A Computer Aided Engineering System for Memory BIST

*Chauchin Su, Shih-Ching Hsiao, Hau-Zen Zhau, and Chung-Len Lee**

Dept. of Electrical Engr., National Central University
Chung-Li, Taiwan 320, R.O.C. ccsu@ee.ncu.edu.tw

Dept. of Electronic Engr., National Chiao-Tung University*
Hsin-Chu, Taiwan 300, R.O.C.

Abstract

Abstract - An integrated memory test system is presented. It includes a reconfigurable memory test module, a test algorithm editor, a memory fault simulator, and a test code generator. For a given memory organization, fault list, and test algorithm, the system automatically reports the fault coverage, generates control assembly codes, and produces circuit net list for test pattern generation. The system has been implemented in 9000 lines of C++ program based on the Microsoft Windows graphic user interface. It has been verified on different test algorithms and memory chips.

1. Introduction

For *System on Chip* (SoC), the integration of memory and logic on a chip is a significant challenge not only for design and manufacture but also for test as well. Due to its smaller feature size than logic circuit, embedded memories are more likely to be be affected by process imperfection. Unlike stand-alone memory chips, embedded memories have practical no control from the chip boundary. Hence, the test methodology is one of the major issues in SoC testing. With limited I/O access, *Built-in Self-Test* (BIST) is an effective and natural solution. In this paper, we would like to propose a BIST architecture and it *Computer-aided Engineering* (CAE) environment.

An SoC IC has many embedded memories of different sizes and organization. Here, we would like to focus on large memories. Large embedded memories are mostly DRAMs. Their characteristics include small feature size, large area percentage, and incompatible process technology. These characteristics make them more likely to be faulty. Hence, their test is critical for function verification. In addition, diagnosis is also crucial for yield improvement. Therefore, the memory BIST in an SoC environment must also provided with diagnosis capability.

In the rest of this paper, following topics will be discussed, (2) BIST architecture (3) CAE architecture (3) memory organization, (4) fault model, (5) Forth Engine, (6) march sequence generator, (7) memory fault simulator, (8) automatic code generator, and (9) hardware emulation experiments.

2. BIST Architecture

For diagnosis purposes the fault type and fault location are important information. In addition, it must provide test engineers a way to execute specialized test algorithm solely for the diagnosis. The conventional hard wired BIST architectures which have the dedicated test pattern generators for specific test algorithms will not meet the diagnosis requirement. Hence, in this paper a processor based BIST architecture is proposed to satisfy the diagnosis needs.

The proposed BIST architecture is shown in Figure 1. It is composed of (1) a simple processor core (MPU) for test flow control and external interface, (2) a programmable test pattern generators, and (3) a memory interface circuitry for the control signal mapping. The processor under consideration is a very simple Forth Engine. It can be replaced by any processor core in SoC. Hence, there is practical no overhead on this part. The programmable test pattern generator is counter based. It is responsible for generating the March algorithm. The interface circuitry is responsible for mapping the functional test sequence into suitable signals for that particular memory. In this architecture, only the interface circuitry is CUT dependent. MPU and Prog. TPG can be shared by different memories to minimize the overhead.

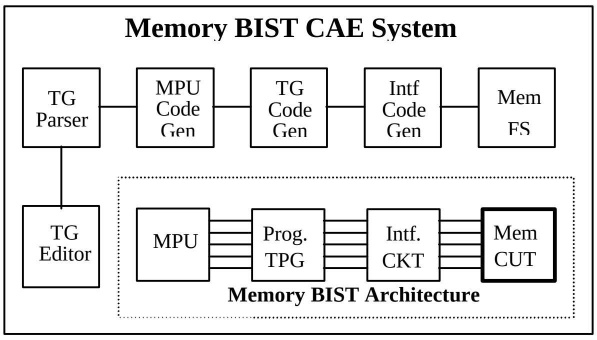


Figure 1 Memory BIST CAE System.

3. Memory BIST CAE System Architecture

The memory BIST CAE system archtiecture is also shown in Figure 1. It is composed of a test algorithm editor, a memory fault simulator, a test algorithm compiler, a Forth code generator, programmable TGP generator, and a interface circuitry net list generator. The whole system is accomplished in 9000 lines of C++ code. In TG Editor, memory organization and I/O specification and test algorithm are input. The system generates the codes and netlist for MPU, Prog. TPG, and Intf. Ckt automatically. It will also report the fault coverage for the given fault list for users to check the algorithm. This is especially useful for diagnosis because users are able to change algorithm easily to see if the target faults are covered or not.

