Processor-Programmable Memory BIST for Bus-Connected Embedded Memories

Ching-Hong Tsai and Cheng-Wen Wu
Department of Electrical Engineering
National Tsing Hua University
Hsinchu, Taiwan 30013
ROC

Abstract—We present a processor-programmable built-in self-test (BIST) scheme suitable for embedded memory testing in the system-on-a-chip (SOC) environment. The proposed BIST circuit can be programmed via an on-chip microprocessor. Upon receiving the commands from the microprocessor, the BIST circuit generates pre-defined test patterns and compares the memory outputs with the expected outputs. Most popular memory test algorithms can be realized by properly programming the BIST circuit using the processor instructions. Compared with processor-based memory BIST schemes that use an assembly-language program to generate test patterns and compare the memory outputs, the test time of the proposed memory BIST scheme is greatly reduced.

I. Introduction

With the advent of deep-submicron VLSI technology, core-based system-on-chip (SOC) design is attracting an increasing attention. On an SOC, popular reusable cores include memories (such as ROM, SRAM, DRAM, and flash memory), processors (such as CPU, DSP, and microcontroller), input/output circuits, etc. Memory cores are obviously among the most universal ones—almost all system chips contain some type of embedded memory. However, to provide a low-cost test solution for the on-chip memory cores is not a trivial task [1, 2].

One possible solution that is also the most widely used for testing embedded memories is built-in self-test (BIST). The integration of BIST with the embedded memory under test greatly minimizes the need for using expensive memory testers. It also reduces the test time [2]. The research in memory BIST has a long history (see, e.g., [2–8]). However, most of the BIST approaches proposed so far assume that the BIST circuit is to be integrated with the RAM circuit, whether the BIST circuit is processor based or finite-state machine (FSM) based. The advantage of such a scheme is that the test time is short and the area overhead is relatively small, especially for the FSM-based approach [2]. There also are good reasons for such one-BIST-per-RAM approach, e.g., intellectual property (IP) protection, test wrapping for IP [9, 10], performance requirement, etc. However, sometimes it is not feasible to have one BIST circuit for each memory core. For example, a typical ASIC or SOC has tens of SRAM cores with different sizes and configurations. If each memory core on chip requires a BIST circuit, then the area and test pin overhead will be untolerable. Serial interface has been proposed that reduces hardware overhead [4], but the test time is long and diagnosis cannot be supported. Therefore, in [11] a BIST scheme which utilizes an on-chip microprocessor to test the memory cores was proposed. The memory BIST is done by executing an assembly-language program in the on-chip microprocessor to generate test patterns (including the address sequence, data patterns, and control signals) and compare the memory outputs with the expected correct data. The advantage of such a scheme is that it is highly flexible because various test algorithms can be realized by simply modifying the assembly programs run on the microprocessor. It also is easy to support testing of multiple memory cores. However, the test time is much longer than using an integrated memory BIST circuit. We will discuss this point and show some of our experimental results in the next section.

In Sec. 3, we will propose a memory BIST scheme which utilizes a processor-programmable BIST circuit to realize a test algorithm using pre-defined test elements. The BIST circuit also compares the memory outputs with the expected correct data to generate a go/no-go signal. The approach is a combination of the on-chip processor-based BIST of [11] and the FSM-based BIST proposed in [2].

Since the BIST circuit is an independent one (i.e., not integrated with any memory core), it can be considered as a core (or IP) in itself. The proposed BIST scheme has at least the following advantages: 1) the test time is short due to dedicated BIST core design, 2) the flexibility of processor-based BIST is maintained, and 3) multiple memory cores can be supported without multiple BIST cores, multiple sets of external test pins, or complicated routing. We have used an on-chip 6502 microprocessor to perform experiments on our idea. The proposed BIST scheme takes about 10N clock cycles to perform the March C– test algorithm, while the on-chip processor-based BIST scheme [11] takes about 114N clock cycles for the same algorithm, where N is the address space of the memory core. Moreover, the memory access frequency (for continuous read/write sequences) in [11] is much lower than the proposed approach due to overheads in the assembly-language program.

