UNIVERSITY OF CALIFORNIA
SANTA CRUZ

BUILT-IN SELF-REPAIR FOR OPENRAM MEMORIES

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Aditi Sinha

September 2020

The Thesis of Aditi Sinha
is approved:

Professor Matthew Guthaus, Chair

Professor Jose Renau

Assistant Professor Scott Beamer

Quentin Williams
Interim Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

Abstract

Built-in Self-repair for OpenRAM Memories

by

Aditi Sinha

Incorporating self-repair capabilities to memories is a standard practice to reduce yield loss from hardware defects. OpenRAM is an open-source memory compilation framework that supports the automated generation of Static Random Access Memories (SRAMs). This thesis extends the OpenRAM memory compiler with a built-in self-repair (BISR) feature. The self-repair logic is implemented as a synthesizable Verilog wrapper, and the OpenRAM SRAM is augmented to include extra rows and columns for remapping faulty cells.

Acknowledgments

I would like to thank my teachers for their guidance and feedback. I also thank my family for their constant support and encouragement.

Chapter 1

Introduction

Embedded memories are an integral part of system-on-chip (SOC) architectures. Modern applications rely heavily on memory, which necessitates the design of high-density memories with fast access speeds. However, as memories get dense, physical defects occur more frequently. At worst, these defects can completely inhibit the functioning of a chip, and at best, lead to logical faults that deteriorate performance. Hence, identifying and possibly correcting such faults is crucial for SOC design.

Design for testability (DFT) techniques focus on making circuits that are easily testable after manufacture. Testability is not considered separate from design but is integrated into design flows. For a target node or signal line to be testable, it must be externally controllable and observable [1]. This means the node should be able to get driven to a value through external inputs, and its value should also be observable externally. Both of these ensure the node can be sensitized from the primary inputs to check for faults, and the effect of any such fault can be propagated to the primary outputs. Thus, the goal of the DFT approach is to make circuits controllable and observable.

However, controlling and observing memories is challenging since high memory densities hinder testing from the I/O pins on a chip. The built-in self-test (BIST) approach offers a solution to this problem. In BIST, test logic is implemented on-chip, so that memories can test themselves for manufacturing failures without the external intervention of automatic test equipments (ATEs).

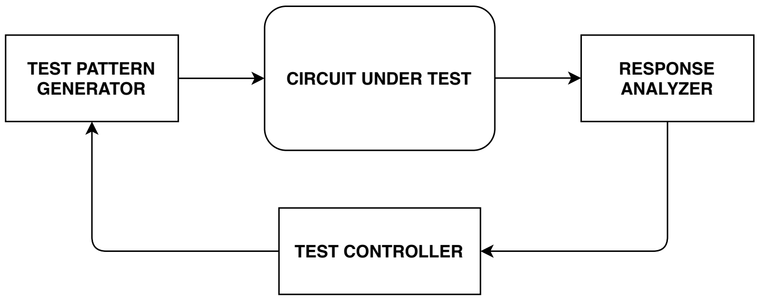


Figure 1.1: Testing a Circuit

Figure 1.1 describes a general memory test procedure, where test stimulus is given to the circuit under test in the form of test patterns. Responses are read back from memory and analyzed to locate faults. On top of this, repair logic can also be added to repair such faults. Once BIST identifies faulty memory cells, they can be remapped to healthy redundant cells through built-in redundancy allocation (BIRA). Together, the BIST and the BIRA form the overall built-in self-repair (BISR) logic. This thesis adds the feature of detecting and correcting memory faults to the OpenRAM memory compiler.

OpenRAM [2] is an open-source memory compiler that automates the generation and characterization of SRAM memories. It supports the generation of layouts, netlists, and Verilog modules for multi-ported SRAMs [3]. The supported process technologies are NCSU's FreePDK45 (10 metal layers) [4], MOSIS's SCMOS 0.35 µm (4 metal layers) [5] and SkyWater's 130nm node (5 metal layers) [6].

For supporting BISR, we add extra bitcells to the SRAM array as the redundancy. A Verilog wrapper, synthesizable with standard cells, is used for implementing the BISR logic. The wrapper tests and repairs memory on boot-up (i.e., reset). The system then accesses the memory via the wrapper, as shown in Figure 1.2. The wrapper also hides the use of redundancy from the system.

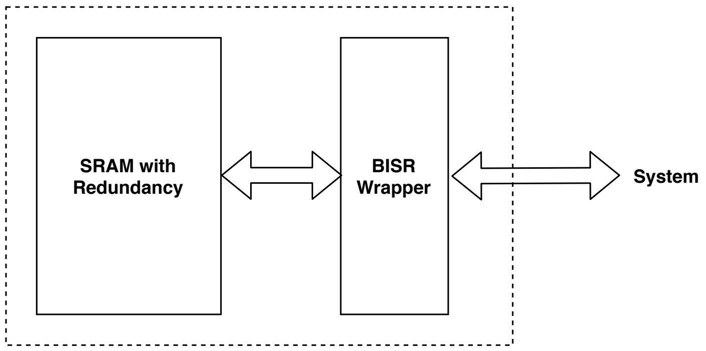


Figure 1.2: BISR Top Module

This thesis is organized as follows. Chapter 2 introduces OpenRAM and describes the implementation of redundancy. In chapter 3, we discuss the design of BIST wrapper. Chapters 4 covers the redundancy allocation BIRA wrapper and its integration with BIST. Chapter 5 concludes with the results and summary of this work.

