RAM BIST for RISC-V OTTER MCU

Maxwell Sotnick

California Polytechnic University, San Luis Obispo
max.sotnick@gmail.com

8/3/2020

Abstract

Every system relies on some form of memory. This fact is supported by the increasing real estate of memory in an SOC, which is nearly 90% at the of writing this paper. Given the prevalence and significance of memory in IC technology, there has been much research into the area of memory fault analysis and repair. The focus of this paper is to document the methodology, implementation, and results of an MBIST integrated into the Otter MCU architecture. Through developing this project it was found that the MBIST worked as expected and allowed for high modularity, synergy, and functionality. However, the only notable downside was the inefficiency of the simple March algorithm used. By making modifications to the linker script (link.ld) and taking better advantage of the spatiality described later in the paper, the MBIST model developed here could be made at least 50% more efficient (e.g. shorter runtime).

Table of Contents

I. Introduction

The goal for the EE 532 (VLSI Circuit Testing Lab) was to develop both analog and digital tests for some VLSI. For our project, the digital team decided to design a custom board for performing digital tests on the RISC-V Otter MCU used in CPE 233. The digital design team then split into separate groups, tasked to design and implement a digital test circuit which could be integrated into the already functioning MCU architecture.

For my digital test I wanted to focus on memory whether it be the PROM or the RAM. Since the memory module is split between both the program memory and the data memory (including separate address and data lines), I decided to focus on developing a test for RAM. The logistics of implementing a comprehensive test for memory is much simpler for RAM than it is for ROM. Given that the MCU will be loaded onto an FPGA—in our case a Xilinx Artix 7—I decided on a simple MBIST model which utilizes a basic MARCH algorithm for generating test sequences. Additionally, it was important to develop a BIST that would function concurrent to the normal operations of the RAM. Given that stipulation, there would need to be an additional module to trigger the test and halt the PC. Furthermore, the BIST would need to be designed with the intention that the program could continue to run post-analysis, therefore the test would have to reload stored data per the address being tested.

Usually when designing test circuitry for some DUT involving sequential logic, it is possible to leverage combinational logic test methods by translating the sequential circuitry into combinational [1]. That is not the case with memory such as the dual-port BRAM used in our MCU's architecture. Where the problem arises is in the way the memory cores are modeled on the FPGA, therefore leading to far more complex hardware overhead. One solution for testing memory comes in the form of the MBIST structure. By utilizing a dedicated hardware component which is integrated into the structure of the RAM, the memory can be tested using a functional fault model rather than a structural one. Moreover, there are several benefits that come with using the MBIST structure:

1. The tests patterns can be easily generated by using counters, shift-registers, and decoders. Furthermore, the aforementioned hardware can be configured to follow a variety of March test algorithms (which will be further explained in the Methodology section).
2. Test vectors need to be written and read throughout the memory, which can be tracked by counters—even being reflected in the encoded sequence given a particular test pattern.
3. RAM is used regularly in mid to high complexity programs, and due to the complexity level of the hardware overhead, are at a risk of developing a multitude of faults [2]. Integrating the MBIST structure into the memory allows for more frequent and seamless tests to be performed.

One of the key benefits in using an MBIST design is in being able to integrate testing into the operation of the circuit. The triggering of a BIST is highly variable depending on application—some BISTs are triggered via an external I/O, while others are designed to run at the start and end of a program. For the MBIST in this project, a separate module is

responsible for triggering the MBIST given a certain condition (further detail is given in the Methodology section of this paper).

It should be noted the types of faults generally present in memory are as follows [1,2]:

1. **Stuck-At Fault**: Memory cell is permanently fixed at 1 or 0.
2. **Stuck-Open Fault**: Memory cell is not accessible.
3. **Transition Fault**: High-to-low and low-to-high transitions are mutually exclusive in the faulty cell.
4. **Coupling Fault**: The transition of a value in one cell causes the change in value of an adjacent/neighboring cell (hence "coupling"). There are a variety of classifications of coupling faults such as "state", "inversion", and "bridging".
5. **Read-Disturb Fault**: Memory cell transitions upon being repeatedly read.

Given the faults above, the goal is to develop an MBIST that can account for as many fault types possible, and easily allow for post-March analysis of the problematic cell/word. Finally, there is some notation associated with March test algorithms in order to easily describe and identify elements of fault models. The following table is a compilation of several commonly used notations, and is what will be used in the following sections of this paper [2,3]:

Symbol	Description
↑	Rising Transition
↓	Falling Transition
↕	Rising or Falling Transition
⇡	Increasing Memory Address
⇣	Decreasing Memory Address
⇡ / ⇣	Increase or Decrease in Memory Address
__ ϵ […]	Operation at Cell/Word
S/*F* ϵ […]	Fault in Cell/Word where *S* is the Value or Operation that Activates the Fault, and *F* is the Faulty Value
R/*W*	Memory Read / Memory Write
R0/*W0*	Memory Read 0 / Memory Write 0
R1/*W1*	Memory Read 1 / Memory Write 1

Table 1. March Algorithm Notation

II. Methodology

As described in the introduction, there are several ways of testing memory (whether it be utilizing some external circuit, or integrating a self-test module into the DUT). Given that the primary goal is to comprehensively test the Otter's dual-port RAM, the test circuit's design hinged on the secondary deliverables. A few examples of these secondary design requirements were:

1. The test circuit must be self-triggered and should not be active unless an error is detected in the RAM.
2. The test circuit should be easily modified to utilize different March algorithms.
3. Any data in RAM at the moment the BIST begins must be retained and available upon completion of the March sequence.

Taken under consideration, an MBIST best matches the design requirements/features listed above. Furthermore, the MBIST structure is relatively simple and can easily be elaborated on for more complex functionality. Fundamentally, an MBIST is comprised of a counter, decoder, comparator, and controller. Variations on this simplified model are what give the MBIST greater functionality, or aid in integrating into some VLSI. Outside of the MBIST, there is a trigger to activate the BIST and disable the MCU's PC. The following sub-sections detail the technical aspects of each module and its components.

