointer	rptr	addr_width	Pointer to read from a memory location
Write Pointer	wptr	addr_width	Pointer to write to a memory location
Read Pattern	rp	data_width	Read value to the memory
Write Patren	wp	data_width	Write value to the memory
Max Size	mms	mem_size	Maximum value or size of memory under test
End Of Program	eop	inst_width	Value in the register indicates the end of routine
Control	cr	inst_width	Set the default limitation of numbers of errors

Overall the instructions are divided into two parts: single-cycle and multi-cycle instructions. Single-cycle instructions are of 4-bits wide and are executed in one cycle. Multi-cycle instructions opcode is also 4-bits, however, it is followed by the immediate operand. The immediate operand can vary from 4 bits to 36 bits.

Next section provides a brief description of the registers, which is followed by the list of instructions.

Table 3.1 provides a list of dedicated registers. The read and write pointers are used by the memory transactor to generate read/write address. The write pattern register is used by the memory transactor to generate write data for the memory under test. During the read operation, the data from the memory is compared with the value in the read pattern register. Based on the control register flags the read checker module will perform validation of the memory. Max size register is used to set the size of the memory under test. End of Program register is used to identify when a program or algorithm has completed. When the program counter of the processor is the same as the EOP register the processor halts and asserts the bist_done signal.

Register Name	Symbol	Width	Description
Program Counter	PC	16	Standard program counter of a processor.
Loop Start	LS	16	Indicates the starting location of the loop in the program memory.
Loop End	LE	16	Indicates the ending location of the loop in the program memory.
Loop Count	LC	16	Indicates the number of remaining iterations in a loop.

Table 3.2: Instruction Fetch Related Register.

Table 3.2 provides a list of instruction fetch related registers. The program counter is typical to a processor, indicating the current location of the instruction fetch. The width of the program counter depends on the size of the program memory. To avoid losing cycles during a loop, the BIST processor uses a zero delay loop. To perform zero-delay loops, three registers are used: LC, LS, LE. A loop is defined as the set of instructions which are between LS and LE. The LC keeps a count on the remaining iterations, and when PC reaches LE it is updated to LS. When the LC value is zero, the processor will continue to fetch the next instruction after LE.

3.1.4 Instruction Set

The registers listed in the previous section are dedicated and have specific instructions which can be used to access them as is shown in Table 3.3 and Table 3.4.

The read/write pointer registers can be updated with either zero or the max size of the memory. During the generation of memory transactions, these pointers have to either decrement or increment. Note that all the pointer register operations are the single cycle. This is because the inner loops of the BIST algorithms are dominated by these instructions. To support high flexibility the BIST processor supports the update of selected registers by immediate values. To perform move immediate takes multiple cycles. The reasoning behind this is that the instruction width is 4-bit and the immediate data can be bigger. Table 3.5 shows a set of pre-defined patterns are supported by the processor. These patterns are used for performing read/write to the memories under test. The pattern operation is of 4 cycles. As with a processor a jump instruction is supported. This is important to perform jumps to sub-routine. To keep the processor hardware cost low, the jump is an instruction which updates the PC value with an immediate value. The immediate value requires multiple cycles depending on the address range of the instruction ROM.

Instruction	Opcode	Description	Syntax
Single Cycle			
Read Increment	0000	Generate a read to memory and auto increment the rptr.	rd++
Read Decrement	0001	Generate a read to memory and auto decrement the rptr	rd- -
Read	0010	Generate a read to memory without update rptr	rd
Complementary Read Increment	0011	Generate a complementary read to memory and auto increment the rptr.	rdc++
Complementary Read Decrement	0100	Generate a complementary read to memory and auto decrement the rptr.	rdc- -
Complementary Read	0101	Generate a complementary read to memory without update rptr.	rdc
Write Increment	0110	Generate a write to memory and auto increment the wptr.	wr++
Write Decrement	0111	Generate a write to memory and auto decrement the wptr.	wr- -
Write	1000	Generate a write to memory without update wptr.	wr
Complementary Write Increment	1001	Generate a complementary write to memory and auto increment the wptr.	wrc++
Complementary Write Decrement	1010	Generate a complementary write to memory and auto decrement the wptr.	wrc- -
Complementary Write	1011	Generate a complementary write to memory without update wptr.	wrc
Move To Multi Cycle	1100	Move to multi cycle instruction following with immediate data.	mv
Branch Return	1101	Move back to the program count where branch start.	brret

Table 3.3: Single cycle instruction set for BIST processor.

Instruction	Opcode	Description	Syntax
Multi Cycle:			
Read pointer	0000	Point out the read location from a memory.	rptr
Write Pointer	0001	Point out the read location from a memory.	wptr
Read Pattern	0010	Performing read to the memories under test.	rp
Write Pattern	0011	Performing read to the memories under test.	wp
Max Size	0100	Move to maximum value of memory under test.	mms
Error Limitation Control	0101	Set the limitation of expected error numbers.	cr
End Of Program	0110	Update eop register.	eop
Loop Start	0111	Update loop start register.	ls
Loop End	1000	Update loop end register.	le
Loop Count	1001	Update loop count register.	lc
Read Pattern Immediate	1010	Performing read to the memories under test immediately.	rpi
Write Pattern Immediate	1011	Performing write to the memories under test immediately.	wpi
Jump	1100	Jump to the PC provided by immediate vaule.	branch

Table 3.4: Multi cycle instruction set for BIST processor.

Pattern	Opcode	Value
all_zero	0000	All zeros
all_one	0001	All ones
max_num	0010	Max size register value
al_01	0011	Alternate zero and one
al_10	0100	Alternate one and zero
al_b_01	0101	Alternate byte zero and one
al_b_10	0110	Alternate byte one and zero

Table 3.5: Pre-defined Pattern to load read and write pattern register.

3.2 Hardcoded BIST

3.2.1 Hardcoded BIST Architecture

The hardcoded BIST has eight pre-defined March algorithms for choose and can store 10 bits error information.

Figure 3.2 shows the top-level block diagram of the hardcoded BIST processor. This processor has a SIB decoder to decode data from a standard interface (i.e. SPI, JTAG or UART), which will be introduced in the next chapter. Then it follows with an FSM multiplexer and eight different March algorithms FSM: MATS, MATS++, March A, March B, March C, March LR, March X, March Y. The user can choose one of the algorithms by sending instructions through an interface. SRAM drive is the one which connected to the memory under test. In this design, it is possible to communicate with multiple same size memories since there is a pin reserved for choosing memory. Read data checker is a comparator which used to compare read data with expected data, error detected by the comparator will be stored in error logger, if the numbers of errors exceed default limitation, FSM controller will stop the algorithms immediately.

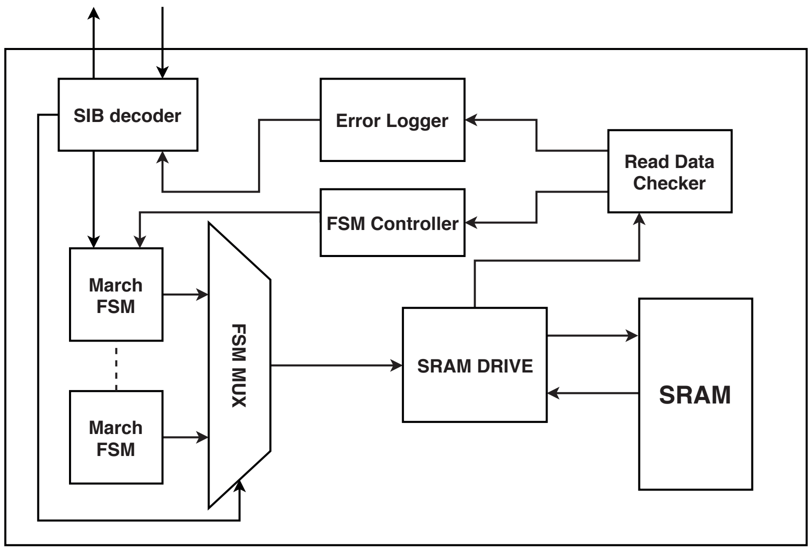

Figure 3.2: Harcoded BIST Top Level Block Diagram.