All contributions to OpenRAM are available on GitHub at:

`https://github.com/VLSIDA/OpenRAM`

Chapter 2

Adding Redundancy

The addition of redundant memory cells to the SRAM is the first step towards built-in self-repair. Healthy spare bitcells will store the data of faulty bitcells. These spares are added to memory in two variants - spare columns and spare rows.

In this chapter, we start with a background on the OpenRAM SRAM and its components. Next, we cover redundancy implementation within SRAM banks and describe how the redundant cells are read or written.

2.1 Background on OpenRAM

Figure 2.1 shows the OpenRAM SRAM architecture [3]. An SRAM is composed of the bitcell array, port address and port data blocks, and additional control logic. The number of words, word size, type of port, write size, and other input parameters can be specified in a configuration file when running OpenRAM. OpenRAM requires a minimum of four metal layer processes for generating layouts; two metal layers for bitcells, and two layers for power and SRAM-level routing.

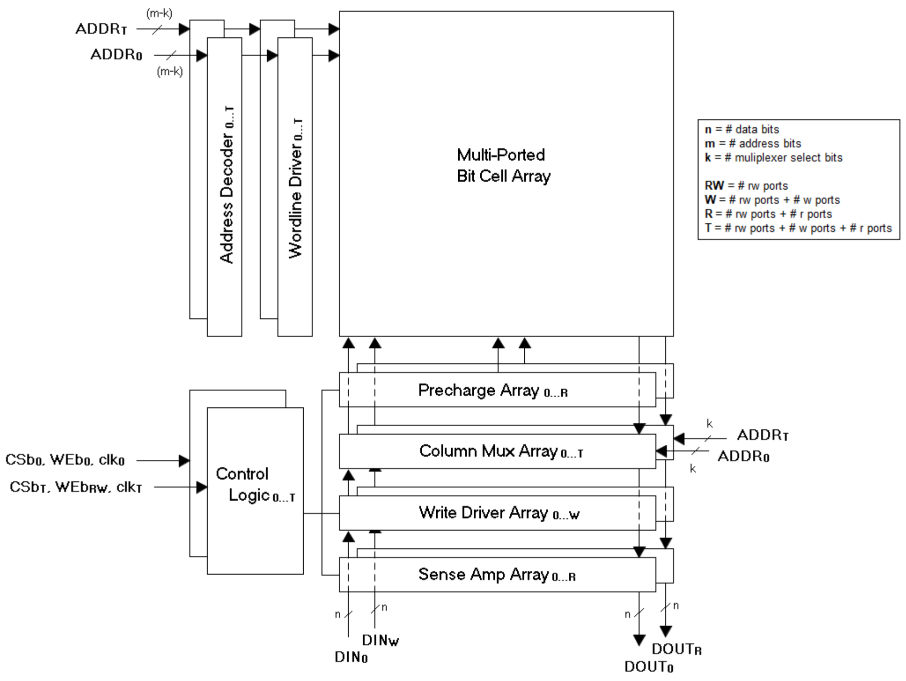


Figure 2.1: Multiported SRAM Architecture

Custom 6T SRAM cells form the bitcell array. The port address and port data modules are responsible for accessing words in the bitcell array. The port address block consists of a hierarchical row decoder, feeding into the wordline driver for driving the selected row. The port data block performs reads and writes using sense amplifiers and write driver arrays. Each port has a port address and port data module. Additionally, address and data input flip-flops give address and data inputs to the port address and port data modules.

2.1.1 Bitcell Array

The bitcell array is appended with columns and dummy bitcells to make a replica bitcell array, shown in Figure 2.2. Replica columns (aqua) help in driving enables to the sense amplifiers for reads, so that read timings are consistent. The dummy cells (red) ensure the layout meets lithography requirements.