II.I MBIST Structure

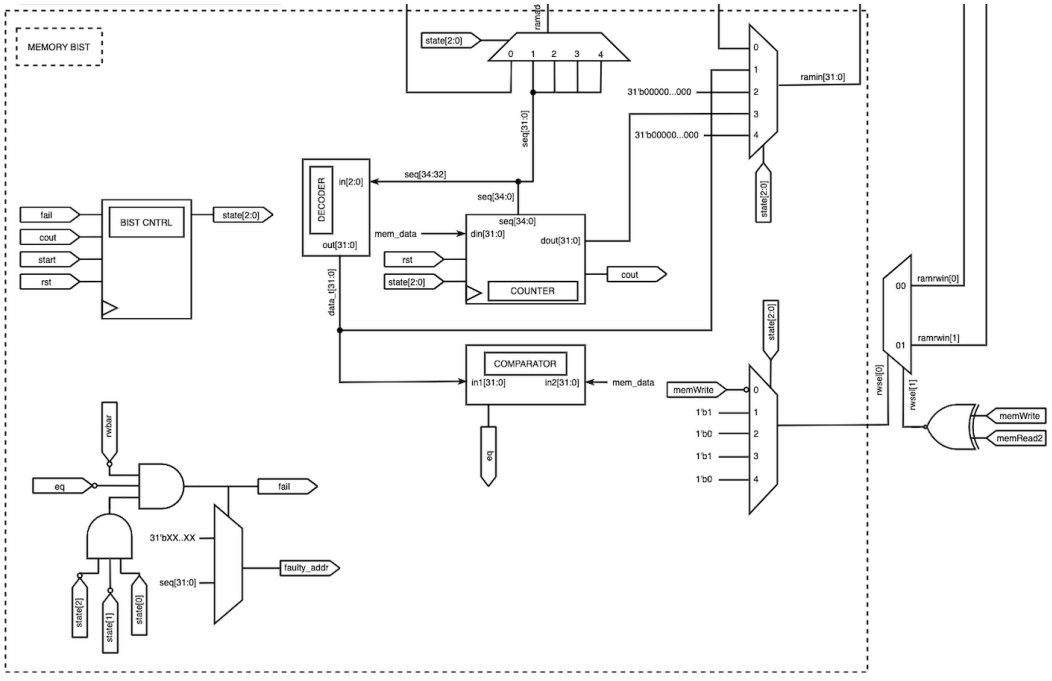

Fig 1. MBIST Black Box Diagram

II.I.A BIST Controller

The BIST Controller is responsible for containing the FSM logic, reacting to flags set by the comparator, counter, and overhead module (wrapper). Through this structure, the MBIST can remain synchronous with the MCU, easily reading and writing to the RAM.

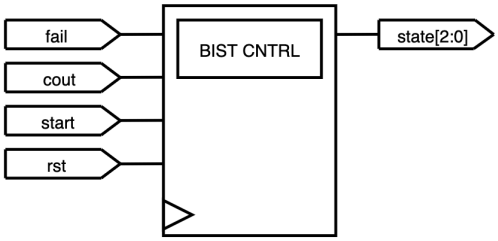

Fig 2. BIST Controller Black Box Diagram

Part of the March sequence implementation is to have the BIST run off of an FSM, allowing the various operations (as specified by the algorithm) to occur in an ordered fashion. Since one of the design requirements is to retain user/program whilst marching through the RAM, the FSM must be designed to account for

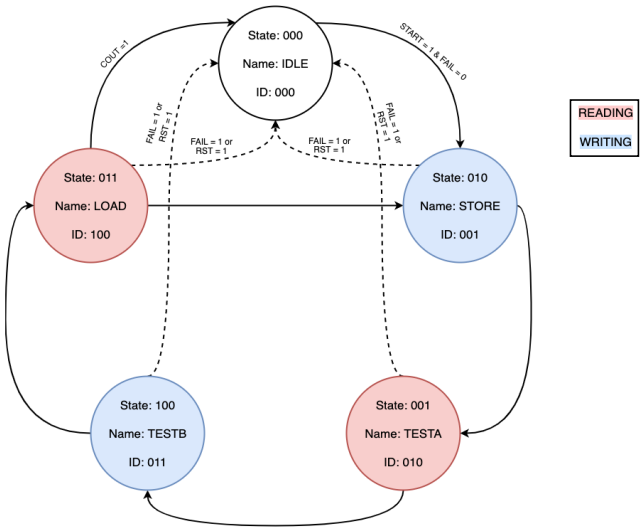

Fig 3. MBIST State Diagram (FSM); red indicates when RAM is being read from, while blue indicates when RAM is being written to.

storing and loading the original data at the current March sequence address. Subsequently, the output of the BIST Controller is the current state, dictating the function of the counter and MBIST wrapper. An effect of the **state** flag is whether

the RAM is being read/written to, as evident in Figure 2. There are five states in the MBIST FSM:

1. **Idle**: This state is defined by normal operation of the MCU. When in this state the MBIST is effectively disabled, and read/write operations perform unhindered.
2. **Store**: The store state is responsible for saving the "original" data (data stored prior to MBIST operation) at the current address being tested.
3. **TestA**: TestA is the first half in testing an address. In this state the sequence generated by the counter is written into the RAM.
4. **TestB**: TestB is the following half in testing an address, where the written sequence value is read out and compared to find any discrepancies.
5. **Load**: The load state is the final stage in the test cycle as it loads the "original" data back into the tested address. Moreover, this state is where the counter increments, therein performing the march.

States cycle from one another as depicted by Figure 2, without any condition other than the positive edge of the clock. However, flags are responsible for some transitions; **cout** and **rst** are responsible for ending the BIST, while **start** and **fail** are needed to start the BIST.

II.I.B Counter

Like the BIST Controller, the Counter is a key component in implementing a March algorithm. The primary job of the Counter is to generate ("march through") sequences that are utilized by the rest of the BIST. Sequences are

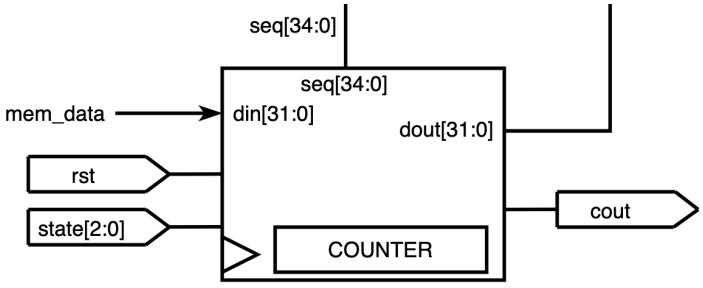

Fig 4. Counter Black Box Diagram

characterized by two sections of binary values. The first three MSBs [34:32] form the encoded value for the test data to be written, read, and compared. Whereas the bottom thirty-two LSBs [31:0] represent the current address being tested. The carryover bit [35] is labeled **cout**, and is used as a flag for stopping the BIST. The process of marching through the RAM is easily accomplished by a counter since the goal is to visit every address whilst testing different values. For the Counter in this paper, I opted to using a simplified structure—where an incrementing value determines the address and data, given a fixed write and read cycle. Such