3.2.2 HDL Implementation

Figure 3.3 shows input instruction(32 bits) for control the BIST processor. The MSB is use to active BIST processor, the following 3 bits indicate which algorithm will be applied, the rest bits is reversed for memory choose and other function.

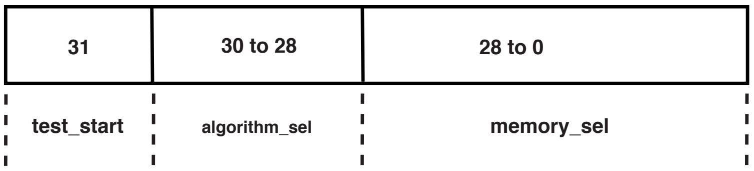

Figure 3.3: Input Instruction of BIST.

In order to cover all kinds of memory faults more comprehensively, this hard-coded BIST inset 8 March algorithms. Figure 3.4 shows the state diagram of March C- algorithm as an example to explain FSM block. *WR_ DATA_ NUM* is a generic signal which indicates a specific address location. March C- have 6 steps:

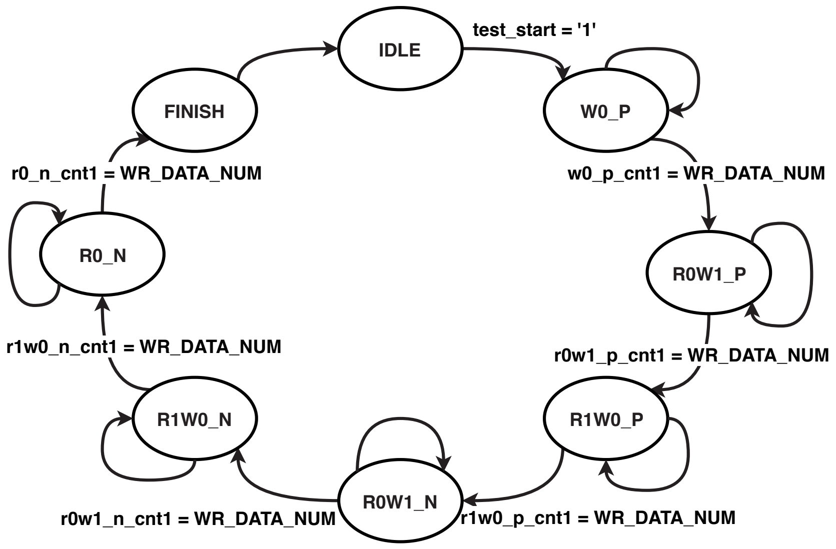

Figure 3.4: State Diagram of March C- algorithm.

1. $\updownarrow$*(w0)*: Write 0s in any order.
2. $\uparrow$*(r0,w1)*: Read (excepted value is 0) from lowest address then write a 1 at this address and move up to next address and repeat until highest address location.
3. $\uparrow$*(r1,w0)*: Read (excepted value is 1) from lowest address then write a 0 at this address and move up to next address and repeat until highest address location.
4. $\downarrow$*(r0,w1)*: Read (excepted value is 0) from highest address then write a 1 at this

address and move down to next address and repeat until lowest address location.
5. $\downarrow$*(r1,w0)*: Read (excepted value is 1) from highest address then write a 0 at this address and move down to next address and repeat until lowest address location.
6. $\updownarrow$*(r0)*: Read (excepted value is 0) from address in any order.

The read data checker module in this design can store a 10 bits data of error information, it can indicate the address of errors and how many bits have faults in this address. It can be expended to 16 bits to meet other requirements. The error logger block is a RAM which can store 20 error data if the error exceeds 20, then the controller will stop the process immediately.

Chapter 4

Reconfigurable Access Network and UART Interface

BIST can be considered as an on-chip testing instrument. Due to this reason, a sufficient and scalable testing interface is very essential. In this chapter, some commonly used interface will be first introduced and discussed. Then we will elaborate on IEEE Std 1687 (IJTAG) and related concepts such as SIB and description language. In the end, the hardware architecture used in this design will be explained.

4.1 Existing Methods For DFT

4.1.1 SPI Interface

SPI is a standard serial four-wire synchronous bus commonly used for sending data between a microcontroller, sensors, shift registers, SRAM and more. The SPI Interface was developed by Motorola to provide full-duplex synchronous serial communication between master and slave devices. It uses separate clock and data lines, along with a select signal to choose which slave you want to communicate with [13].

Figure 4.1 shows a typical 4 wire SPI configuration with multiple slaves. The device which generates the clock signal is called the master. SPI works in asynchronous mode, which means data transmitted between the master and the slave is synchronized to the clock generated by the master. *MOSI* and *MISO* are the data lines which are responsible for transmitting data from the master to slaves and back. *SS* signal is used to select slaves. SPI have many advantages like complete flexibility for the bits transferred, i.e. not limited to 8 bit, sample hardware connection, high speed. But one drawback of SPI is that each slave needs an individual select signal from the master. For hardware constraints, it is expensive for a dedicated testing port with at least 4 pins.

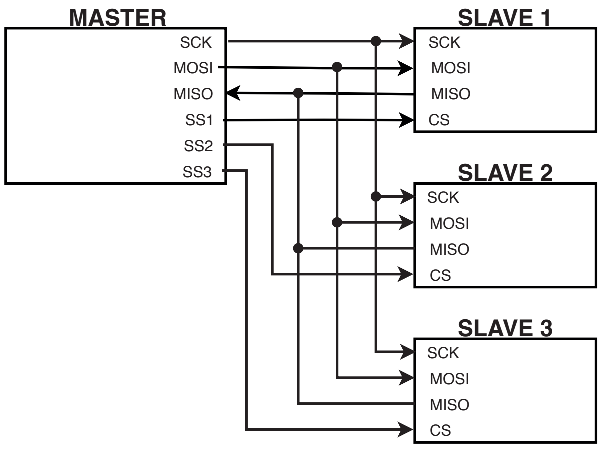

Figure 4.1: SPI with multiple slaves.

4.1.2 UART Interface

UART is an asynchronous serial communication protocol for transmit data. Figure 4.2 shows a trasmission of UART.

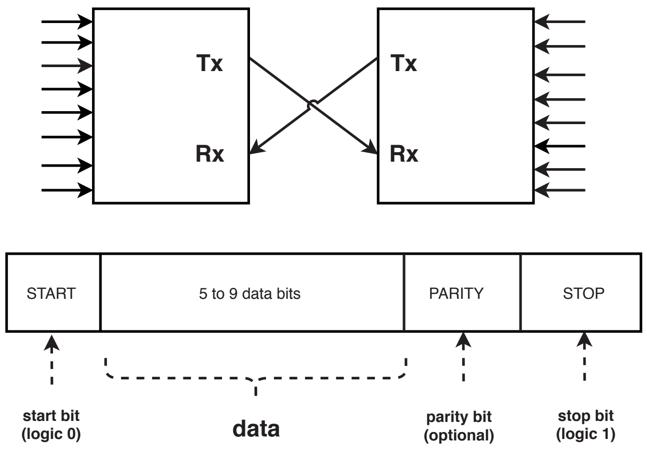

Figure 4.2: UART Transmission.

Data will be transferred into parallel form then transmits into UART. After the UART gets the parallel data, it adds a start bit, a stop bit and a parity bit, creating the data packet then send out the data bit by bit at the *Tx* pin. The receiving UART reads the data packet bit by bit at its Rx pin. The receiving UART then converts the data back into parallel form and removes the start bit, parity bit, and stop bits. Finally, the receiving UART transfers the data packet in parallel to the data bus on the receiving end. The advantage of UART is that it only needs 2 signal to transmits data.