II. On-Chip Processor-Based Memory BIST

Almost any SOC (or even ASIC) design has on-chip processor and memory cores. In addition to normal operation, the on-chip processor core also can be used to test other cores on the same chip. In [11], the processor was used to test embedded logic and memory cores using instructions of the processor core. An assembly-language program was used to realize memory test algorithms and compare the memory outputs with the expected correct data. We also have used the 6502 processor in a similar experimental on-chip processor-based memory BIST set-up. Though a 6502 assembly program was used to implement such a BIST scheme (see Appendix A), our approach is applicable to other on-chip processors. The memory test algorithm used in our experiment is the March C– algorithm [12], as shown in Table I.

TABLE I
The March C– test algorithm.

M_0	M_1	M_2	M_3	M_4	M_5
$\Updownarrow(w0)$	$\Uparrow(r0w1)$	$\Uparrow(r1w0)$	$\Downarrow(r0w1)$	$\Downarrow(r1w0)$	$\Updownarrow(r0)$

A memory test algorithm consists of a finite sequence of *test elements*. Each test element specifies a certain address (sequence) and a combination of read/write operations (and maybe other memory operations) to be applied to the specified address (sequence). Each test element in a March algorithm is called a *March element*. In Table I, there are six March elements, i.e., $M_0,\ M_1, \ldots,\ M_5$. In each March element, the address sequence is specified first: $\Uparrow$ stands for the ascending addressing order, $\Downarrow$ stands for the descending addressing order, and $\Updownarrow$ means that the addressing order can be either $\Uparrow$ or $\Downarrow$. For each memory cell addressed in the specified order, we perform the read/write operations given inside the parentheses before advancing to the next specified address. The algorithm is sometimes called the March $10N$ algorithm since it requires $10N$ read/write operations, where N is the number of memory cells.

TABLE II
Clock cycles for major 6502 instructions.

Instruction	Addressing Mode	Clock Cycles
LDA	Immediate	2
LDA	Absolute Index X	4
LDX	Immediate	2
STA	Absolute Index X	4
INX	Imply	2
CPX	Imply	2
BNE	Relative	$2 \sim 4$
CMP	Imply	2

Table II shows the clock cycles for some major 6502 instructions used in the MARCH C– test algorithm. From the table, we can calculate the total execution time of the assembly program in Appendix A in terms of number of clock cycles, assuming that each memory read/write operation takes one clock cycle. The March element M_0 takes $12N$ clock cycles, $M_1 \sim M_4$ take $22N$ clock cycles, and M_5 takes $14N$ clock cycles. Therefore, the total number of clock cycles for March C– is $114N$. The execution time of the above assembly program is about 9.6 seconds for a 4-Mbit memory core (assuming a 50-MHz clock). In contrast, the test time for the same memory core using the integrated BIST core proposed in [2] is only about 0.4 seconds. The test time of the on-chip processor-based BIST approach apparently is much longer than the integrated BIST approach. In the next section, we will propose a memory BIST core with short test time while maintaining the flexibility of the processor-based approach.

III. Processor-Programmable BIST Core Design and Test Flow

A. BIST Architecture

Figure 1 shows the architecture of the proposed BIST scheme. The BIST core is inserted between the CPU core and the on-chip bus, which also connects the memory cores. In normal operation mode, the CPU transparently access the system bus with slight time overhead introduced by the multiplexers. The overhead can be minimized by careful design of the multiplexers which can be integrated with the bus drivers. In memory BIST mode, the BIST circuitry takes over the control of the on-chip bus. It executes certain test algorithm programmed by the CPU and generates the addresses, input data, and control signals for the memory core. It also compares the memory output response with the expected correct data. Since the memory core can be considered as a pure channel, the data received from the memory should be equivalent to those written to the memory previously. The comparison thus can be done without complicated manipulation of the data. To be able to allow these two different modes, we use several multiplexers to multiplex the address bus (ADDR), data input bus (DATAI), data output bus (DATAO), and control bus between the CPU core and the BIST circuitry.