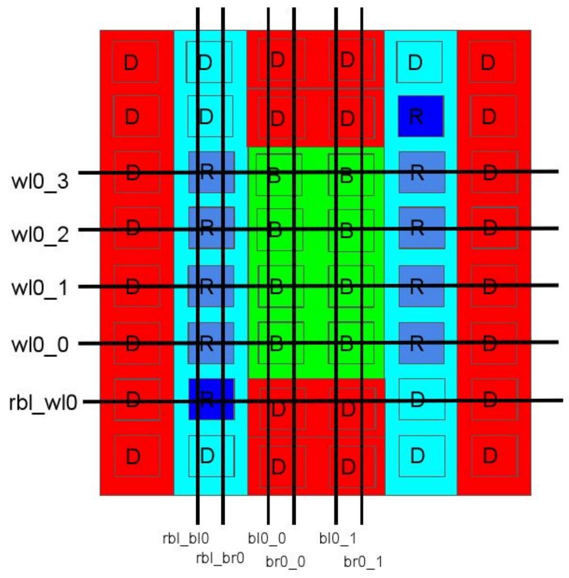


Figure 2.2: Replica Bitcell Array

2.1.2 Port Address

The port address module in Figure 2.3 has a hierarchical address decoder and a wordline driver, and is generated for every port. The row address decoder takes row address bits from the address input and selects the appropriate row. Multiple predecoders form the row decoder; hence it is hierarchical. The wordline driver enables access to the words on the selected row by driving the word lines that gate individual transistors on that row.

2.1.3 Port Data

The port data module in Figure 2.4 consists of the precharge array, sense amplifier array, write driver array, and the column multiplexer array.

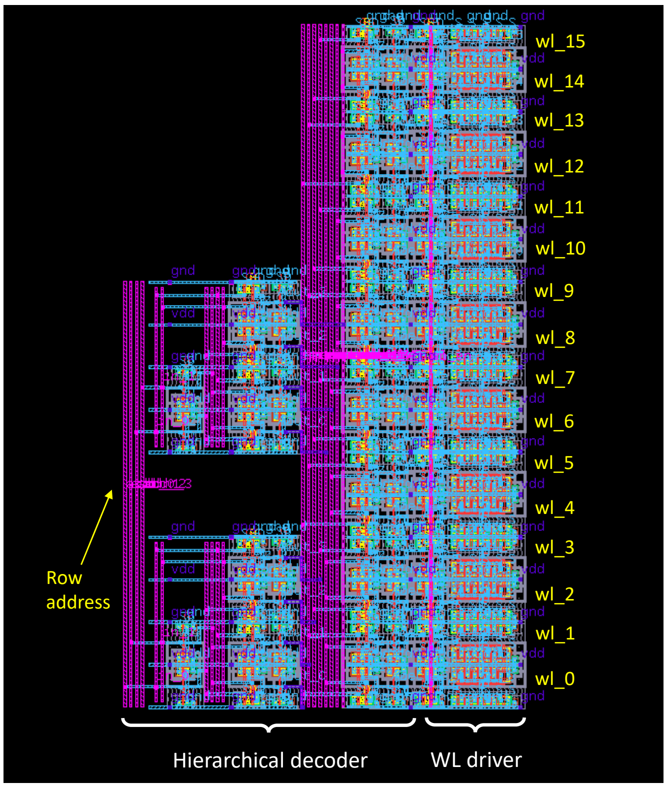


Figure 2.3: Port Address Layout for 16 rows in the SCMOS process node

The precharge array drives a "1" onto the bitlines to precharge them for reads. The sense amplifiers sense the difference in bitline voltages for reads, while the write drivers write to words. The length of these two arrays is equal to the word size. The column multiplexer selects the appropriate word using the lower address bits when there are multiple words per row. If there is only one word per row, the column multiplexer is not needed.

2.1.4 Control Logic

The control logic generates internal control signals such as a buffered clock (*clk_buf*) and the enable signals for port data. An internal write enable signal