4.1.3 IEEE Std 1149.1 JTAG

JTAG is the name of the group that defined the IEEE 1149.1 standard. This standard defines the TAP controller logic used in processors with JTAG interface which is commonly used as an interface for debugging, testing and programming. The hardware interface to the JTAG port usually consists of four main pins: *TCK* test clock, *TMS* test mode select, *TDI* input, *TDO* output.

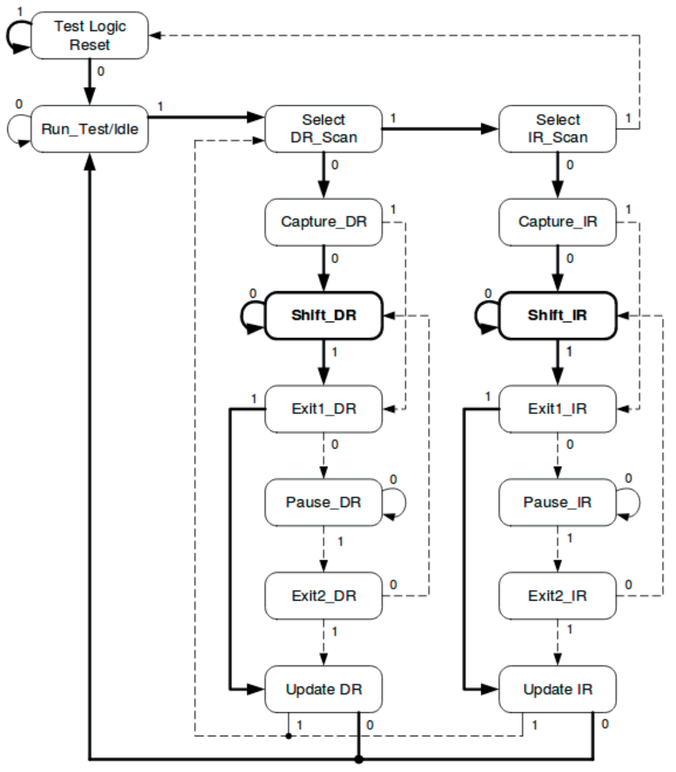

Figure 4.3: TAP State Machine [14].

Figure 4.3 shows a 16-state finite state machine TAP controller which responds to changes at the *TMS* and *TCK* signals of the TAP and controls the sequence of operations. It also controls the scanning of data into the various registers of the JTAG architecture. Two state transition paths are used to capture or update data by scanning through the instruction register (IR) or a data register (DR). All state transitions of the TAP controller shall occur based on the value of *TMS* at the time of a rising edge of *TCK*. Actions of the test logic shall occur on either the rising or the falling edge of *TCK* in each controller state. JTAG has many advantages, but its lacks flexibility of hardware and absence of a language to program an instrument independently of its position or configuration [15].

4.2 IEEE Std P1687 IJTAG And Related Concept

IEEE P1687 (IJTAG) standard defines an embedded Instrumentation to the JTAG with efficient, flexible and unified standardized instrument interface and variable scan chain path. The scan path changes in the scan chain insertion node-of SIB, making the system have flexible and variable characteristics. Access for embedded devices based on the access operations needs to choose the path of the scan chain to remove unnecessary scan chain, thereby reducing the length of the scan chain shift operation in the testing process to save the testing clock and improve efficient access [16].

Due to these advantages, IJTAG is a good choice for test systems that need to be built by multiple BISTs. In this project, we try to connect multiple hardcoded BISTs with an IJTAG based dynamic re-configurable network with UART interface which proposed in Gani and Prathamesh's thesis [17].

4.2.1 Segment Insertion Bit

A new class shift register device, the SIB, is defined in the IEEE P1687 standard. The SIB is a key part of constructing the IEEE P1687 scan chain path. The SIB is equivalent to a shift register with status bits. In addition to the ScanIn (*TDI*) and ScanOut (*TDO*) ports used to connect to the main scan chain, the SIB also has one for connecting to the next layer (which can be an embedded instrument or Other SIB) Hierarchical Interface Interface (HIP). HIP has a total of 3 ports: HIP-ToScanIn (*WSIO*), HIP-FromScanOut (*WSOI*) and HIP-ToSel (*Seli*). The SIB has two states like a switch circuit when it is selected. When it is on (as shown in Figure 4.4b), the SIB connects ScanIn (*TDI*) with HIP-ToScanIn (*WSIO*), ScanOut (*TDO*) Connect to HIP-FromScanOut (*WSOI*) to connect the device or network connected to the HIP to the scan chain. When the SIB is off (e.g. Figure 4.4c shows that ScanIn (*TDI*) is directly connected to ScanOut (*TDO*), skipping the device connected to the HIP. By using the switching function of multiple SIBs, the shift registers of many embedded instruments in the chip can be connected to the network or shielded, so that SIB can be added to the key nodes in the scan chain, and the scan chain can be completed by the serial scan chain. Flexible configuration and change [18].

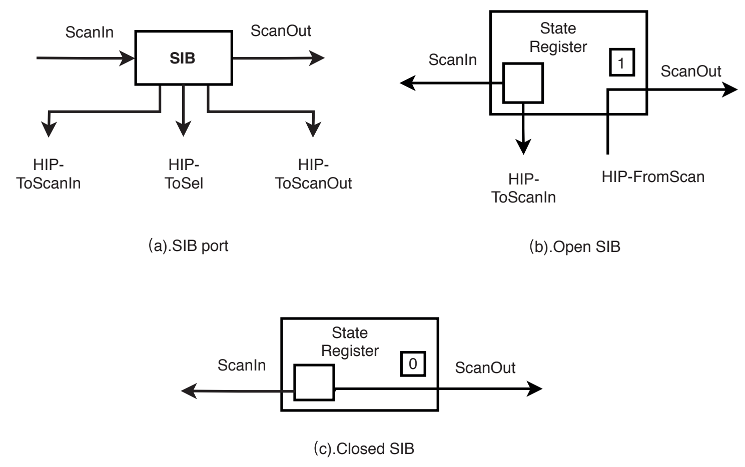

Figure 4.4: Simplified view of the SIB component [19]

4.2.2 Description Languages

The IEEE P1687 standard defines two new description languages, one for describing the architecture and the other for describing processes and operations (test vectors).

- ***1. Instrument connection language ICL:***
  The Instrument Connectivity Language (ICL) describes the various scan paths that may exist on the chip. It provides a mapping for the location of the SIBs in the IEEE P1687 network. Re-targeting tool can utilize the architecture described by the ICL mapping to activate the networks and instruments in the scan path. By activating any of these paths, a specific set of test vectors (e.g. data length and position) can be applied to some instruments.

- ***2. Procedural Description Language PDL:***
  Procedural Description Language (PDL), which represents the test vector or operational process applied to an IEEE P1687 embedded instrument. These languages simplify the use and portability of embedded instruments. Once the scan path of the IEEE P1687 network is determined, the test vectors defined by the PDL language can be transferred to one or more embedded instruments on the activated segment of the scan path [19].

With these two languages, the user can program the instrument in a network, but they do not generate the input vectors that need to be shifted into the network to configure the instruments. BIST need instructions to select algorithm and memory, so a tool which can generate input vector need to be created. This tool will be introduced in the following section.

4.3 Main Challenge

There are many different IEEE Std 1687 network architectures such as a flat network (see Figure 4.5), hierarchical network (see Figure 4.6), multiple networks and daisy-chained network [20]. Since we try to integrate multiple BISTs in the design, a simple and generic network which can easily add or remove BISTs from the scan chain should be discussed.