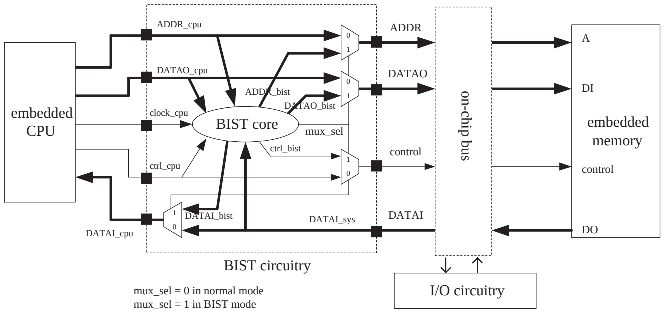

Fig. 1. BIST Architecture.

The BIST circuit is synchronized with CPU core by using the clock signal from the clock input port *clock_cpu* of the CPU. The signals on the address input port *ADDR_cpu*, data input port *DATAO_cpu*, and control input signals *ctrl_cpu* come directly from the embedded CPU's address outputs, data outputs, and control signal outputs, respectively. In the next section we will show that the BIST functions (i.e., the test algorithm) can be programmed by the CPU via the above three BIST input ports. The address output port *ADDR_bist*, data output port *DATAO_bist*, and control output port *ctrl_bist* of the BIST core are connected to the output multiplexers. The outputs of these multiplexers are connected to the address bus *ADDR*, data output bus *DATAO*, and control bus *control*, respectively. During the memory BIST process, the multiplexer selection signal *mux_sel* is set to 1 so that the test addresses, data patterns, and control signals can be sent to the embedded memory via the on-chip bus, and the memory output response can also be read via the input port *DATAI_sys* that is directly connected to the data input bus *DATAI*. The data output port *DATAI_bist* of the BIST circuit can display the contents of the BIST core's internal registers when appropriate address values appear on *ADDR_cpu*. The status of the memory BIST process can be read by the CPU via *DATAI_bist* during the BIST mode because the multiplexer connects *DATAI_bist* to the CPU's data input port. Details about this will be discussed next.

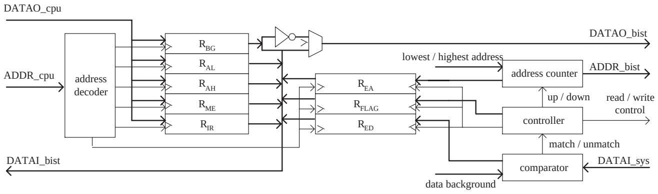

Fig. 2. Block diagram of the proposed BIST circuit.

B. BIST Implementation

Figure 2 shows the block diagram of our BIST circuit. There are several registers in the BIST circuit that are used to store necessary information during the memory BIST process (e.g., data background, lowest and highest addresses of the embedded memory, type of March element, etc.) or store the memory test result (e.g., the BIST core's status, erroneous output response, faulty address, etc.). Table III summarizes the register symbols and their corresponding test functions. Register R_{BG} stores the background data, which is used during the March test. Registers R_{AL} and R_{AH} are used to store the lowest and highest addresses of the memory under test, respectively. Register R_{ME} stores the current March element instruction. The test function performed by the BIST circuit depends on the content of R_{ME}. Register R_{IR} stores the instruction of the BIST circuit. For example, if the CPU writes a *START* instruction into R_{IR}, the BIST circuit will start to run memory BIST. Both registers R_{ED} and R_{EA} store the erroneous data. When the BIST circuit detects a fault in the memory core, the error response and faulty address will be saved into R_{ED} and R_{EA}, respectively. Register R_{FLAG} is the BIST flag register that stores the current status of the BIST circuit. For example, if the BIST circuit detects a fault, the *ERROR* flag will be saved into R_{FLAG}. All the above data registers can be enabled by the address decoder. When proper address values appear in input port $ADDR_{cpu}$, the address decoder will enable the corresponding data register.