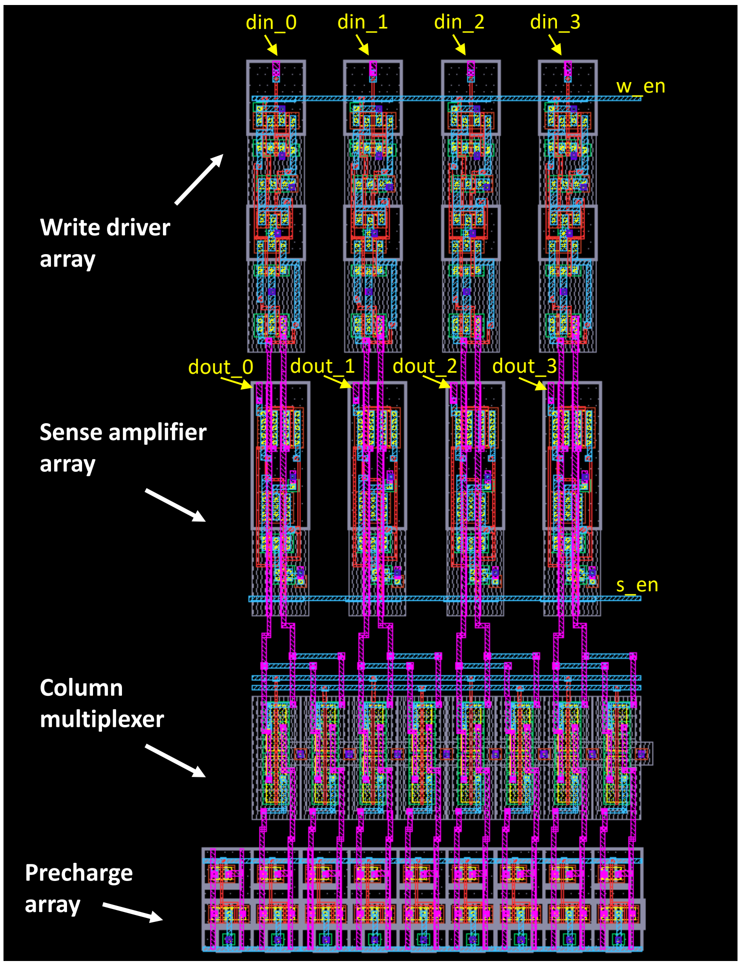


Figure 2.4: Read-write Data Port Layout for 4-bit word SRAM in the SCMOS process node

(w_en) is asserted to enable the write drivers when external chip select (*csb*) and write enable (*web*) signals are active low. When *web* is high, an active high sense amplifier enable (*s_en*) allows the sense amplifiers to read the selected word. Input data and external signals are captured and stored in flip flops when the external clock (*clk*) is high, while reads/writes to bitcells occur when the clock goes low.

2.2 Redundancy Overview

Redundancy is implemented in OpenRAM by adding spare cells in the form of extra rows and extra columns to the bitcell array. The input parameters for enabling redundancy are the number of spare rows (*num_spare_rows*) and the number of spare columns (*num_spare_cols*). Unless specified in the configuration file, their values default to zero, and the functionality of OpenRAM remains unchanged. Adding redundancy to the regular bitcell array itself instead of implementing it as a separate memory block, avoids incurring excess area and power overheads. In our design, bitcells in the spare columns are meant to replace the faulty bits of a word. If too many words are faulty in a row, it is replaced with a spare row. This is explained in detail in Section 4.1.

Figure 2.5 shows how the spare rows are added to the top of the bitcell array, while spare columns are added on the right. The spare bitcells common to the spare columns and rows (in yellow) are not used.

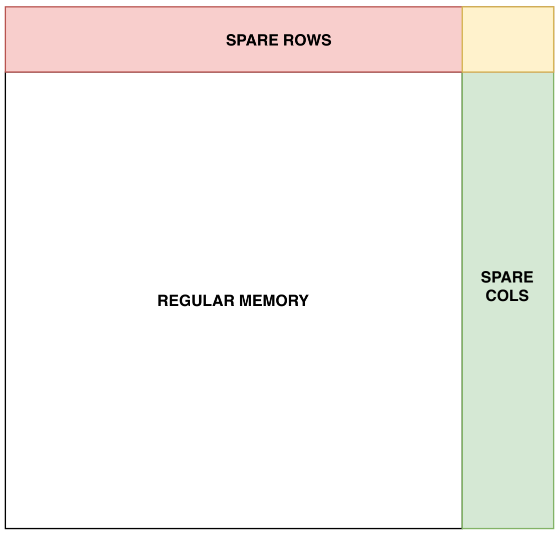


Figure 2.5: Adding Redundancy to Bitcell Array

2.3 Spare Columns

Implementing spare columns in the bitcell array is necessary for bit-level granularity in repairs, say when only a single bit is faulty in an entire row which needs to be remapped to a healthy bit. We add functionality for individual writes to spare columns. Peripheral circuitry, such as port data and control logic, is modified to support read and write operations to spare columns. The major steps are – creating the spice netlist, creating the layout, performing functional testing, and changing the Verilog model of the SRAM.

2.3.1 Netlist

The spice netlist for every module is generated by instantiating and connecting sub-modules, adding pins at the current hierarchy level, and computing some constants like the number of total columns, bus sizes, etc.

The first step is to create and connect more columns in the bitcell array. The wordline driver also needs the new total number of columns input so that it drives an entire selected row, including the spare column bitcells belonging to that row. The port data module and its sub-modules are changed significantly.

2.3.1.1 Write Driver Array

A word-sized array of write drivers is used to give data inputs to the SRAM. For writing to spare columns, individual write drivers are instantiated and connected, one for every spare column. Every spare driver has its own unique enable input. At this level, enable pins for spares are named *en_1*, *en_2* and so on. The first enable pin, *en_0*, is the single enable for all write drivers of the regular word. This numbering changes if OpenRAM's write masking feature [7] is enabled.

2.3.1.2 Sense Amplifier Array

The word-sized array of sense amplifiers is enabled during read operations. Spare sense amplifiers are instantiated similar to spare write drivers, i.e., one for every extra column. In this case, the difference is that spare-sense amplifier cells will share the same enable as regular ones. Hence, all spare column bits are also read along with the regular word and can be ignored if not required. The control logic is also modified so that the *s_en* enable driver has enough strength to drive all amplifiers, including the extra ones.