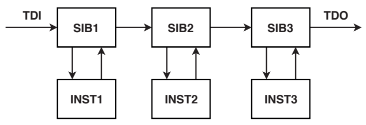

Figure 4.5: Flat Architecture

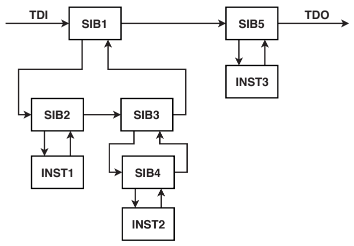

Figure 4.6: Hierarchical Architecture

Another problem is how to read error information from the BIST. UART is slow, and the error information may miss before the SIB receives the capture command from the controller. Also, BISTs may detect many errors from memory, so changing the controller to do multiple captures and getting the error information at the correct time would be a challenge.

4.4 Hardware architecture

In this design, each BIST can connect multiple memory through the memory wrapper. To save area, only 3 hardcoded BISTs are integrated into the network. However, due to the flexibility of the IEEE Std 1687 network, BISTs can be added at any time to meet new test requirements. Programmable BIST is flexible but it will not integrate into the network because of the area overhead of programmable BIST is several times of hardcoded BIST.

Figure 4.7 shows a flat network architecture of SIB network. As we see, each BIST connects with two SIBs. According to the selected BIST, one of SIB1, SIB3, SIB5 will be activated and sending instructions to BIST through *sib_ in* signal. When a read error command is sent, one of SIB2, SIB4, SIB6 will be activated and capturing the error information out from the BIST through *error_ out* signal. The hardware overhead is minimal with a flat network, but since SIBs are always on the scan path, it may lead to a higher time overhead.

Figure 4.8 shows a hierarchical architecture of SIB network. This idea is mainly achieved by separating error logger block from the BIST and connect it

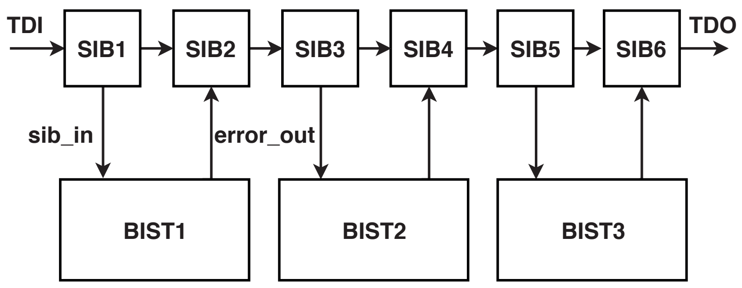

Figure 4.7: Flat Network With BISTs Architecture

with a SIB. In this architecture, when the user needs to read the error information from the error logger when the BIST is finished executing the program, the corresponding shift-registers and dedicated SIBs of BIST will not be activated. This hierarchical architecture approach will reduce the OAT by excluding the SIBs themselves from the scan chain. But it will lead to a high area overhead since the extra signal will be added from BIST to error logger, and also the error logger size will be quiet big to store all the error information of BISTs.

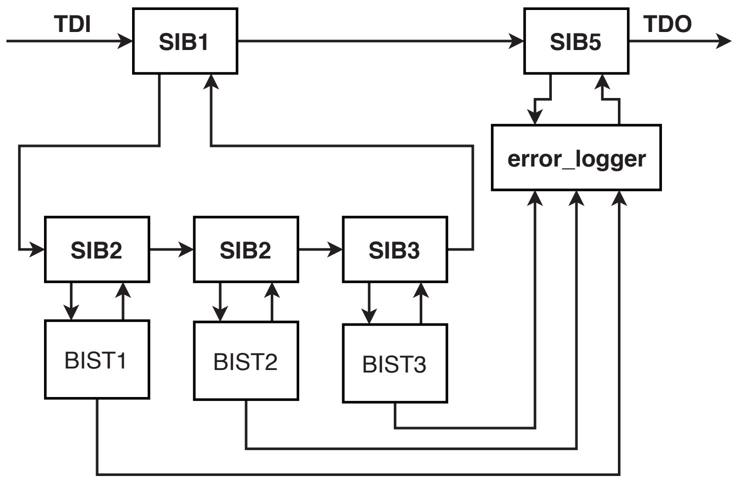

Figure 4.8: Hierarchical Network With BISTs Architecture

The flat network architecture has been chosen for application because the flat network has low area overhead. BIST is an on-chip testing module, so area cost is more important here. The complete hardware architecture is shown in Figure 4.9. The TAP controller is replaced by a UART protocol in this project [17], the master controller block is responsible for control the network. There are two components that are called SIB Control Register (SCR) and Instrument Length Memory (ILM) in this controller block. When instructions transferred in, the SCR will store data that indicate which SIB is part of the active scan path and what operation is to be executed on it. The ILM holds information about the instrument data lengths of the instrument in the network and the address of each instrument. The values in the SCR and ILM are transmitted by the software re-targeting tool through the UART channel [17].

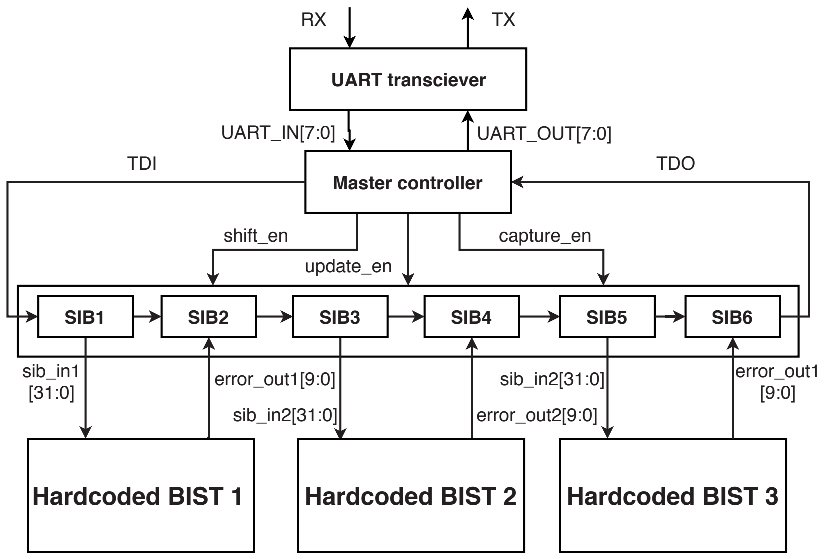

Figure 4.9: Hierarchical Network With BISTs Architecture

The first version of the whole system has one network which contains 6 SIB components. Although the network is flexible, it is hard to wrap. So here we separate the entire network block into 3 individual blocks which are modularizing the entire network with BIST and the memories under test. With this architecture, the user can easily add or decrease the number of BIST modules. The final Top-level diagram is shown in Figure 4.10.