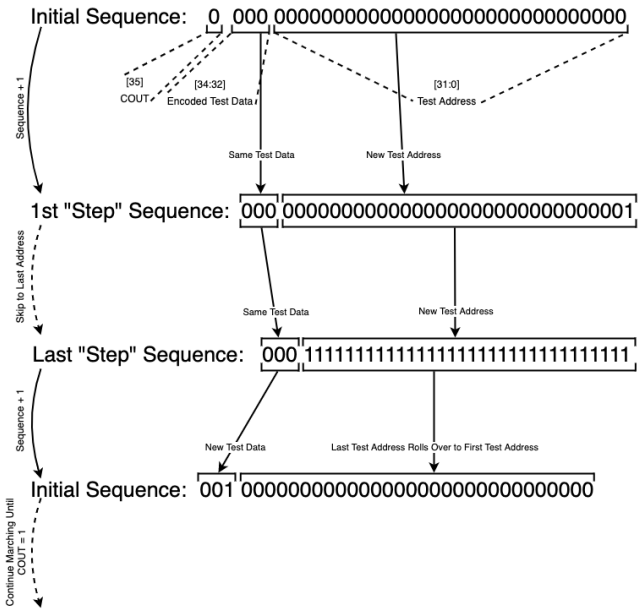

Fig 5. Sequence Generation as Done by the Counter Module; Further Detail on Test Data is Given in the Decoder Section (II.I.C)

that a different March algorithm were to be used, the Counter module could be easily modified. Nonetheless, even though this MBIST utilizes a simple March, most of the faults listed in the Introduction can still be caught by it—Read-Disturb Faults require a more involved algorithm in order to achieve the conditions needed to observe said fault. Further clarifying, Figure 4 depicts the sequence cycling through all of the addresses per a single test data. Upon completing one cycle of read/writes through the RAM, the next cycle begins with a new test data. More information on the test data will be given in the Decoder section (I.I.C).

Finally, the counter is also responsible for storing and loading the "original" data stored from the address currently being tested. The reason for adding this functionality to the counter is to better match the timing of the FSM, and to take advantage of the register already needed to store the current sequence. Consequently, the **state** flag is an input used to select the current function of the counter—whether that be storing/loading the "original" data, or incrementing (generating) the sequence. Just as the **rst** flag is used in the BIST Controller, it functions similarly in resetting the counter to its initial state and sequence.

II.I.C Decoder

Where the first two modules discussed were sequential, the last two modules are instead combinational. Being of the later category, the Decoder module has a relatively straightforward function: decode the input 3-bit sequence and output

the corresponding 32-bit test data. As a reference, Table 2 lists all of the encoded

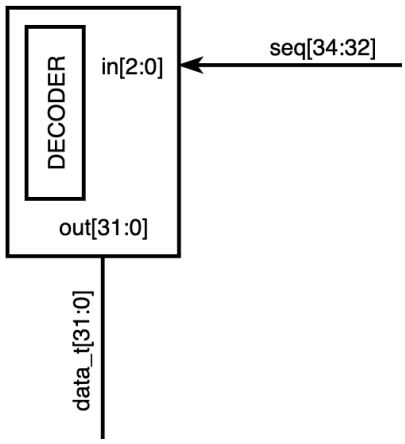

Fig 6. Decoder Black Box Diagram

values and their corresponding decoded test data.

Encoded Input: seq [34:32]	Decoded Output: data_t [31:0]
[000]	[00000000000000000000000000000000]
[001]	[00000000000000001111111111111111]
[010]	[00110011001100110011001100110011]
[011]	[01010101010101010101010101010101]
[100]	[11111111111111111111111111111111]
[101]	[11111111111111110000000000000000]
[110]	[11001100110011001100110011001100]
[111]	[10101010101010101010101010101010]

Table 2. Decoded Tests to Verify RAM Functionality

Compared with the Counter, the Decoder is very simple and easily modified. The benefit of this is that test data can be modified easily without having to change how sequences are generated. From Table 2, the following description assign to the simple March used:

{⇑(W0, R0); ⇑(Wϵ [000..111], Rϵ [000..111]); ⇑(Wϵ [0011..0011], Rϵ [0011..0011]); ⇑(Wϵ [010..101], Rϵ [01..01]); ⇑(W1, R1); ⇑(Wϵ [111..000], Rϵ [1100..1100]); ⇑(Wϵ [101..101], Rϵ [101..101])}

II.I.D Comparator

Being the simplest module of the MBIST, the Comparator is a relatively straightforward circuit. The purpose of the comparator is to contrast both the test data from the Decoder and the read data from RAM. If both values match, the Comparator sets the **eq** flag high, otherwise the flag is kept low (indicating an error has been detected).

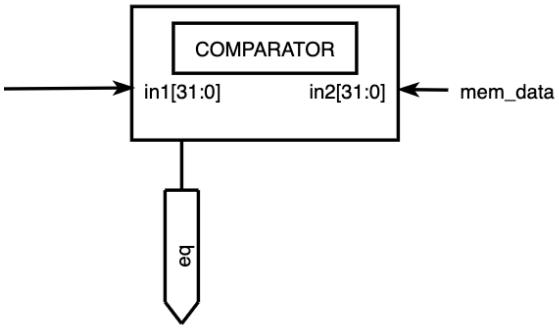

Fig 7. Comparator Black Box Diagram

II.I.E Muxes & Demux

There are four muxes and one demux in the MBIST. Three of the muxes are used to select between the normal MCU data lines (data, address, r/w) and the data lines given by each state of the MBIST. The selector line for theses three muxes is the **state** flag, hence choosing the output based on the current state.

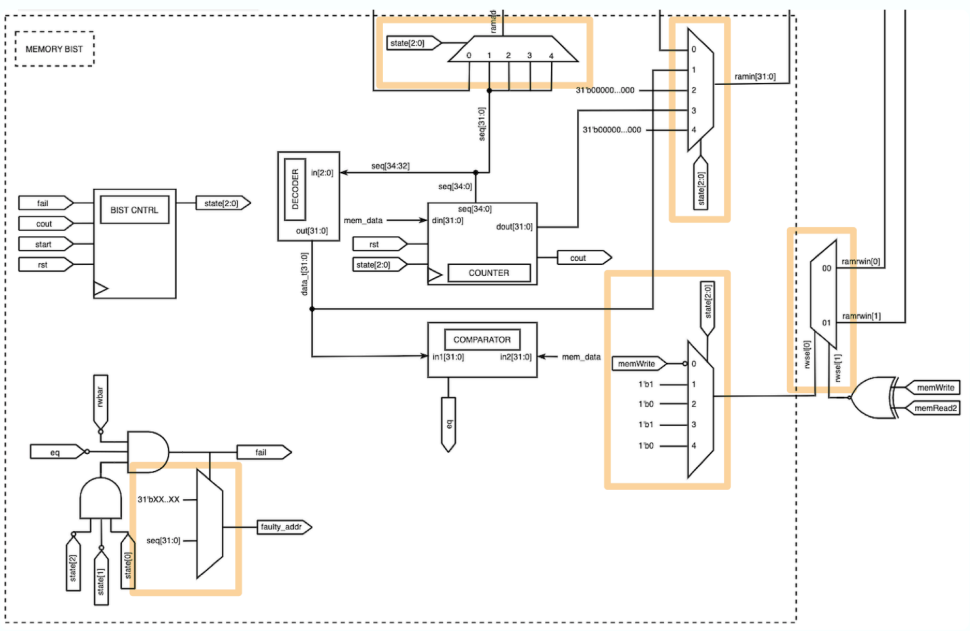

Fig 8. MBIST with Muxes and Demux Highlighted

Connected to the r/w mux is a demux. The RAM in the MCU uses two separate signals to indicate a read or a write. Consequently, r/w mux—which treats reading and writing as a single signal operation—requires a demux to properly select the correct line to set high. The demux utilizes two bits to choose from four outputs, however its only the first two output (00 and 01) which effect the read and write lines to the RAM. The final mux is connected to a small portion of logic which is used to determine the whether or not to set the **fail** flag. Here the mux is connected to the **fail** line and selects between the current address and 0, where if there is a detected fault the current address is outputted as the **faulty_addr**.