TABLE III
SYMBOLS AND FUNCTIONS OF THE DATA REGISTERS IN THE BIST CORE.

Register	Function
R_{BG}	store background data
R_{AL}	store lowest address
R_{AH}	store highest address
R_{ME}	store current March element
R_{IR}	instruction register of BIST circuit
R_{FLAG}	status register of BIST circuit
R_{ED}	erroneous response of defective memory cell
R_{EA}	address of defective memory cell

Other blocks in the BIST circuit include 1) an address counter which generates the test address sequence; 2) a comparator which compares the memory output response with the expected correct data; and 3) a BIST controller which controls the BIST circuit. The address counter is just a simple up/down counter whose value is between R_{AL} and R_{AH}. The comparator compares the memory output response with the content of R_{BG} or its complement, depending on which March element is used. When a discrepancy is found by the comparator, it indicates a fault. The controller design is very simple. When the *START* instruction is stored into R_{IR}, the controller starts its function. It first decodes the current March element instruction stored in R_{ME}, then controls the data output multiplexer *datao_mux* and address counter to generate the appropriate March element for the embedded memory. When an error is found, the controller saves the erroneous output response and faulty address in registers R_{ED} and R_{EA}. When the content of the address counter reaches the lowest or highest address, the controller will write a *FINISH* flag into R_{FLAG} to inform the processor that the current March element is finished.

C. BIST Procedure

The test flow using the proposed BIST scheme is illustrated in Fig. 3. Initially, the CPU core writes the lowest and highest addresses of the memory under test into R_{AL} and R_{AH}, respectively. It then writes the current March element instruction into R_{ME}, and the *START* instruction into R_{IR} to activate the BIST circuit. When the BIST controller senses that the *START* instruction has been written into R_{IR}, the memory BIST procedure begins. The address counter generates the address sequence, the data output multiplexer sends the data background, and the BIST controller generates the memory read/write control signals. During the memory BIST process, the CPU test program continues polling the register R_{FLAG}. If the BIST circuit detects an error, the BIST controller will write an *ERROR* flag into R_{FLAG}, and the CPU core will execute an error handling routine which can feedback the error response and faulty cell address to the test engineer. If all memory cells pass the current March element, the BIST circuit will write a *FINISH* flag into R_{FLAG}. When the CPU core detects the *FINISH* flag, the test program proceeds to the next March element, or quit the test program by writing an *END* instruction into R_{IR} if no more March element is in the queue.

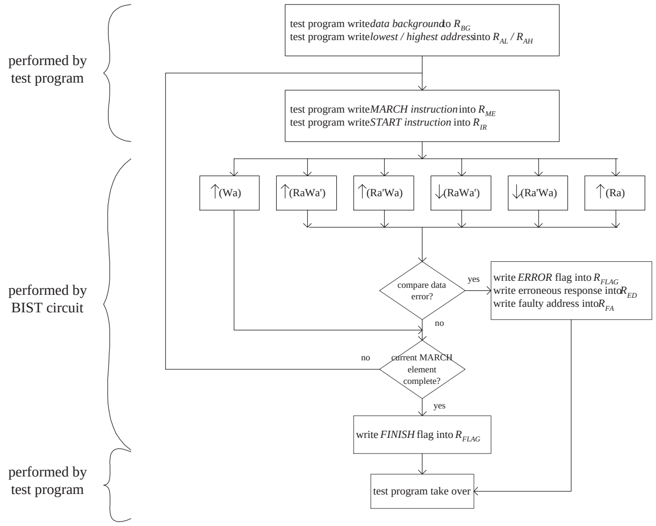
performed by test program
performed by BIST circuit
performed by test program
test program write data background to R_{BG}
test program write lowest / highest address into R_{AL} / R_{AH}
test program write MARCH instruction into R_{ME}
test program write START instruction into R_{IR}
↑(Wa)
↑(RaWa')
↑(Ra'Wa)
↓(RaWa')
↓(Ra'Wa)
↑(Ra)
compare data error?
yes
no
write ERROR flag into R_{FLAG}
write erroneous response into R_{ED}
write faulty address into R_{FA}
current MARCH element complete?
no
yes
write FINISH flag into R_{FLAG}
test program take over

Fig. 3. Embedded memory test flow.