4.5 HDL Implementation

There are two levels of PDL language. Level 0 PDL allows for static programming, which means no loops, branch condition and the interactive command will be executed. Level 1 PDL has all the features of a fully mature programming language [21]. Since BIST only need read and write operation, so here level 0 PDL is applied. Level 0 PDL has two types of command: *iWrite* and *iRead* are setup command, *iApply* is the action command. The setup commands only take

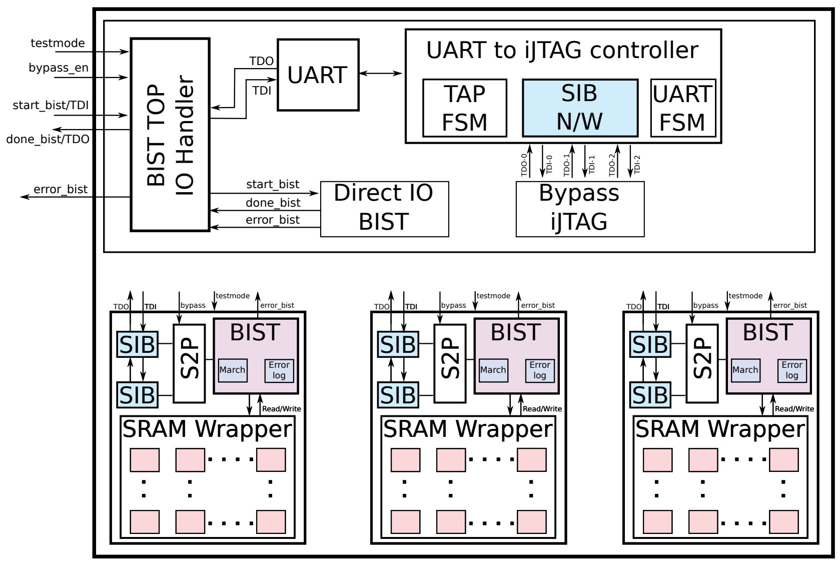

Figure 4.10: Top-level diagram of BIST

effect when a action command has been executed. The PDL and ICL describe the instruments in a network and commands to be executed on them. However, they do not generate the input vectors that need to be shifted into the network to configure the instruments. A instruction re-targeting tool has been designed that it based on SIB control register configuration in master controller [17].

Figure 4.11 shows a complete sequence of PDL instructions to activate hard-coded BIST1 and send error information out. Next, I will elaborate on the meaning of each byte.

- Byte 1-2: The two MSB bits of byte 1 indicates the *iWrite* command. The remaining part of byte 1 and byte 2 indicates the SIB address, which means that which SIB in the scan chain needs to be activated. In this case, the LSB of byte 2 is 0, so SIB0 is activated.
- Byte 3-4: The '10' in byte 3 is the *iApply* command, and the rest bits in byte 3 and byte 4 state how many bits of data will be sent to the network. As we mentioned in the previous chapter, hardcoded BIST has a 32 bits input to run the program, so it means 4 bytes of data need to transfer to BIST ('00000100' in binary notation).
- Byte 5-8: These 4 bytes are the instructions for the hardcoded BIST.
- Byte 9-10: The two MSB bits of byte 9 indicate the *iRead* command. The remaining part of byte 9 and byte 10 indicates the SIB address, which means

that which SIB in the scan chain needs to be activated. In this case, the LSB of byte 10 is 1, so SIB1 is activated.

- Byte 11-12: '10' in byte 11 indicates *iApply* command, here no data will be shifted into the network, so the rest of byte 11 and 12 is all 0.

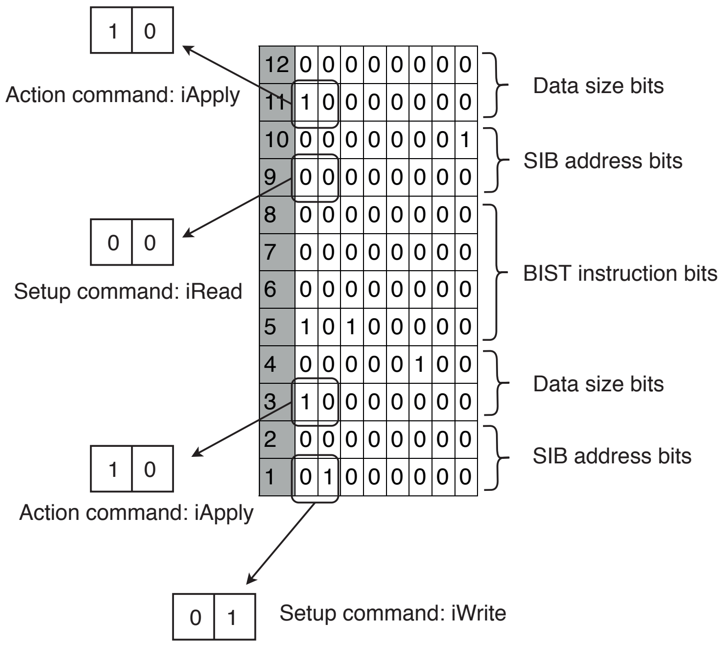

Figure 4.11: Bit-wise description of PDL instruction

This entire 12-byte instruction is a complete read and write operation to hardcoded BIST1. This seemingly complex instruction becomes very easy to implement with a PDL compiler written in Python (see listing 1).

After explaining the PDL re-targeting tool, the main FSM features in the master controller will be discussed (see Figure 4.12). There are 3 control states: SHIFT, UPDATE, and CAPTURE, which are used to control SIBs in the network. During the SHIFT state, the parallel data from UART will transfer into serial bit sequence and shift into the network, and then the FSM moves to the UPDATE state. In this state, it updates the required SIB and pulls the associated shift register into the valid scan path. When FSM moves to the CAPTURE state, the error information data which stored in the BIST will be shifted out if it following with a *iRead* command.

The FSM then returns to the IDLE state and waits for the data byte sent by the re-targeting tool. When the *iWrite* command is received, the following 4 bytes BIST instruction is moved to the network. When an *iRead* command has

Algorithm 1: PseudoCode of Re-targeting Tool

```
Result: Generate BIST instruction data
 1: function iWrite(bist = ' ')
 2:     write_cmd = (write command + Which BIST)
 3:     return write_cmd
 4: end function
 5:
 6: function iRead(bist = ' ')
 7:     write_cmd = (write command + Which BIST)
 8:     return write_cmd
 9: end function
10:
11: function iApply(mode)
12:     apply_cmd = (write or read mode)
13:     return write_cmd
14: end function
15:
16: function hardcoded_bist(start,algorithm_sel,memory_sel)
17:     algorithm = {"MARCH_A":'000',
18:                  "MARCH_B":'001',
19:                  "MARCH_C":'010',
20:                  "MARCH_LR":'011',
21:                  "MARCH_MATS":'100',
22:                  "MARCH_MATS+":'101',
23:                  "MARCH_X":'110',
24:                  "MARCH_Y":'111'}
25:     hd_instruction = (start+algorithm_sel+memory_sel)
26:     return hd_instruction
27: end function
28:
29: iWrite(bist = 1)
30: iApply('w')
31: hardcoded_bist(start=1, algorithm_sel= "MARCH_C",
    memory_sel=1)
32: iRead(bist = 1)
33: iApply('r')
```

been applied, a series of dummy bits are shifted into the scan chain to shift the error information data out. When the BIST instruction arrives, a bit sequence is created to move data or dummy bits into the correct SIBs.

When it comes to data phase, first the FSM will move to DATA_SHIFT state. In DATA_SHIFT state, the BIST instruction or dummy bits will be shifted in the network. In order to shift the data a HOLD state is added after DATA_SHIFT state. Then the FSM transitions to the DATA_UPDATE state and loads data from the shift register into the hardcoded BIST in parallel.

The resting state is for capturing error information out from the network. After DATA_UPDATE state the FSM will move to COMMAND_CHECK state. In this state, the FSM will check if a *iRead* command has been received. If this command is not received, the FSM will move to the RESET_STATE state and reset all counters and states of other FSM. If the *iRead* command is received, the FSM will come to the POLL_ERROR state. According to the preset numbers, the FSM goes to CAPTURE state and repeat capture operation until reach the predefined capture error numbers. After all, it will back to RESET_STATE state and wait for another iteration.

As mentioned in the previous section, the essence of BIST is the constant reading and writing of memory. When BIST detects an error, the error message is quickly stored in the error logger module. For different sizes of memory and different algorithms applied, the time to perform a test is also different. However, the UART is relatively slow, and the UART speed is limited by the baud rate. The common baud rates are 2400, 4800, 9600, 19200, and 115200 BPS. However, even with the highest baud rate, the UART still misses some useful data when the measured memory size is small, or the measured address range is short. The main idea to solve this problem is to display the error information after the BIST is completed, and because the UART read rate is relatively slow, each error information will display for 2ms, and when the FSM enters the POLL_ERROR state and the error count smaller than preset value, it will move to the CAPTURE state and repeat to perform a capture operation every 2ms.