II.II MBIST Trigger

As per the design requirements for this project, the MBIST needs some way to become active amidst normal operation of the MCU. Through the use of a trigger module, an MBIST test is more smoothly integrated. Moreover, using a trigger module gives the designer/user the ability to easily change the conditions needed for activation.

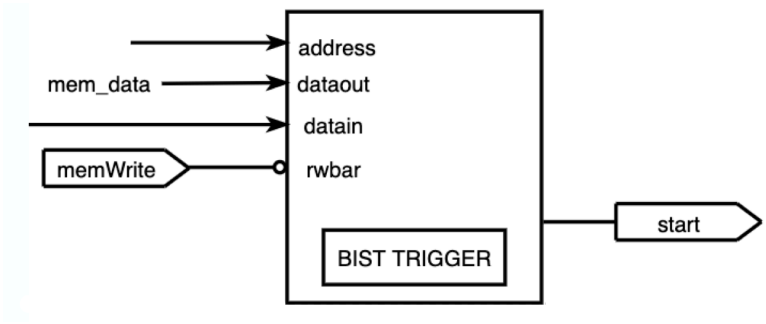

Fig 9. BIST Trigger Black Box Diagram

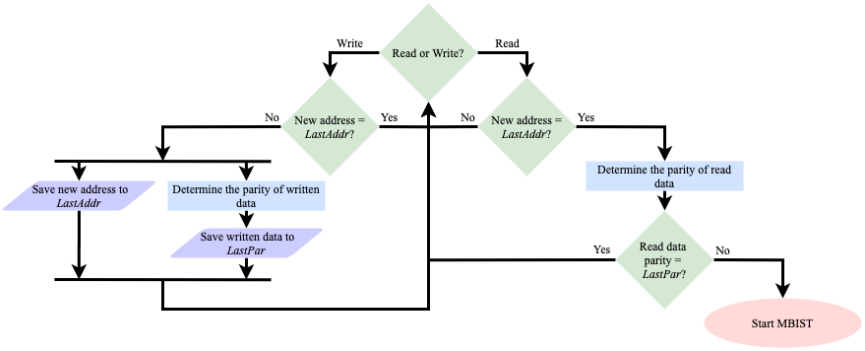

Fig 10. Conditional Flowchart for How the BIST Trigger Functions

The BIST Trigger functions by leveraging the temporality of address read operations. In programs it is common to have an address read from shortly after being written to. A primary example of this temporality is present in subroutine calls, where the program address is stored in RAM and then read back once the subroutine is completed. Given this aspect of RAM usage by the MCU, the BIST Trigger compares parities of the written

and read value (of the same address) to spot an error. The challenge in developing a trigger module for a BIST comes in the form of compromise. To make a really involved and comprehensive trigger means slowing down the MCU and diminishing its efficiency. Given that core compromise, there are a slew of techniques and strategies for making an effective trigger. As detailed above, the strategy chosen for this project's trigger was to leverage bit parity and the temporality of memory.

II.III MBIST & Trigger Integration

Connecting the BIST and BIST Trigger to the MCU architecture requires some rewiring and flag integration. The MCU has two different address and data lines, one set designated for the instruction file, and the other for scratch RAM during runtime. It is the second pair of data lines that have been rewired in order to integrate the MBIST. As for the BIST Trigger, the **start** flag is connected to an AND gate alongside the **PCwrite** flag. By ANDing **PCwrite** the inverse of **start**, the PC is disabled during MBIST activation, and only begins once testing of the memory is over.

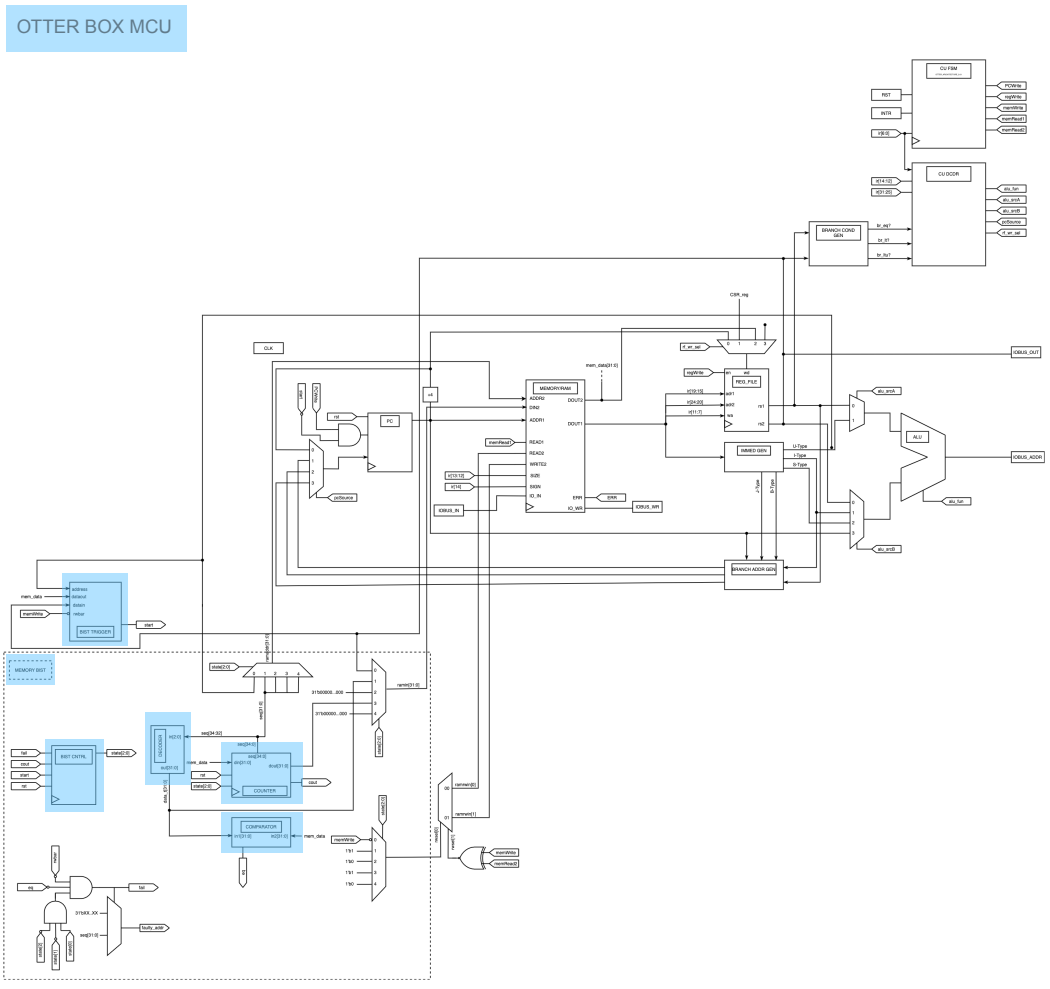

Fig 11. Otter Box MCU with MBIST and BIST Trigger Integrated [5]. Highlighted Regions are Hyperlinked to their Respective Code at the end of the Document.