2.3.1.3 Port Data

In the port data module, extra precharge cells are added for spare columns. The spare precharge cells' bitlines are named *sparebl_*{} and *sparebr_*{}, to distinguish them from regular ones. This lets us connect these bitlines to those of the spare write drivers or sense amplifiers without name conflicts. The spare column bitlines are not connected to the column multiplexer since they are individually enabled. The spare write enable pins are renamed to *bank_spare_wen*{}.

2.3.1.4 SRAM

At the SRAM-level, the topmost level in the netlist hierarchy, D flip-flops are added for the spare write enable inputs and extra data inputs. They are supplied a buffered clock (*clk_buf*).

2.3.2 Layout

The SRAM layout is generated by placing and connecting modules, routing the supplies and data lines, and adding the required pins. DRC checks ensure no design rule errors are present, and LVS checks confirm that the extracted netlist matches the schematic.

2.3.2.1 Write Driver Array

Figure 2.6 shows a sample layout for write driver array with spare columns enabled. The spare write drivers are placed such that they are vertically aligned with the precharges of spare columns. So, if there is a single word per row, as in the figure, the spacing will continue as before. If there are multiple words per row, spare column write drivers will be farther away from the regular drivers.

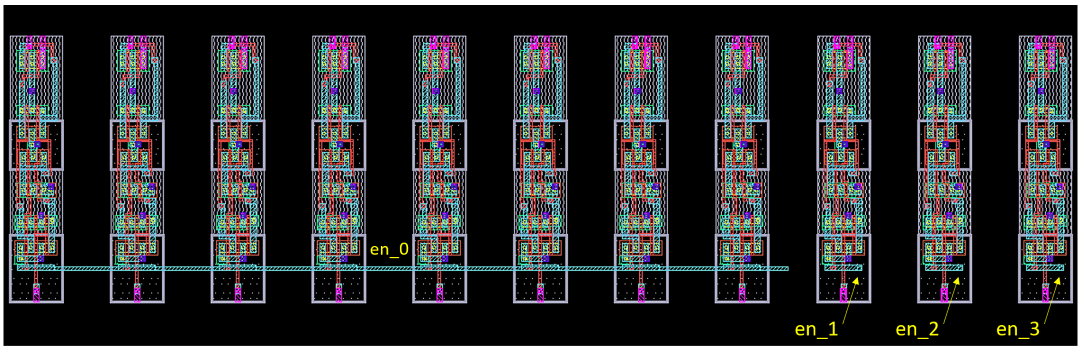


Figure 2.6: SCMOS layout of write driver array for 8b x 16 SRAM with 3 spare columns and 1 word per row

The *metal1* rail (in aqua) of the regular word write-enable *en_0* is reduced in length to avoid shorting with the spare drivers. Individual enable pins in the spares have the smallest width possible, i.e. the minimum width specified by the process node. This avoids via spacing errors at SRAM level. The enables are positioned on the lower right of the write driver cell, as pointed by the yellow arrows in the figure.

2.3.2.2 Sense Amplifier

In the layout for sense amplifier array, as shown in Figure 2.7, we add extra sense amplifiers, one for every spare column. All sense amplifiers are supplied the enable through a common enable (*en*) input. The spacing between sense amplifiers is similar to that in the write driver array. The address and data remapper, explained in Section 4.3, is responsible for either ignoring or using the spare column outputs on every read. Their data is ignored when reading from a fault-free cell.

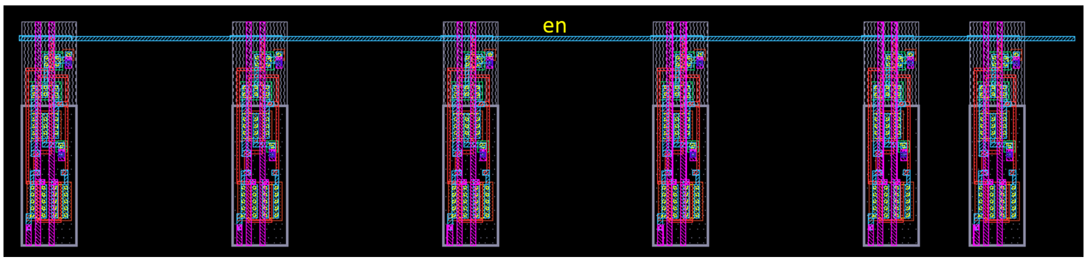


Figure 2.7: SCMOS layout of sense amplifier array for 4b x 16 SRAM with 2 spare columns (rightmost)

2.3.2.3 Port Data

Bitlines are channel routed in the port data module from the precharge array to the write driver and/or sense amplifier arrays. The column multiplexer is not involved in the spare columns.

Figure 2.8 shows the layout of port data module for a write port. The first enable pin of the write driver, *en_0*, is renamed to *w_en* at this level to be consistent with the naming in the absence of spare columns.