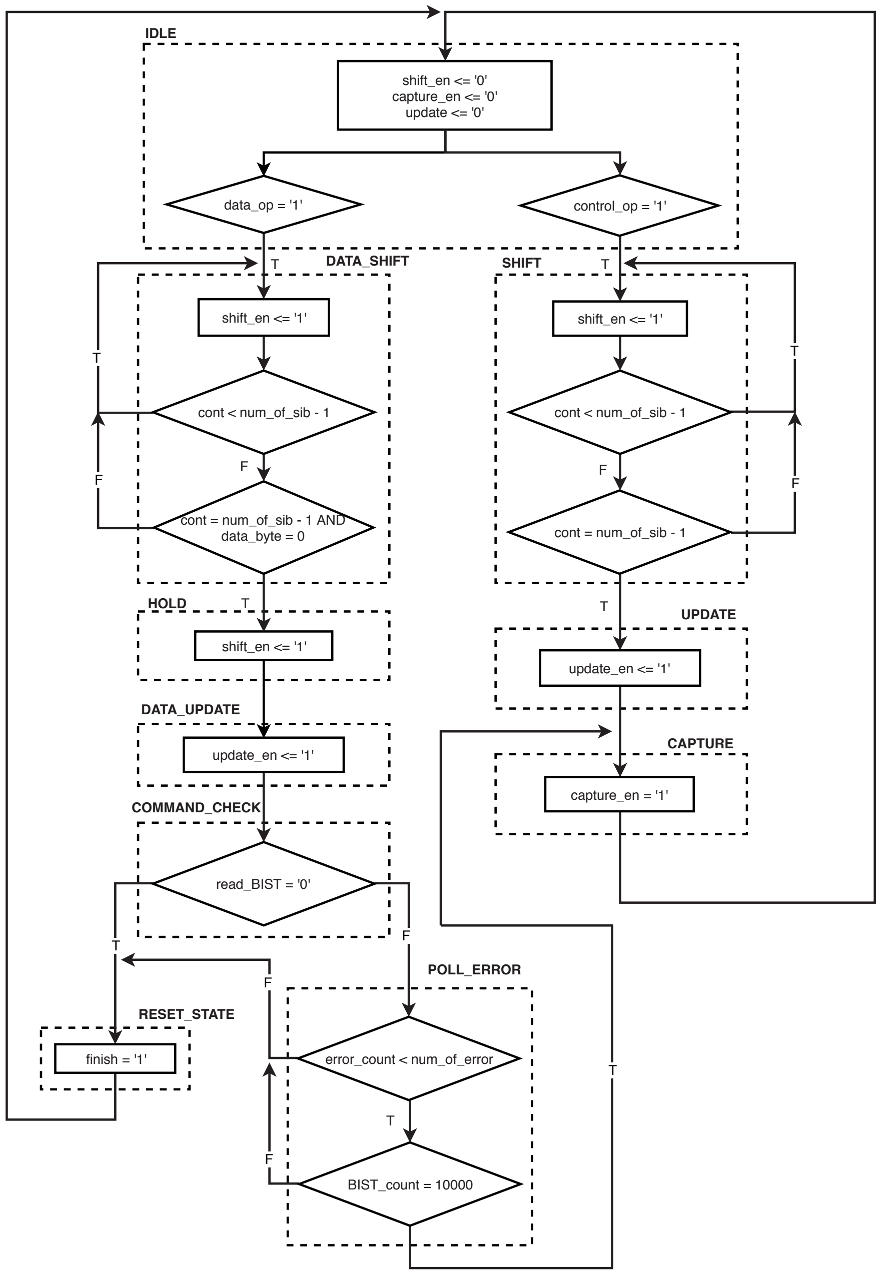

Figure 4.12: ASMD Diagram Of Main FSM Of Master Controller

Chapter 5

Verification And Results

In this chapter, the verification of the hardcoded BIST, the programmable BIST and the network will be discussed first. After that, some synthesis report of the system will be listed.

5.1 Verification of BIST

5.1.1 Hardcoded BIST

The verification is based on Xenergic memory model. Here a 2048*32 SRAM model is been chosen for testing. The address width of this memory model is 11 bits and the data width is 32 bits. March C- algorithm has been applied in this test since it can cover SAF fault. Some SAF errors which been added before testing are:

- Stuck at 1: Address location:3, Bit location:2
- Stuck at 1: Address location:3, Bit location:4
- Stuck at 0: Address location:1, Bit location:1

Figure 5.1 shows the simulation waveform result of the detection of stuck at 1 error. From the picture, we can observe that the *c_ s* signal which indicates the FSM state is 2. As we mentioned in chapter 2, the second state of March C- is $\uparrow$*(r0, w1)*, so now the BIST is doing an incremental read 0 and write 1 operation. The *wr_ data_ i* signal and *wr_ data_ en_ i* signal are responsible for write operation. Here it writes 0xffffffff (0b11111111111111111111111111111111 in binary notation) to every address location. When it comes to reading phase, we can see *rd_ data_ i* is always 0 except when at address location 3. 0x00000014 (0b10100 in binary notation) indicates in bit 2 and bit 4 is 1 in address location 3 which match the fault we preset in memory. After read this value, the *error_ cnt* signal increase to 1 and the *error_ inf* signal storing the error information. The read data checker module in this design can store a 10 bits data of error information which indicates the address of errors (4 down to 0) and how many bits have a fault in this address (9 down to 5). So 0x042 (0b0001000010 in binary notation) is the correct error information data which error logger will store.

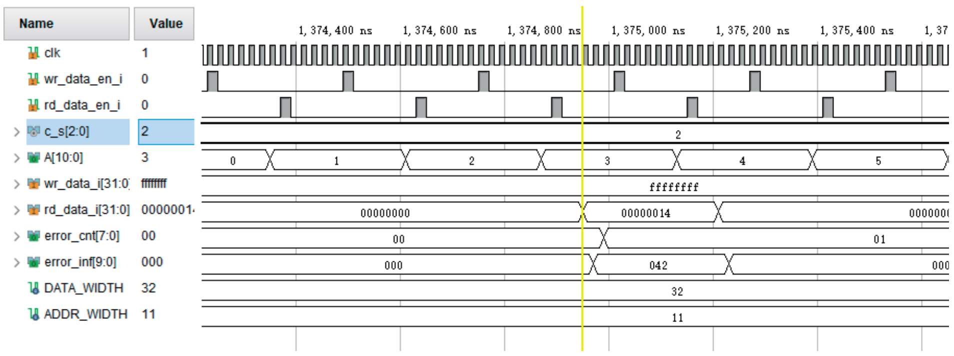

Figure 5.1: Error Detection Waveform of Hardcoded BIST 1.

Figure 5.2 shows the detection waveform of stuck at 0 fault. Here the *c_ s* signal is 3 which means the FSM is in ↑*(r1, w0)* step and the read data value should be all 1. From the picture we can see when reading at address 1, the *rd_ data_ i* is 0xfffffffd (0b11111111111111111111111111111101 in binary notation) which means the first bit of address 1 is stuck at 0. This data confirms the fault we set earlier. Also the *error_ cnt* signal increase to 2 after detect this error and the *error_ inf* signal storing 0x021 (0b0000100001 in binary notation) into error logger block.