4. Memory Organization

Figure 2 shows a functional model of a DRAM module [1,3]. It consists a command decoder (block A), an address buffer (block B), a memory cell array (block C), a data control circuit (block D), and a refresh logic (block E). Block C is word-oriented. It has 2 banks. Each bank has 2048 rows and 512 columns of 8-bit words.

5. Memory Fault Model

The faults being considered are mostly conventional

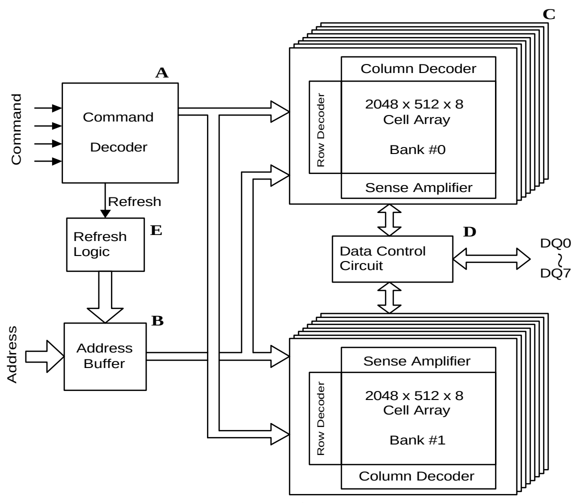


Figure 2: DRAM functional model

faults [3,4]. The notations being used are <S/F> for single cell faults and <S;F> for multiple cell faults. Here, S is the condition for sensitizing the fault and F describes the value of the faulty cell.

Single Cell Fault:

Stuck-at fault (SAF): SA0, SA1.

Transition fault (TF): <↑/0>, <↓/1>.

Read disturb fault (RDF) : <r0/↑>, <r1/↓>.

Multiple Cell Fault:

Inversion coupling fault (CFin): <↑;inv>, <↓;inv>.

Idempotent coupling fault (CFid):
<↑;0>, <↑;1>, <↓ ;0>, <↓;1>.

Bridging fault (BF): ABF, OBF.

State coupling fault (SCF):
<0;0/1>, <0;1/0>, <1;0/1>, and <1;1/0>.

Disturb fault (CFds): <r0;↑>, <r0;↓>, <r1;↑>, <r1;↓> ,<w0;↑>, <w0;↓>, <w1;↑>, <w1;↓>.

Address Decoder Fault: *Address decoder faults* include no access, no accessed, multiple access, and multiple accessed. Their combinations are shown in Figure 3.

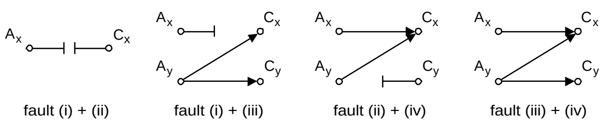


Figure 3: Combinations of address decoder faults.

6. Forth Engine

Forrth Engine [4,5] is a very simple stack based microprocessor. An 20-bit Forth Engine MPU-21 [4] can be implemented by only 7,000 transistors using 1.2mm CMOS processor. The block diagram of MPU-21 is shown in Figure 4. MuP21 uses an RISC-like instruction set with only 25 machine instructions. It has two stacks. Data Stack is 5-levels deep and Return Stack has 4. There are 31 primitive words created from 25 machine instructions. Pograms are built by creating sub-units, which are called words, rather than by writing down a series of statements.

One of the major advantages of Forth is that it can be extended easily. One can effectively create new words to

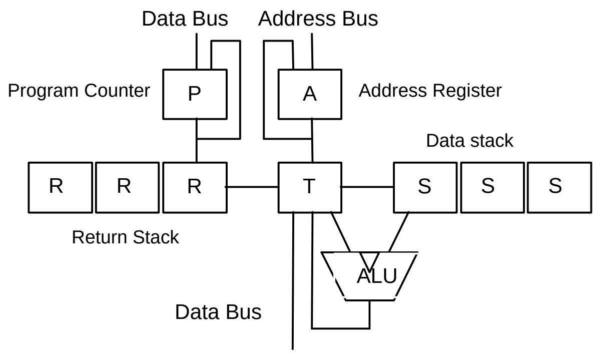


Figure 4. Forth Engine Architecture

One of the major advantages of Forth is that it can be extended easily. One can effectively create new words to suit the particular needs. For different algorithm, we can create different words to control the tester. For example, we are able to build march elements as words, such as ⇕(w0), ⇑(r0,w1), and ⇑(r1,w0). Then, we can compiles these words into a March C algorithm such as:

⇕(w0); ⇑(r0,w1); ⇑(r1,w0); ⇓(r0,w1); ⇓(r1,w0); ⇕(r0).