III. Results

To test the functionality of the MBIST, the modules were built and simulated in Vivado. In writing up the testbench, it was important to highlight different aspects of the MBIST's operation such as the way "original" data is stored prior to any single test. Moreover, showing how the MBIST detects and responds to a fault (in the case used for the tests below, a stuck-at-fault). There are four images related to operation and function of the MBIST:

1.

Fig 12A&B. Start of the MBIST Simulation. Runtime of Figure(s) is 0 to 120 ns

At the very beginning of the simulation is what would be normal operation of the Otter MCU. In the case of the testbench, two write instructions were hardcoded in—both showing the RAM prior to testing, and establishing which addresses have data stored. Addresses 4 and 7 have data stored (DEL457D2 and FFFFE0BF, respectively) and will be tracked later on in the test cycle to show data retention for an address under test. The last notable aspect of Figure(s) 12A&B is that the MBIST trigger detects an error upon reading the data at address FFFFFFFE. Instead of the data being EEBA2822, the data read out is ECBA2822—fault with one of the bits means that the parity check is triggered. Because of the detection, the **start** flag is set high and the MBIST is activated.

2.

Fig 13A&B. Start of the MBIST Simulation and First 7 Addresses to be Tested. Runtime of Figure is 0 to 740 ns.

Following the MBIST activation in Figure 12A&B, the MBIST begins to store, test, and load data to each address of the RAM. The first eight addresses are stored, tested, and loaded as per what was detailed in the Methodology section. Furthermore, at addresses 4 and 7 the "original" data that was written at the very beginning of the simulation, is successfully retained. Markedly, the data being tested at each address (per this test cycle) is 32'b000…0000 since the first test data is all 0's. Once all addresses have been tested per the current test data, the upper address rolls over to address 0 and the test cycle starts again with a new test data.

3.

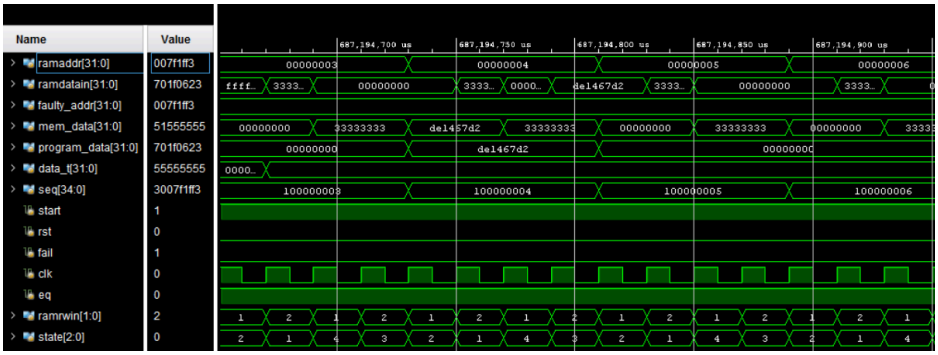

Fig 14A. Second Test Cycle of MBIST (Current Test Data is 32'b0000…11111). Runtime of Figure is 687,194,650 us to 687,195,250 us.

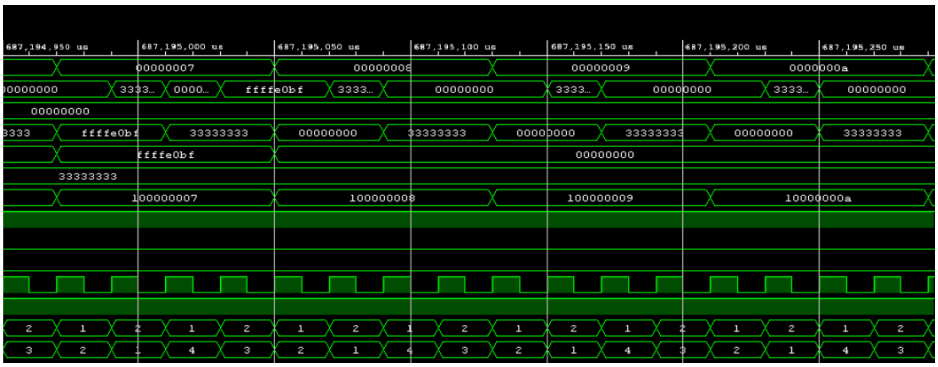

Fig 14B. Second Test Cycle of MBIST (Current Test Data is 32'b0000…11111).
Runtime of Figure is 687,194,650 us to 687,195,250 us.

Moving further along the test runtime, Figure 14A&B highlights another data test used to catch faults. Moreover, the images above show that the "original" data at addresses 4 and 7 have been retained due to the store and load stages of the MBIST.

4.

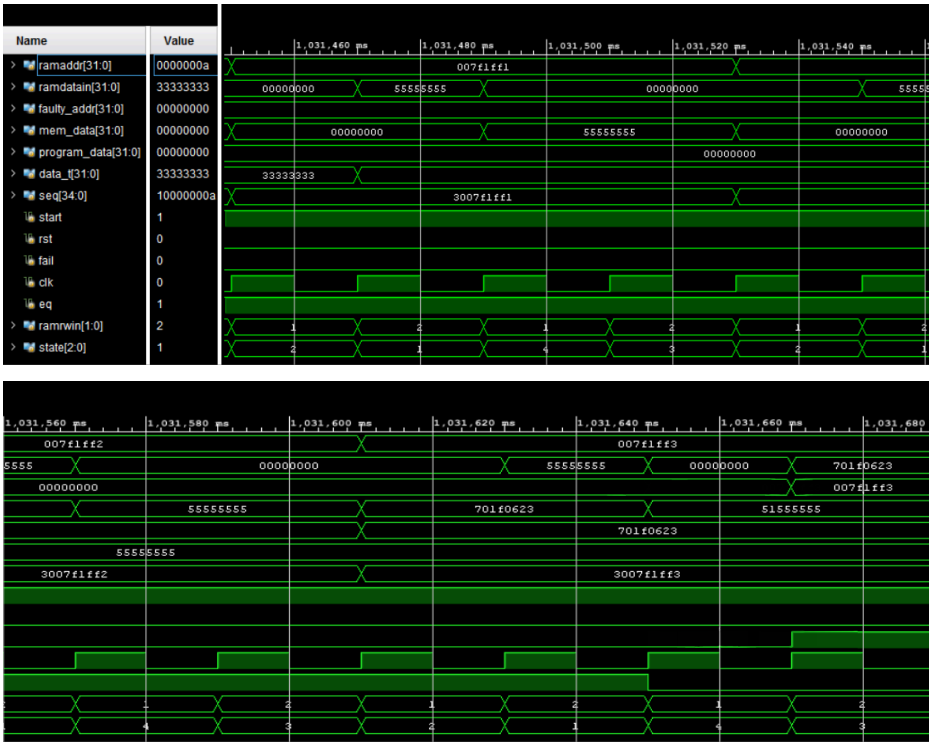

Fig 15A&B. MBIST Finally Catches Faulty Address and Sets Fail Flag High.
Runtime of Figure is 1,031,455 to 1,031,680 ms.