IV. Experimental Result

We have used the Verilog hardware description language to simulate the behavior of the proposed BIST scheme. As described above, the BIST circuit is inserted between the 6502 CPU core and an embedded RAM. In our experiment, the memory test program was stored in a 256x8 ROM. To test the embedded RAM, the assembly program first writes necessary instructions and data into registers R_{BG}, R_{AL}, R_{AH}, and R_{ME}, and activates the BIST circuit by writing the *START* instruction into R_{IR}. Then the test program continues to monitor the flag register R_{FLAG} until the current March element is finished. In Appendix B we show the 6502 assembly program that performs the March C– test algorithm under the proposed BIST scheme. Also, in Table IV we show the March element encoding of R_{ME} for March C–, and in Table V we show the addresses of the registers in the same experiment.

The total test time (in terms of clock cycles) of the proposed BIST scheme equals the cumulated test time of all the March elements, plus 30 clock cycles—the time to initialize the BIST circuit. The test time of each March element is the same as the number of memory read/write operations in each March element (assume each memory read/write operation takes one clock cycle). Note that March element M_0 takes $1N$ clock cycles, March elements $M_1 \sim M_4$ take $2N$ clock cycles each, and March element M_5 takes $1N$ clock cycles, where N is the number of memory cells. Therefore, the proposed BIST scheme takes only $10N$ clock cycles to perform the March C– test algorithm. The test time is greatly reduced as compared with the on-chip processor-based approach. Apparently, using a dedicated BIST core for test algorithm generation and data comparison makes big difference so far as performance is concerned. Moreover, the BIST procedure is controlled by an assembly program, so the programmability of our approach is comparable to the processor-based approach.

TABLE IV
March elements and the corresponding R_{ME} instruction encoding.

March element	M_0	M_1	M_2	M_3	M_4	M_5
Instruction	0_H	1_H	2_H	3_H	4_H	5_H

TABLE V
Addresses of the registers in the BIST experiment.

Register	Address
R_{BG}	FFE0
R_{AL}	FFE1 ~ FFE2
R_{AH}	FFE3 ~ FFE4
R_{ME}	FFE5
R_{IR}	FFE6
R_{FLAG}	FFE7
R_{ED}	FFE8
R_{EA}	FFE9 ~ FFEA

TABLE VI
Comparison of various embedded memory BIST methodologies.

BIST scheme	Test time	Hardware overhead	Routing overhead
Integrated BIST core	Short	Low	High
On-chip processor	Very long	Zero	Zero
Ours	Short	Very low	Zero

Table VI shows the comparison of some embedded memory BIST schemes, including the integrated memory BIST core approach (e.g., [2]), on-chip processor-based approach (e.g., [11]), and our BIST approach. The area overhead of our BIST implementation is lower than that uses an integrated memory BIST core because the BIST controller is very simple (most of the BIST procedure is controlled by the assembly program executed on the CPU). In the case of multiple memory cores in an SOC design, we can use a centralized BIST circuit to test all memory cores, so the routing overhead will be zero because of the use of on-chip bus to access the memory cores. In contrast to using a single dedicated BIST circuit for multiple RAM cores, the routing overhead can be very high because of the wires connected directly between the BIST circuit and the memory cores. Two possible solutions for such a problem are 1) to use the bus-based connection, and 2) to group the memory cores and duplicate the BIST circuits so that routing can be localized.