Other march algorithms can be built by similar elements.

7. Memory Test Editor and Parser

A graphic user interface is provided for users to enter the memory organization, memory fault list, and test algorithm. It then stores the information into separate files. After the test files are built, parser are called to analyze the syntax and build necessary data structure for Forth code, Programmable TPG, and Interface Circuitry generation. The tokens and grammars being used are as follows.

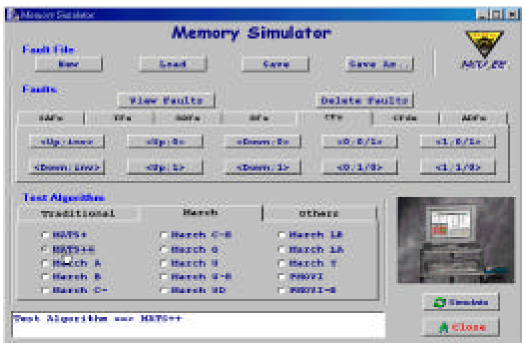
Tokens

Page Burst U D w r ( ) = , ; data \

Grammars

<page size> ::= Page = <data>;
<burst length> ::= Burst = <data>;
<element-list>::=<element>|<element-list>;<element>
<element>::= U (<operation-list>)| D(<operation-list>)
<operation-list>::=<operation>|<operation-list>
<operation>::= w<data>|r<data>
<data>::= 0 | 1 | 2 | 3 | …..| 7 | 8 | 9

Graphic user interface is used to guide users to edit the required information. The GUI of test algorithm and fault list editor is shown in Figure 5.



Test Algorithm use MATS++

Figure 5. Test algorithm and fault list editor.

8. Test Algorithm Compiler

In addition to 15 pre stored test algorithms, shown in Figure 5, for users to chose from, users are able to edit their own test algorithm. Due to its simple syntax and grammar structure, a one-pass compiler is sufficient for compiling the test algorithms. There are three steps in compilation process – scanning, parsing, and code generation.

Scanner

The task of the scanner is to recognize keywords, operators, and data backgrounds. It will ignore the comments after the symbol "\". For efficiency and simplicity of later usage, we assign each token a certain integer code.

Parser

After the token scan, test algorithms must be recognized as some construct described by the grammar. A parser performed in this process is called *syntactic analysis* or *parsing*. For example, Figure 6 shows a parse tree of statement U(r0, w1, r1).

Code Generator

We use the uniform notation [2,3] for memory tests as the syntax. A march test consists of a sequence of march elements, which are separated by semicolons ';'.

<march test> ::= <march element>{;<march element>} Each march element consists of a symbol to denote the addressing order and a sequence of operations, which are delimited by parentheses '( )' and separated by commas.

<march element> ::= <addr order> (<op> {,<op>})

<addr order> ::= ↑ | ↓ | ↕

<op> :: = r0 | r1 | w0 | w1

An example of well-known test algorithm, March B, will be written as follows.

↕(w0); ↑(r0, w1, r1, w0, r0, w1); ↑(r1, w0, w1); ↓(r1, w0, w1, w0); ↓(r0, w1, w0)

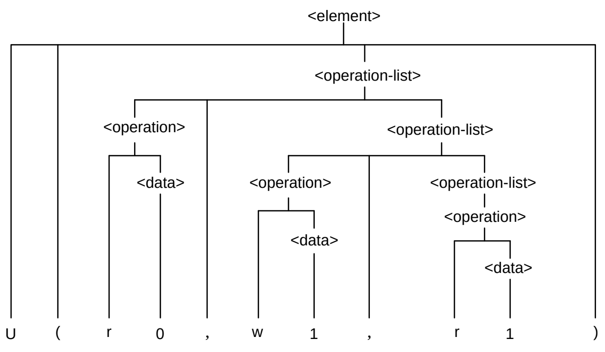


Figure 6: Parse tree of statement U(r0, w1, r1)

9. Memory Simulator

Memory Organization File

The memory organization files have to be stored in advance. The information includes the following items.