Figure 2.9 shows the layout of port data module for a read-only port, with an eight-way column multiplexer, a 2-bit word and 3 spare columns. As shown, spare bitlines are directly connected from the precharge array to the sense

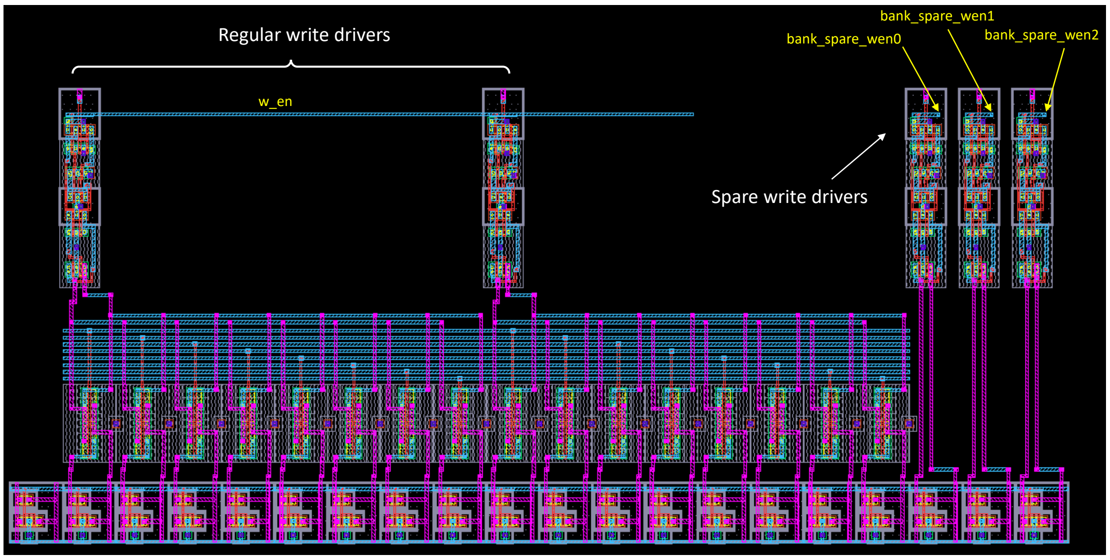


Figure 2.8: SCMOS layout of Port Data for W-port, with 3 spare columns and 2-bit words

amplifier array. Since this is a read port, a single *s_en* metal1 rail stretches across all cells.

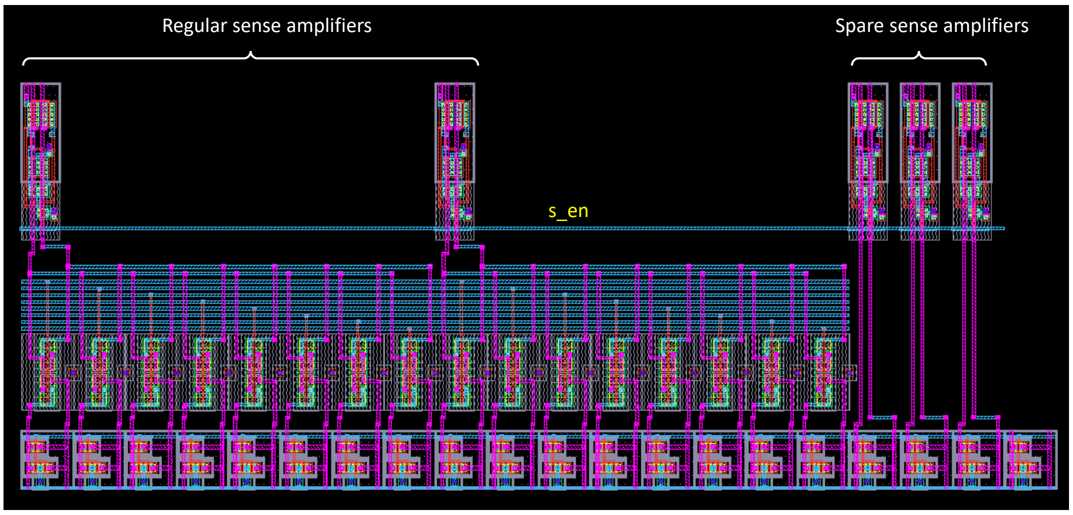


Figure 2.9: SCMOS layout of Port Data for R-port, with 3 spare columns and 8 words per row

2.3.2.4 Bank

At the bank level, the spare write enable pins are renamed to *bank_spare_wen*{}_{}, where the numbering on the left indicates the port number. At this level, the internal enable inputs are provided by connecting to vertical *metal2* rails. This is shown in Figure 2.10. These signals are generated by the control logic, which will be instantiated and connected at the SRAM level.