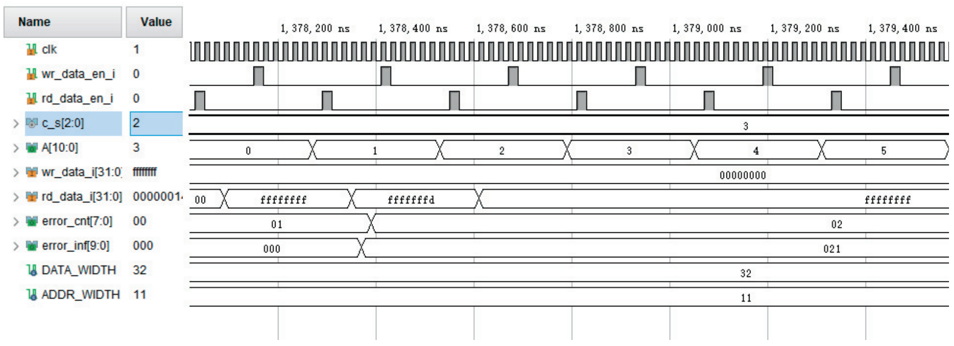

Figure 5.2: Error Detection Waveform of Hardcoded BIST 2.

5.1.2 Programmable BIST

The programmable BIST has been connected to the same memory model as in hardcoded one. The main idea of programmable error detection is that the error handler block will compare the reading data with the read pattern data, if there is a mismatch then it means there is an error. The error information which programmable BIST stored is the same as hardcoded BIST. From figure 5.3 we can see the *read_ pattern* signal is the pattern generate by register bank, and the *tb_ mem_ datain* is the reading data. It shows a data mismatch at address location 3. After capturing this mismatch, the *error_ inf* signal capture this error and store it in error logger.

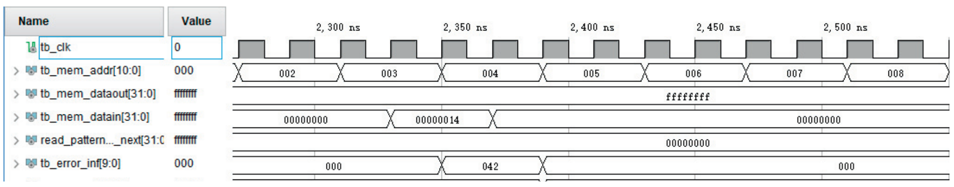

Figure 5.3: Error Detection Waveform 1.

In this test, we use the assembly2binary tool which writes in Python to generate the instructions (a March C- algorithm) to instruction rom first. Since the same memory model and the same algorithm have been applied, so the error information which programmable BIST stored should be the same as shown in hardcoded one. (see figure 5.3 and figure 5.4)

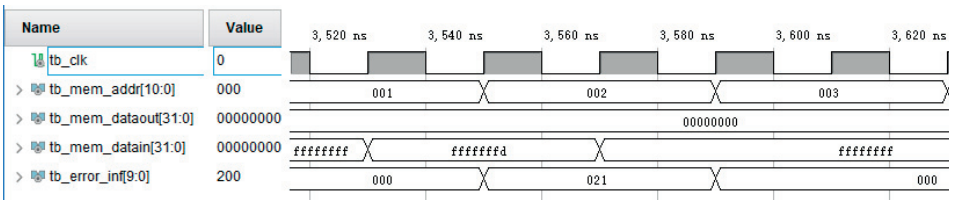

Figure 5.4: Error Detection Waveform 2.

5.2 UART and Network Functionality Verification

In this section, the functionality of the UART protocol and IJTAG based network will be verified. First we use the re-targeting tool to send an instruction which active hardcoded BIST 2 with a March C- algorithm and then read the error information out.

From figure 5.5 we can see that the entire system is running correctly. The *rx* signal is the input interface of the UART. In the figure, we can see that, as mentioned before, if the measured memory is small or the selected measurement address interval is small, the BIST is completed before the UART instruction is completely delivered. This may result in the system not being able to capture some errors. However, since the FSM has added the corresponding state and the error logger module, the error information will be displayed every 2ms after SIB captures the test end signal, and the controller block enters the POLL_ERROR state to capture the error message at the synchronous rate.

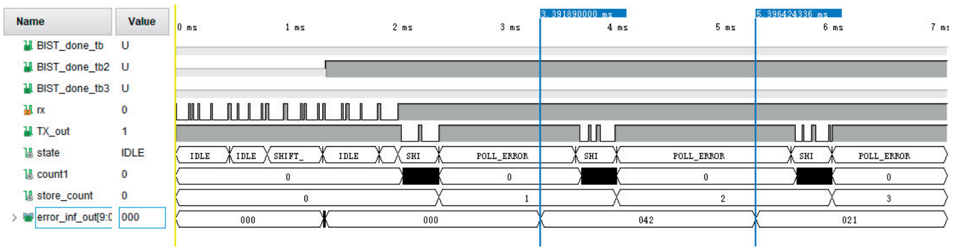

Figure 5.5: UART Transceiver waveform.

Tx_out is the UART output signal which is '1' when idle. Transmission starts with a '0' and is followed by 8 bits data, then end up with an optional parity and a stop bit, which is '1'. Since the error information is a 10-bit data, and the UART can only transmit 8 bits of data at a time, the error information is split into two part and shift out of the system. Through the *count1* signal, we can more intuitively verify the correctness of the data transmitted by the UART. The *error_inf_out* signal shows that the current error information is 0x042, so the data read by *Tx_out* should be the same as 0x042. The first byte of *Tx_out* is 0b00001000, and the second byte is 0b00000001. Since the data is transmitted by the UART from the LSB, the data read should be the first byte plus the first two bits of the second byte. Bit, which is 0b0001000010. This match the data stores in the error logger.(see figure 5.6)

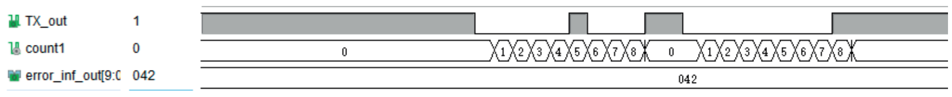

Figure 5.6: Error Detection Waveform 2.

Figure 5.7 shows another error information. After this verification, it proves this re-configurable access network with a UART interface can be applied to the BIST.

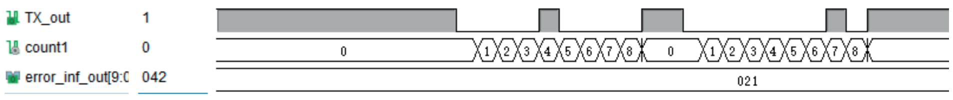

Figure 5.7: Error Detection Waveform 2.

5.3 Synthesis Results

In this section, some synthesis results will be listed (see table 5.1). It shows the gate area of the whole system which contains master_control block, 3 hard-coded bist, 3 networks and 3 memory wrapper. The memory model uses in synthesis is a Xenergic 2048*32 SRAM which has 32-bit data width and 11-bit address width.

Table 5.1: Syhtnesis area report.

	Gate area
bist_top	195755.47
master_control and network	17373.36
hardcoded bist	10069.96
sram_2048*32	48519.48

Figure 5.8 and figure 5.9 shows gate area distribution of two BIST implementations. The total gate area of hardcoded BIST is larger than programmable BIST, but hardcoded BIST contains 8 algorithms FSM. The programmable BIST only contains a March C- algorithm. If only choose March C- FSM, the total gate area of the hardcoded BIST is 3862.66 and the total gate area of programmable BIST is 8730.32.