1. memory chip code,
2. the number of banks,
3. the row address bit length,
4. the column address bit length,
5. the word length,
6. the shrinking ratio

Fault Files

In a write or read operation, the fault list will be accessed to determine whether the faulty cells are activated. If so, proper actions will be taken according to the fault type. Hashing of 64 entries is used to speed up the fault search. For each fault, the recorded information is

1. the address of the aggressor,
2. the faulty aggressor bit,
3. the fault type,
4. the address of the victim cell,
5. the fault victim bit.

Since there can be more than one victim cells, a linked list is used to record the victim cells. The graphic representation is shown in Figure 7. When a fault is detected, we have to trace back the fault file and mark the fault. In the previous hash table, the searching keys of hash table are the address of aggressor cells. Here, we have to create another hash table using the victim cells as the searching keys to record the test result. This new hash table is similar to the previous one. However, the search key is reversed.

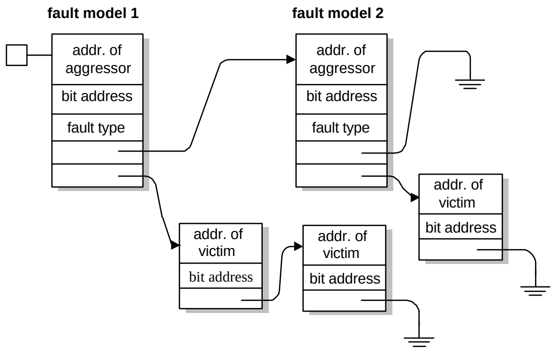


Figure 7. Fault list data structure.

10. Hardware Emulation Experiment

To verify the proposed memory BIST architecture and the associated CAE system, we use discrete components to built a BIST system. The test environment is shown in Figure 8. It is composed of a memory test board, a Forth Engine demo board, and a personal computer for the user interface and code generation. On the memory test board, there are multiple memory under test and a FPGA for the implementation of Programmable TPG and Interface Circuitry. Users are able to use PC for algorithm and technology file editing. After the compilation, the PC will transfer the test flow control codes to MPU-21 and down load the circuit netlist to the FPGA.

Synchronous DRAMs have more complicated operations as compared to other memories. The simplified state diagram

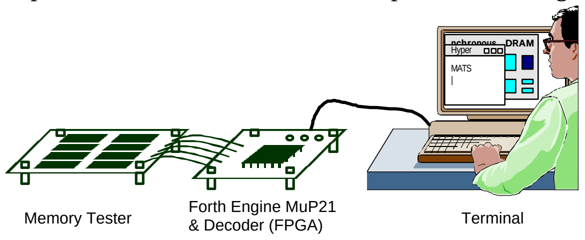


Figure 8 Memory BIST environment.

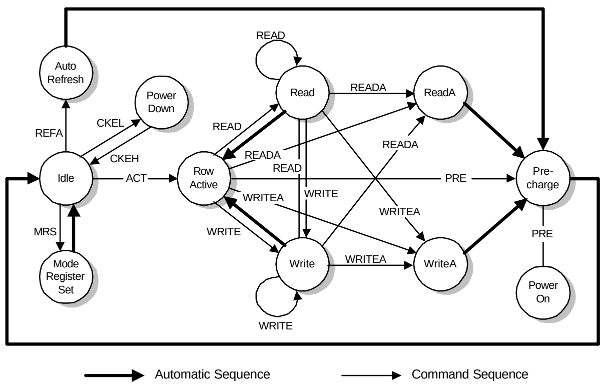


Figure 9. Simplified state diagram of SDRAM.

is shown in Figure 9. The most significant advantage of the proposed BIST architecture is that we are able to use the processor (MPU-21) to deal with the different state diagrams of different memory types. Hence, the hardware complexity of the subsequent circuits can be reduced significantly.

The block diagram of the BIST hardware structure is shown in Figure 10. The high level control commands are sourced from MPU21 board. Test patterns and interface protocals are generated from the FPGA (the lower part). The circuit diagram of the TPG and interface circuits is shwon in Figure 11.