There is one restriction in our BIST scheme—the proposed BIST scheme cannot be directly applied to those memory cores where the CPU and the embedded memories are connected to different system buses, or the CPU accesses the memories via a memory controller. In such cases, we must slightly modify our BIST scheme so that the BIST core is connected to the same system bus with embedded memories, and the BIST core must be able to be programmed by the CPU via the memory controller.

V. CONCLUSION

We have proposed a flexible and cost-effective BIST scheme for single or multiple memory cores in the SOC environment where an on-chip processor is available. Our approach is flexible because the memory access waveforms can be implemented by the BIST hardware, and different memory test algorithms can be realized by executing proper assembly programs on the on-chip processor core. It is cost-effective because the test time is short and the hardware overhead is low. Besides, with the proposed BIST scheme, neither the CPU nor the memory design need not be modified, thus the BIST design cost is reduced.

REFERENCES

[1] C.-W. Wu, “Testing embedded memories: Is BIST the ultimate solution?”, in *Proc. Seventh IEEE Asian Test Symp. (ATS)*, Singapore, Dec. 1998, pp. 516–517.

[2] C.-T. Huang, J.-R. Huang, C.-F. Wu, C.-W. Wu, and T.-Y. Chang, “A programmable BIST core for embedded DRAM”, *IEEE Design & Test of Computers*, vol. 16, no. 1, pp. 59–70, Jan.-Mar. 1999.

[3] R. Dekker, F. Beenker, and L. Thijssen, “A realistic self-test machine for static random access memories”, in *Proc. Int. Test Conf. (ITC)*, 1988, pp. 353–361.

[4] B. Nadeau-Dostie, A. Silburt, and V. K. Agarwal, “Serial interface for embedded-memory testing”, *IEEE Design & Test of Computers*, vol. 7, no. 2, pp. 52–63, Apr. 1990.

[5] R. P. Treuer and V. K. Agarwal, “Built-in self-diagnosis for repairable embedded RAMs”, *IEEE Design & Test of Computers*, vol. 10, no. 2, pp. 24–33, June 1993.

[6] P. Camurati, P. Prinetto, M. S. Reorda, S. Barbagallo, A. Burri, and D. Medina, “Industrial BIST of embedded RAMs”, *IEEE Design & Test of Computers*, vol. 12, no. 3, pp. 86–95, Fall 1995.

[7] S. Tanoi, Y. Tokunaga, T. Tanabe, K. Takahashi, A. Okada, M. Itoh, Y. Nagatomo, Y. Ohtsuki, and M. Uesugi, “On-wafer BIST of a 200-Gb/s failed-bit search for 1-Gb DRAM”, *IEEE Journal of Solid-State Circuits*, vol. 32, no. 11, pp. 1735–1742, Nov. 1997.

[8] J. Dreibelbis, J. Barth, H. Kalter, and R. Kho, “Processor-based built-in self-test for embedded DRAM”, *IEEE Journal of Solid-State Circuits*, pp. 1731–1740, Nov. 1998.

[9] Y. Zorian, “Test requirements for embedded core-based systems and IEEE P1500”, in *Proc. Int. Test Conf. (ITC)*, Oct. 1997, pp. 191–199.

[10] S. Adham, D. Bhattacharya, D. Burek, C. J. Clark, M. Collins, G. Giles, A. Hales, E. J. Marinissen, T. McLaurin, J. Monzel, F. Muradali, J. Rajski, R. Rajsuman, M. Ricchetti, D. Stannard, J. Udell, P. Varma, L. Whetsel, A. Zamfirescu, and Y. Zorian, “Preliminary outline of the IEEE P1500 scalable architecture for testing embedded cores”, in *Proc. IEEE VLSI Test Symp. (VTS)*, Apr. 1999, pp. 483–488.

[11] R. Rajsuman, “Testing a system-on-a-chip with embedded microprocessor”, in *Proc. Int. Test Conf. (ITC)*, 1999, pp. 499–508.

[12] A. J. van de Goor, “Using march tests to test SRAMs”, *IEEE Design & Test of Computers*, vol. 10, no. 1, pp. 8–14, Mar. 1993.