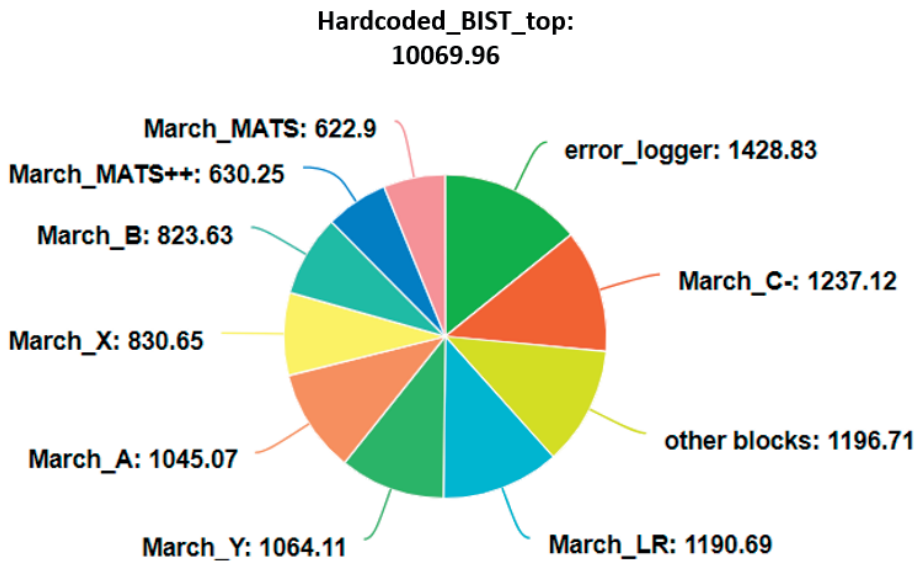

Figure 5.8: Gate area distribution of hardcoded BIST with 8 algorithms

From the result, we can see that if we run the March C- algorithm, the size of programmable BIST is 2.3 times larger than hardcoded BIST. This is due to large area of the instruction rom block, instruction decode and register bank block of the programmable BIST.

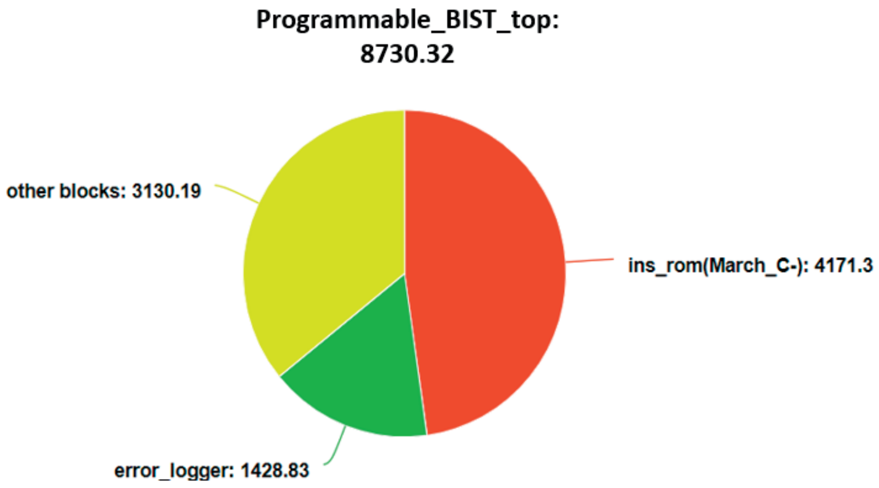

Figure 5.9: Gate area distribution of programmable BIST with 1 algorithm

Overall, hardcoded BIST can meet the requirement of testing a large number of memories and give good fault coverage. The programmable is mainly for testing specific type of memory faults, which user can create their own March based algorithm.

Chapter 6

Conclusions

Multiple BISTs and a reconfigurable network with a UART interface as a BIST system have been successfully built and verified. After the literature review of SRAM fault types and BIST-related theories, I elaborated on the way to design and implement two different BISTs, which includes BIST architectures and HDL implementations. The drawbacks of existing test interfaces are analyzed and compared with the IEEE Std.1687 IJTAG interface. Additionally, I introduced the advantages of this network. In the final implementation phase, major challenges are listed and analyzed. The pros and cons of the different network structures were discussed. Afterwards, I modified the original network and connected it with multiple BISTs. Moreover, an ASMD diagram was attached to explain how the network works. In the end, I verified both BISTs and network and provided gate area synthesis results of the BIST system.

Chapter 7

Further Work

In this project, only hardcoded BIST has been applied to the network because of the large area cost of programmable BIST and the large instruction sets. There is also a limitation on the amount of data bytes that the *iApply* command can send. One solution is to modify the PDL re-targeting tool which enables the connection of the programmable BIST to the network.

All embedded memory blocks, that are implemented as RAMs, support byte enable signals. These byte enable signals mask the input data so that only specific bytes or bits of data are written. The unwritten bytes or bits retain the previously written values. In some cases, faults happen in a byte enable signal. An algorithm which can detect the byte enable error and corresponding logic could be designed in the future. For example:

$$\{\updownarrow w0\}, \{r0, w1_{be1}\}, \{r1_{be1}, w1\}, \{r1, w0_{be2}\}, \{r0_{be2}, w1\}, \{r1, w0_{be3}\}, \{r0_{be3}\}$$

The above algorithm is a modified March C- algorithm which can detect the byte enable errors. In step 2, after reading 0 from all address locations, BIST will write 1 to a specific byte and then read 1 from that byte, and all the value in other bytes will also be checked. If the value in one of the other bytes is not 0, an error should be captured in error logger.

Bibliography

[1] R. Ohlendorf, “A Network Processor Architecture with Application-Optimized Reconfigurable Processing Paths (FlexPath NP),” Sep 2010.

[2] A. A. Chien and V. Karamcheti, *Moore’s Law: The First Ending and a New Beginning.* Sec 2013.

[3] R. Rajsuman., *System-On-A-Chip: Design and Test (Artech House Signal Processing Library).* Artech House, 2000.

[4] S. Luo., *Digital Integrated System Chip (SoC) design.* Beijing Hope Electronic Press, 2002.

[5] A. L. Crouch, *Design-for-test for Digital IC’s and Embedded Core Systems.* Prentice Hall PTR, 1999.

[6] L. Si’an, L. nian, S. Haibin, and Y. Xiaolang, “Principle and Implementation of Embedded Memory Built-in Self-test,” Feb 2004.

[7] J. Soetemans, “Method and system for testing a random access memory (RAM) device having an internal cache,” Apr 2009.

[8] K. Dalal and Rajni, “A Single Ended SRAM cell with reduced Average Power and Delay,” Sep 2019.

[9] J. Duarte, “SRAM Memories,” Dec 2014.

[10] B. BAI HONG FANG, “Embedded memory bist for systems-on-a-chip,” diploma thesis, Mcmaster University, Oct 2003.

[11] Suk and S. M. Reddy, “A March Test for Functional Faults in Semiconductor Random Access Memories,” Dec 1981.

[12] A. J. van de Goor., *Testing semiconductor memories: theory and practice.* J. Wiley & Sons, 1991, 2007.

[13] P. Dhaker, “Introduction to SPI Interface,” Sep 2018.

[14] S. Labs, “Programming Flash Through THE JTAG INTERFACE,”

[15] M. Portolan, S. Goyal, B. V. Treuren, C.-H. Chiang, T. Chakraborty, and T. B. Cook, “A New Language Approach for IJTAG,” Dec 2008.

[16] K. Posse, A. Crouch, J. Rearick, B. Eklow, M. Laisne, B. Bennetts, J. Doege, M. Ricchetti, and J. f Cote, “IEEE P1687: Toward Standardized Access of Embedded Instrumentation,” 2006.

[17] K. Gani and M. Prathamesh, “Reconfigurable instrument access network with a functional port interface,” diploma thesis, Lund University, May 2019.

[18] J. Rearick and A. Volz, “A Case Study of Using IEEE P1687 (IJTAG) for High-Speed Serial I/O Characterization and Testing,” Oct 2006.

[19] E. Larsson and F. G. Zadegan, “Accessing Embedded DfT Instruments with IEEE P1687,” 2012.

[20] F. G. Zadegan, E. Larsson, A. Jutman, A. Jutman, and R. Krenz-Baath, “Design, Verification, and Application of IEEE 1687,” Nov 2014.

[21] A. Richardson, “Using Test Access Standards Across The Product Lifecycle,” May 2016.

Printed by Tryckeriet i E-huset, Lund 2019

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2019-722
http://www.eit.lth.se