Figure 15A&B are the last collection of images featuring the simulation results. The data sequence has now moved onto 32'b010…101, and upon testing the address 007F1FF3, the **eq** flag goes low. Since a fault has been detected, the MBIST waits for the next clock edge to set the **fail** flag and capture the current address in **fault_addr**.

IV. Discussion

Through the simulations it is evident that the MBIST ran successfully and achieved all of the design requirements laid out at the beginning of this paper. Given that the project was a success, there is much room for improvement and optimization. Moreover, there are several variations that wouldn't necessarily improve the MBIST, but instead give the MBIST a different functionality.

IV.I Improvements

The largest improvement/optimization would be to reduce the runtime of the MBIST. If the MBIST were to run during normal operation of the MCU (running a program), the user experience would come to a halt having to wait for the RAM test to run in its entirety. Due to the scope of this project, these improvements were not completely pursued—though may be future additions of a MBIST V.2:

1. **link.ld**: The Otter Box MCU utilizes a linker file to allocate certain parts of the RAM for specific storage reasons. This way scratch RAM used by a program does not overlap with the program file, register, etc. If edited, the linker file could allow for specific addresses to be reserved for testing (avoiding the need to store and load data outside of the test itself). By designating certain addresses for testing, the MBIST could remain as effective in catching faults due to the spatiality of faults in memory. The core idea behind this method of BIST testing is that only a fraction of the addresses need to be tested since any given memory cell's "health" is tied to the memory cells neighboring it. Granted, this is not a fool proof method of testing, however it reduces both the number of addresses needed for testing, and the cycles per address needed to store and load data.

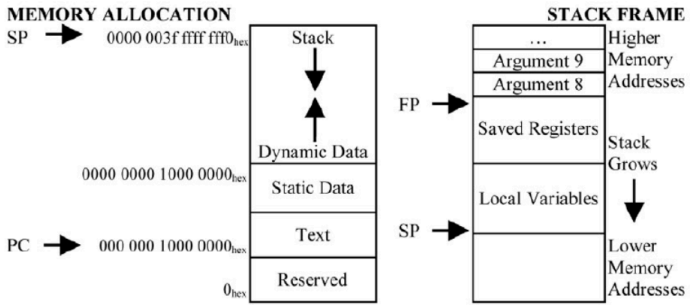

Fig 16. Representation of Memory Allocation as Described by the link.ld [4].

2. **March Steps**: Since specifying certain test address in the linker file is quite difficult to do, the other method of achieving a similar effect would be to increase the size of the steps taken by the Counter module. Currently, the Counter is adding the sequence by 1, which means that it is checking every address for faults. By setting the increment to 5 or 10, the counter would be checking a few addresses per neighborhood of cells. Moreover, a more complex counting algorithm could be used to target specific test cells, minimizing the addresses tested while maximizing the coverage of RAM

being tested. The only drawback with this method is that the additional store and load cycles (which are associated with the "original" memory) must be kept. However, the runtime improvement may cancel the added cycling time, so this method may be considered an overall optimization.

IV.II Variations

I would be remise if I didn't mention other March algorithms and how they would be implemented; therefore, I will briefly describe them and talk about how the MBIST may be edited to accommodate their implementation.

Even though March algorithms are all meant to test memory, they come in a variety of flavors and operate uniquely from one another. An example of this is in the types of faults that each algorithm covers—March C- is very bad at detecting stuck-open faults, while MATS++ has a near perfect detection of said fault.

TABLE 8.3 ■ Fault Coverage of MATS^{++}

Fault	P_1	P_2	P_3	$P_{2,3}$	$P_{1,2,3}$	P_{all}
SAF	100%	100%	100%	100%	100%	100%
SOF	100%	100%	100%	100%	100%	100%
TF	100%	100%	100%	100%	100%	100%
AF	99.7%	99.9%	99.9%	100%	100%	100%
CFin	100%	100%	100%	100%	100%	100%
CFid	37.5%	37.5%	37.5%	62.6%	75.9%	89.1%
CFst	50.0%	50.0%	50.0%	75.0%	87.5%	100%

TABLE 8.4 ■ Fault Coverage of March X

Fault	P_1	P_2	P_3	$P_{2,3}$	$P_{1,2,3}$	P_{all}
SAF	100%	100%	100%	100%	100%	100%
SOF	0.8%	0.8%	0.8%	0.8%	0.8%	0.8%
TF	100%	100%	100%	100%	100%	100%
AF	99.7%	99.9%	99.9%	100%	100%	100%
CFin	100%	100%	100%	100%	100%	100%
CFid	50.0%	50.0%	50.0%	78.1%	90.7%	100%
CFst	62.5%	62.5%	62.5%	84.4%	93.0%	100%

TABLE 8.5 ■ Fault Coverage of March C^{-}

Fault	P_1	P_2	P_3	$P_{2,3}$	$P_{1,2,3}$	P_{all}
SAF	100%	100%	100%	100%	100%	100%
SOF	0.8%	0.8%	0.8%	0.8%	0.8%	0.8%
TF	100%	100%	100%	100%	100%	100%
AF	99.7%	99.9%	99.9%	100%	100%	100%
CFin	100%	100%	100%	100%	100%	100%
CFid	99.9%	99.9%	99.9%	99.95%	100%	100%
CFst	99.9%	99.9%	99.9%	99.95%	100%	100%

Fig 17. Fault Coverage Table of Several March Algorithms [2].

Given the improvements described above, the March algorithm that I think would best be suited as a variation for the current MBIST structure is March C-. Since the improvements highlight the spatiality of faults and the relationships of addresses in neighborhoods, March C- would be the best fit since it has high detection rates for coupling faults. To implement a different March algorithm, the Counter and Decoder modules need to be altered to accommodate the factors used in generating test sequences. In modifying the counter, the key is in understanding how the sequence is modified from each test cycle. Once a pattern is observed, then it becomes easy to modify the Counter and Decoder to match the desired characteristic.