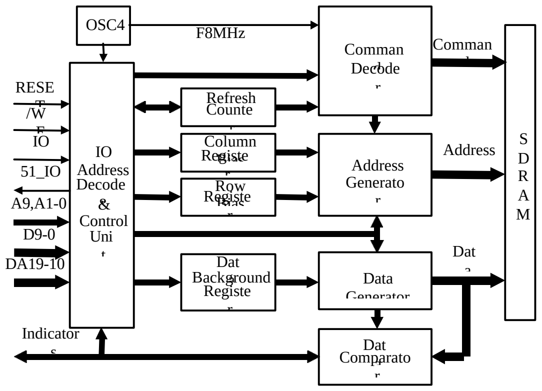


Figure 10 Memory BIST emulator circuit structure

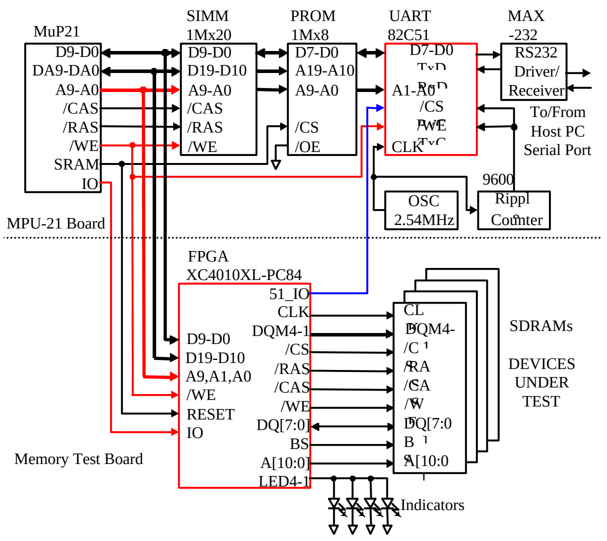


Figure 11. Programmable TPG and Interface Circuitry.

With such a flexible architecture, we are able to test multiple SDRAM with different algorithms. The picture of the MPU-21 board and the memory test board is shown in Figure 12 and 13. The system has been verified on 64M SDRAM [1]. Due to the space limited, the test waveforms are not shown.

11. Conclusion

In this paper, we have presented a flexible memory BIST architecture. It is composed of a simple processor for test flow control and a programmable logic for test pattern and interface signal generation. With such a flexible architecture, it is able to target different types of memories with ease. In addition, we have implemented a computer aided engineering system for automatic test code and test hardware netlist generation. The system has been implemented and verified. The capability in testing multiple memories simultaneously reasserts the flexibility of the architecture.

References

[1] TC59S1616AFT-8,-10,-12A,-12,TC59S1608AFT-8,-1,-12A,-12,TC59S164AFT-8,-10,-12A,-12 Synchro-nous Dynamic RAM Datasheet, Toshiba

[2] R. Dekker, F. Beenker, L. Thijssen, “A Realistic Fault Model and Test Algorithms for Static Random Access Memories”, IEEE Trans. Comp. (USA), C-9, (6), pp. 567-572, 1990

[3] A. J. van de Goor, Aad Offerman, “Toward a Uniform Notation for Memory Tests”,IEEE, pp. 420-427, 1996

[4] P.H.W. Leong, P.K. Tsang, and T.K.Lee, "A FPGA Based Forth Microprocessor," *IEEE Design and Test of Computers*, pp. 245-255, May 1998.

[5] C.H. Ting and C.H. Moore, "MPU21 - A High Performance MISC Processor," *Forth Dimensions*, Jan. 1995.

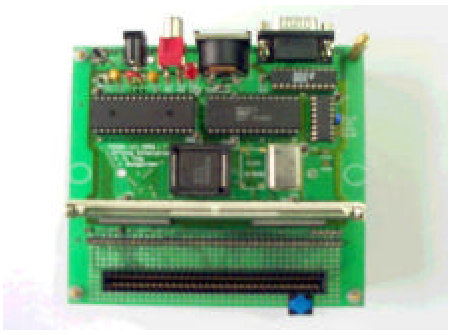


Figure 12 Picture of MPU-21 processor board

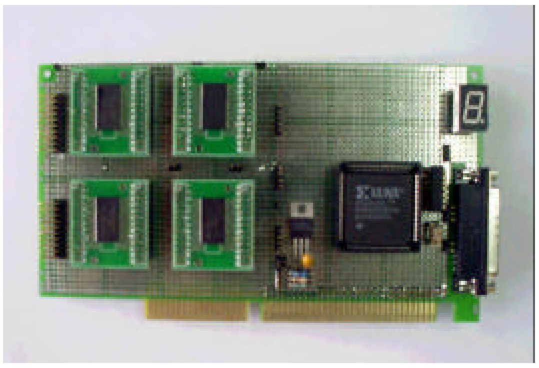


Figure 13. Picture of Memory test boar