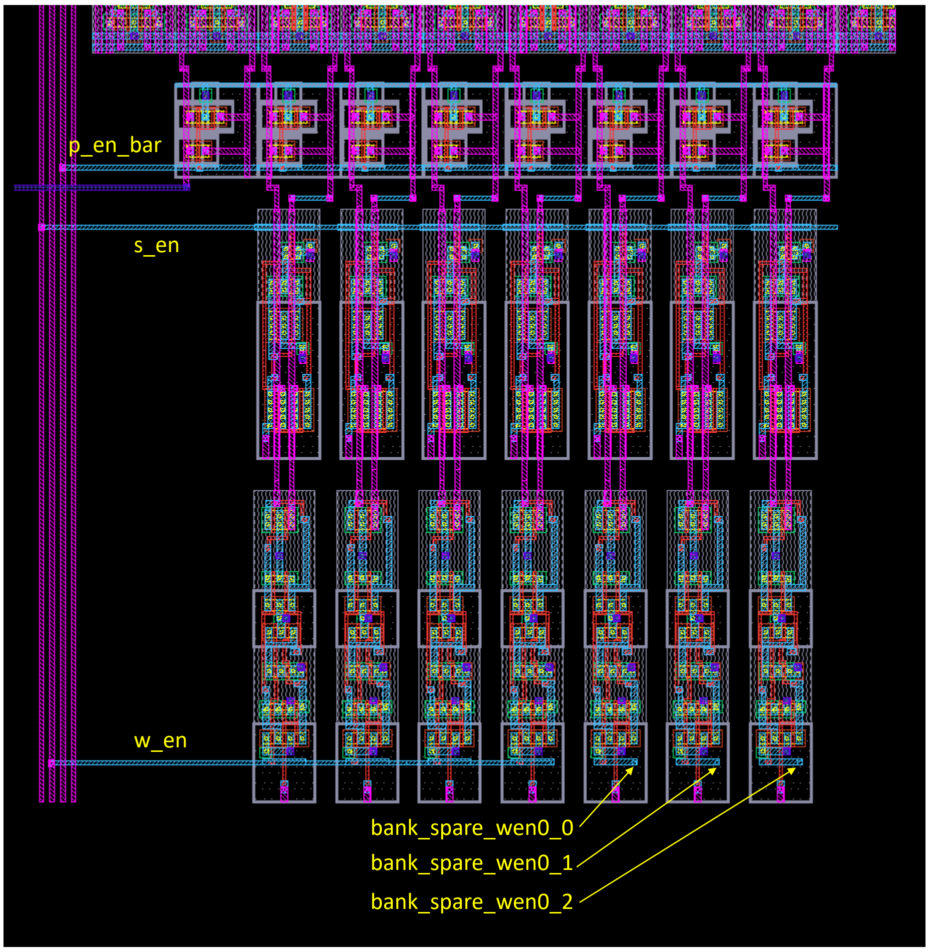


Figure 2.10: SCMOS layout of the bottom part of a bank, showing port data for port 0 (a RW port), with 3 spare columns, word size of 4 and 1 word per row

2.3.2.5 SRAM

External inputs to the SRAM are provided using D flip-flops, as shown in Figure 2.11. The spare write enable inputs are channel routed to the spare enables of the bank pins in *metal1* (in aqua) and *metal2* (in pink). The D flip-flops are placed such that there is enough space for the jogs in these routes. The data inputs are channel routed to the *din_*{} pins in *metal3* (in purple) and *metal2* (in pale yellow). If there are no spare columns, the data DFFs are routed in the *metal1* layer stack. All flip-flops are supplied a buffered clock by the control logic.

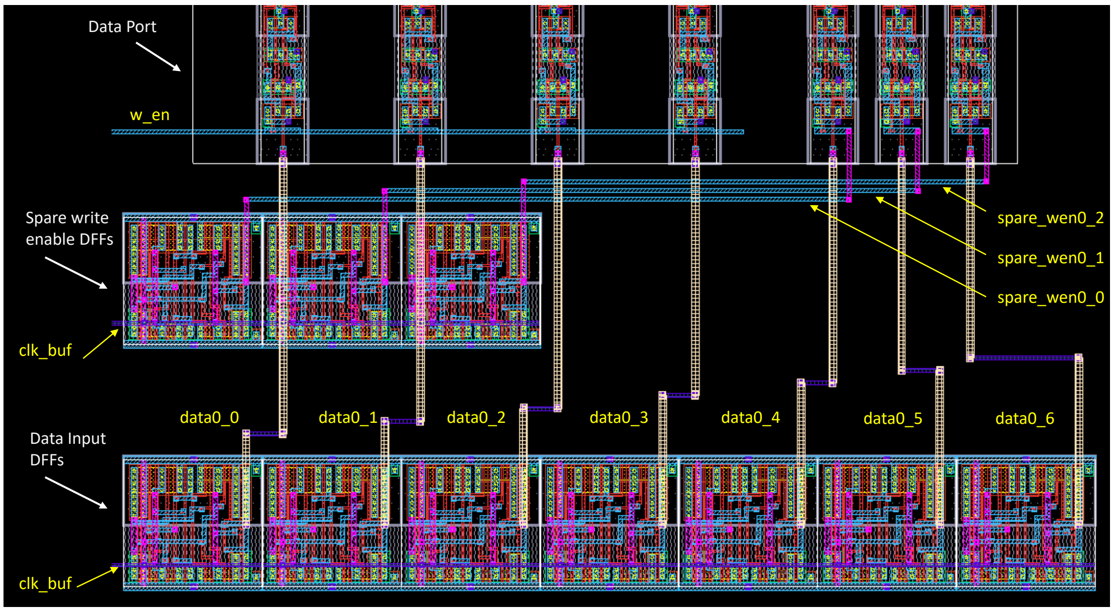


Figure 2.11: SCMOS Layout of the bottom part of an SRAM with 3 spare columns, showing how inputs are channel routed to the bank pins