V. Conclusion

The purpose of this project, as stated at the beginning of this paper, was to develop test circuitry for the dual-port BRAM of the RISC-V Otter MCU. The test circuitry needed to keep "original" RAM data intact, be self-triggered, and be easily modifiable for different March algorithms. It was determined that an MBIST structure would best meet all of the design requirements specified above. Moreover, the MBIST would allow for tests to be easily conducted alongside the operation of the MCU. Through implementation and simulation, it was demonstrated that the MBIST functioned as expected. However, it was also noted that the time needed to run the MBIST in its entirety is especially long in comparison to other MCU operations. With the modifications given in the Discussion section, the runtime of the MBIST would reduce by at least 50%. I may eventually optimize the MBIST in the manner described above, but that is currently out of the scope of this paper. In summary, the project of developing test circuitry for the dual-port BRAM of the RISC-V Otter MCU, was ultimately a success, and promises room for development/modification.

V.I Postscript

I would like to make a special note to the professors who aided me in developing/implementing this project—Dr. Tina Smilkstein and Dr. Joseph Callenes-Sloan. This project would have not been possible without their insight and guidance.

VI. Code

VI.I MBIST

```
module MEMORY_BIST #(parameter ADDRESS_WIDTH = 32, DATA_WIDTH = 32)

(start,rst,clk,rwbarin,address,datain,dataout,fail,faulty_addr,ramin,ramaddr,rwbar);

input start, rst, clk, rwbarin;
input [ADDRESS_WIDTH-1:0] address;
input [DATA_WIDTH-1:0] datain;
output [DATA_WIDTH-1:0] dataout, ramin;
output [ADDRESS_WIDTH-1:0] faulty_addr, ramaddr;
output fail, rwbar;

wire cout, eq;
wire [2:0] state;
wire [36:0] seq;
wire [DATA_WIDTH-1:0] data_t, data_l;
reg fail;
reg [ADDRESS_WIDTH-1:0] faulty_addr;
reg [31:0] seq_addr;
assign last_addr = seq[31:0]-1;


//module assignment
//1) bist controller sets flags for the whole block (houses FSM)

//2) counter is responsible for all of sequencing (ie marching)
//   that occurs in the bist. Also the counter contains the logic
//   for storing and loading the original (program) data at the
//   tested address-keeping all of the memory intact

//3) decoder is responsible for decoding the sequence and providing
//   the RAM with the correspoding data to be stored and compared

//4) comparator is responsible for setting the eq flag for when the
//   data out from the RAM matches the data generated by the BIST
//   (ie data_t)

BIST_CONTROLLER CTL
(.start(start), .rst(rst), .clk(clk), .cout(cout), .fail(fail), .state(state));
COUNTER CNT
(.clk(clk), .rst(rst), .state(state), .din(dataout), .dout(data_l), .seq(seq), .cout(cout));
DECODER DEC (.in(seq[34:32]), .out(data_t));
COMPARATOR CMP (.in1(dataout), .in2(data_t), .eq(eq));

//mux assignment
//All mux's are 4-1 to account for the 4 states of the FSM.
//Moreover the mux's work in conjunction with the module logic
//to properly carry out normal use of RAM and the added functionality
//of the BIST
```

```
MUX5_1 RWMUX
(.in1(rwbarin), .in2(1'b1), .in3(1'b0), .in4(1'b1), .in5(1'b0), .sel(state), .out(rwbar));
MUX5_1 #(32) DATAMUX (.in1(datain), .in2(data_t), .in3(32'b00000000000000000000000000000000),
.in4(data_l), .in5(32'b00000000000000000000000000000000), .sel(state), .out(ramin));
MUX5_1 #(32) ADDRMUX
(.in1(address), .in2(seq[31:0]), .in3(seq[31:0]), .in4(seq_addr), .in5(seq[31:0]), .sel(state
), .out(ramaddr));


//end condition/error detection check
//The purpose of this check is to terminate the march upon finding an
//error, set a fail flag, and report on the address of the ensuing fault

always @ (posedge clk) begin
    if (~eq && (rwbar == 1'b0) && state == 3'b011) begin
            fail <= 1'b1;
            faulty_addr <= seq[31:0];
        end else begin
            fail <= 1'b0;
        end
end

endmodule
```

VI.II Counter

```
module COUNTER #(parameter SEQUENCE_WIDTH = 35)
                (clk, rst, state, din, dout, seq, cout);

parameter testa = 2'b001,
          store = 2'b010,
          load = 2'b011,
          testb = 2'b100;

//flags for triggering the counter, reseting the counter
input clk, rst;
input [2:0] state;
input [31:0] din;
//the sequence output is used for decoding test inputs,
//test adresses, and test r/w
output [SEQUENCE_WIDTH-1:0] seq;
output [31:0] dout;
//cout acts as a flag that the counter has iterated through all sequences
output cout;

reg [31:0] dout;

//program_data is a register used to store the data at the current
//address being tested
reg [31:0] program_data;

//cnt_reg (aka counter) is responsible for keeping track of the position
//that the BIST is at in testing/sequencing the memory
```

```
reg [SEQUENCE_WIDTH:0] cnt_reg;

always @ (posedge clk) begin
    if (rst) begin
        cnt_reg <= 36'b000000000000000000000000000000000000;
    end
    else if (state == store) begin
        //read and store data at current address
        program_data <= din;
    end
    else if (state == load) begin
        //load data originally in the place of tested adress
        dout <= program_data;

        //if not reseting, than the counter increments
        cnt_reg <= cnt_reg + 1;
    end
end

assign seq = cnt_reg[SEQUENCE_WIDTH-1:0];
assign cout = cnt_reg[SEQUENCE_WIDTH];

endmodule
```

VI.III Decoder

```
module DECODER (in, out);

//input encoded test value from sequence produced by counter
input [2:0] in;
//test datain value for comparator and the RAM
output [31:0] out;

wire [31:0] out_temp;

//standard tests to check for memory cell errors
assign out_temp = (in[2:0] == 3'b 000) ? 32'b 00000000000000000000000000000000 :
                  (in[2:0] == 3'b 001) ? 32'b 00000000000000001111111111111111 :
                  (in[2:0] == 3'b 010) ? 32'b 00110011001100110011001100110011 :
                  (in[2:0] == 3'b 011) ? 32'b 01010101010101010101010101010101 :
                  (in[2:0] == 3'b 100) ? 32'b 11111111111111111111111111111111 :
                  (in[2:0] == 3'b 101) ? 32'b 11111111111111110000000000000000 :
                  (in[2:0] == 3'b 110) ? 32'b 11001100110011001100110011001100 :
                  (in[2:0] == 3'b 111) ? 32'b 10101010101010101010101010101010 :
                                         32'b zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz ;

assign out = out_temp;

endmodule
```

VI.IV Comparator

```
module COMPARATOR (in1, in2, eq);

input in1, in2;
output eq;

always @ (in1,in2) begin
    if (in1 == in2) begin
        eq = 1'b1;
    end
    else begin
        eq = 1'b0;
    end
end

endmodule
```