A. 6502 ASSEMBLY PROGRAM FOR MARCH-C

```
        LDX     #$$00
        LDA     #$$55           ; 8 bit data background 01010101

M0:     STA     0000,X          ; March element 0
        INX
        CPX     #$$FF
        BNE     M0
        LDX     #$$00

M1:     LDA     0000,X          ; March element 1
        CMP     #$$55           ; compare output with 01010101
        BNE     ERROR
        LDA     #$$AA
        STA     0000,X          ; write 10101010 to memory
        INX
        CPX     #$$FF
        BNE     M1
        LDX     #$$00

M2:     LDA     0000,X          ; March element 2
        CMP     #$$AA           ; compare output with 10101010
        BNE     ERROR
        LDA     #$$55
        STA     0000,X
        INX
        CPX     #$$FF
        BNE     M2
        LDX     #$$FF

M3:     LDA     0000,X          ; March element 3
```

```
        CMP     #$$55       ; compare output with 01010101
        BNE     ERROR
        LDA     #$$AA
        STA     0000,X
        DEX
        CPX     #$$00
        BNE     M3
        LDX     #$$FF

M4:     LDA     0000,X      ; March element 4
        CMP     #$$AA       ; compare output with 10101010
        BNE     ERROR
        LDA     #$$55
        STA     0000,X
        DEX
        CPX     #$$00
        BNE     M4
        LDX     #$$00

M5:     LDA     0000,X      ; March element 5
        CMP     #$$55       ; compare output with 01010101
        BNE     ERROR
        INX
        CPX     #$$FF
        BNE     M5
        JMP     FINISH      ; quit memory test program
```

B. 6502 Assembly Program for Proposed BIST Scheme

```
        LDA     #$$55       ; load data background 01010101
        STA     0HFFE0      ; write to $R_{BG}$
        LDA     #$$00       ; load lower byte of lowest address
        STA     0HFFE1      ; write to lower byte of $R_{AL}$
        LDA     #$$00       ; load upper byte of lowest address
        STA     0HFFE2      ; write to upper byte of $R_{AL}$
        LDA     #$$FF       ; load lower byte of highest address
        STA     0HFFE3      ; write to lower byte of $R_{AH}$
        LDA     #$$00       ; load upper byte of highest address
        STA     0HFFE4      ; write to upper byte of $R_{AH}$

M0:     LDA     #$$00       ; load $M_0$ March element
        STA     0HFFE5      ; write to $R_{ME}$
        JSR     BIST        ; run 0th March element test

M1:     LDA     #$$01       ; load $M_3$ March element
        STA     0HFFE5      ; write to $R_{ME}$
        JSR     BIST        ; run 1st March element test

M2:     LDA     #$$02       ; load $M_2$ March element
        STA     0HFFE5      ; write to $R_{ME}$
        JSR     BIST        ; run 2nd March element test

M3:     LDA     #$$03       ; load $M_3$ March element
        STA     0HFFE5      ; write to $R_{ME}$
        JSR     BIST        ; run 3rd March element test

M4:     LDA     #$$04       ; load $M_4$ March element
        STA     0HFFE5      ; write to $R_{ME}$
        JSR     BIST        ; run 4th March element test

M5:     LDA     #$$05       ; load $M_5$ March element
        STA     0HFFE5      ; write to $R_{ME}$
        JSR     BIST        ; run 5th March element test

END:    LDA     #$$04       ; load END instruction
        STA     0HFFE6      ; write to $R_{IR}$
        JMP     FINISH      ; exit test program
```

```
BIST:   LDA     #$$00       ; load START instruction
        STA     0HFFE6      ; write to $R_{IR}$

LOOP:   LDA     0HFFE7      ; read $R_{FLAG}$
        CMP     #$$01       ; check if ERROR flag is set
        BEQ     ERROR       ; jump to error handling routine
        CMP     #$$FF       ; check if FINISH flag is set
        BNE     LOOP

        RTS                 ; else return to main program
```