VI.V BIST Controller

```
module BIST_CONTROLLER (start, rst, clk, cout, fail, state);

input start, rst, clk, cout, fail;
output [2:0] state;

reg current = idle;

parameter   idle = 3'b 000,
            testa = 3'b 001,
            store = 3'b 010,
            load = 3'b 011;
            testb = 3'b 100;


always @ (posedge clk) begin
    if (rst || fail)
        current <= idle;
    else
        case (current)
            idle:   if (start && !fail) //start BIST flag
                        current <= store;
                    else
                        current <= idle;

            testa:
                    current <= testb;

            testb:
                    current <= load;

            store:
                    current <= testa;
```

```
            load:
                    if (cout)
                        current <= idle;
                    else
                        current <= store;
            default:
                        current <= reset;
        endcase
end

endmodule
```

VI.VI Otter MCU Integration (Only Parts That I Added)

```
//MBIST tie-in into RAM and rest of Otter
////////////////////////////////////////////START/////////////////////////////////////////////
    wire [31:0] ramaddr, ramdatain, faulty_addr;
    wire start, rst, rwbarin, fail;
    wire [1:0] rwsel, ramrwin;

    assign rwbarin = ~memWrite;
    assign rwsel[0] = rwbar;
    assign rwsel[1] = ~(memWrite ^ memRead2);

    DEMUX1_2 RWSEL (.sel(rwsel), .out(ramrwin));

    MEMORY_BIST MEMBST (.start(start),.rst(rst),.clk(clk),.rwbarin(rwbarin),
    .address(aluResult),.datain(B),.dataout(mem_data),.fail(fail),.faulty_addr(faulty_addr),
    .ramin(ramdatain),.ramaddr(ramaddr),.rwbar(rwbar));

    BIST_TRIGGER BSTTRG (.address(aluResult), .datain(B), .dataout(mem_data),
    .rwbar(rwbarin), .start(start));


    OTTER_mem_byte #(14) memory (.MEM_CLK(CLK),.MEM_ADDR1(pc),.MEM_ADDR2(ramaddr),
    .MEM_DIN2(ramdatain),.MEM_WRITE2(ramrwin[1]),.MEM_READ1(memRead1),.MEM_READ2(ramrwin[0]),
    .ERR(),.MEM_DOUT1(IR),.MEM_DOUT2(mem_data),.IO_IN(IOBUS_IN),.IO_WR(IOBUS_WR),
    .MEM_SIZE(IR[13:12]),.MEM_SIGN(IR[14]));
////////////////////////////////////////////END/////////////////////////////////////////////

//PC EDIT (added AND Gate)
////////////////////////////////////////////START/////////////////////////////////////////////
    wire pcWrite_BST;
    assign pcWrite_BST = pcWrite & ~start;

    OTTER_CU_FSM CU_FSM (.CU_CLK(CLK), .CU_INT(INTR), .CU_RESET(RESET),
    .CU_OPCODE(opcode), //.CU_OPCODE(opcode),
    .CU_FUNC3(IR[14:12]),.CU_FUNC12(IR[31:20]),
    .CU_PCWRITE(pcWrite_BST), .CU_REGWRITE(regWrite), .CU_MEMWRITE(memWrite),
    .CU_MEMREAD1(memRead1),.CU_MEMREAD2(memRead2),.CU_intTaken(intTaken),.CU_intCLR(intCLR),
    .CU_csrWrite(csrWrite),.CU_prevINT(prev_INT));
////////////////////////////////////////////END/////////////////////////////////////////////
```

VI.VII BIST Trigger

```
module BIST_TRIGGER #(parameter ADDRESS_WIDTH = 31, DATA_WIDTH = 31)
                     (address, datain, dataout, rwbar, start);

input rwbar;
input [1:0] state;
input [ADDRESS_WIDTH-1:0] address;
input [DATA_WIDTH-1:0] datain, dataout;
output start;

reg [ADDRESS_WIDTH-1:0] last_addr;
reg saved_parity;

//BIST trigger is meant to activate the march test of the memory
//if there is a discrepancy detected between the parity of the
//stored data and the data being read from the memory. The process
//is best described below:
/*

                read or write
                /            \
               /              \
1) check if addr is same        1) save the addr currently being
as stored addr                  written to
2) check if parities of         2) store the parity of the
dataout and the stored parity   written data
match

*/
//This type of trigger benefits from minimizing the space needed to
//save data (instead of saving all bits of data, parity is stored instead).
//Moreover, this trigger takes advantage of the temporality of address calls
//in memory (ie addresses which are written to are likley to be called
//soon after -- eg. branching to isr/subroutine). The primary fault of this
//trigger method is that it will not pick up on 100% of memory faults since
//there can be errors that result in the same parity (eg. 10111 != 11011,
//however they have the same parity).

always @ (posedge clk) begin
    if (start == 1'b0)
        if ((rwbar == 1'b1) && (address != last_addr)) begin
            last_addr <= address;
            saved_parity <= ^datain;
        end else if ((rwbar == 1'b0) && (address == last_addr))
            if (saved_parity != ^dataout) begin
                start <= 1'b1;
            end else begin
                start <= 1'b0;
            end
    else begin
        start <= 1'b0;
```

```
    end
end

endmodule
```

VI.VIII MUX 5-1

```
module MUX5_1 #(parameter DATA_WIDTH = 2)
              (in1, in2, in3, in4, in5, sel, out);

//default input(s) and output sizes are a single bit
input [DATA_WIDTH-1:0] in1;
input [DATA_WIDTH-1:0] in2;
input [DATA_WIDTH-1:0] in3;
input [DATA_WIDTH-1:0] in4;
input [DATA_WIDTH-1:0] in5;
input [2:0] sel;
output [DATA_WIDTH-1:0] out;

assign out =    (sel == 3'b000) ? in1:
                (sel == 3'b001) ? in2:
                (sel == 3'b010) ? in3:
                (sel == 3'b011) ? in4: in5;



endmodule
```

VI.IX DEMUX 2-1

```
module DEMUX1_2 (parameter DATA_WIDTH = 2)
                (sel, out);

input [1:0] sel;
output [DATA_WIDTH-1:0] out;

reg [DATA_WIDTH-1:0] out;

always @(in, sel) begin
    case (sel)
        2'b00:  begin
                    out[0] = 1'b1;
                    out[1] = 1'b0;
                end

        2'b01:  begin
                    out[0] = 1'b0;
                    out[1] = 1'b1;
                end

        2'b10:  begin
                    out[0] = 1'b0;
```

```
                    out[1] = 1'b0;
                end

        2'b11:  begin
                    out[0] = 1'b0;
                    out[1] = 1'b0;
                end

        default:    out = 2'b00;
    endcase
end

endmodule
```

VII. References

[1] Z. Navabi, Digital System Test and Testable Design. Worcester, MA: Springer Science+Business Media, LLC, 2011.

[2] L. Wang, C. Wu and X. Wen, VLSI Test Principles and Architectures. Elsevier, 2006.

[3] M. Bushnell and V. Agrawal, Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits. Kluwer Academic Publishers, 2000.

[4] J. Callenes-Sloan, OTTER (RV32I) Assembly Instructions CPE 233. p. 3. "Figure 1: Memory Allocations". 2020.

[5] J. Mealy, RISC-V OTTER MCU Architecture Diagram V2.01. 2020.