Figure 2.12 shows the full SRAM layout with three spare columns and supply routing turned off for clarity. If it is enabled, a supply grid routed in *metal3* and *metal4* connects all the *vdd* and *gnd* pins in the sub-parts of the SRAM. This routing can be disabled in the configuration file if required.

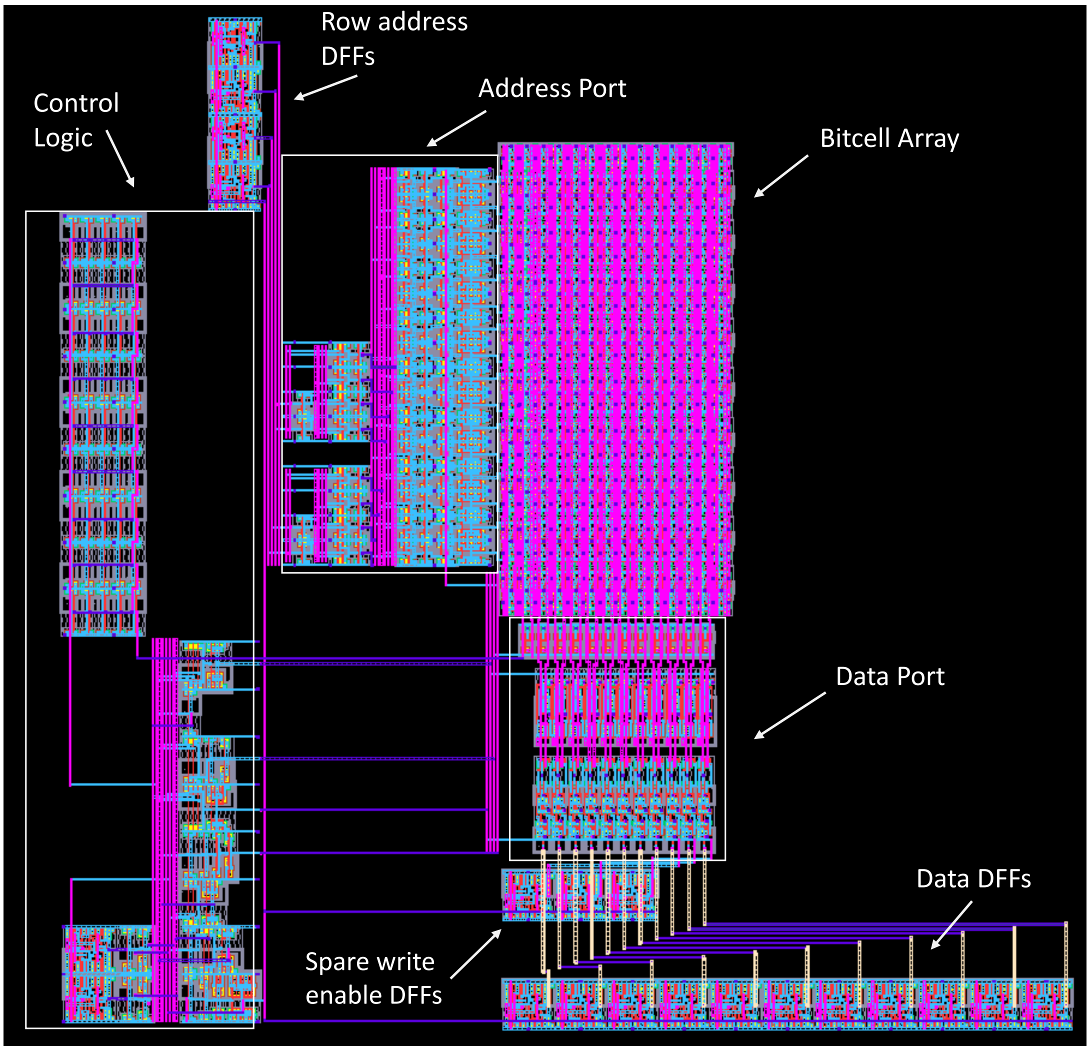


Figure 2.12: SCMOS Layout of an SRAM with 8-bit words, 3 spare columns and no column multiplexer

2.3.3 Functional Tests

If the bits at an address have been mapped to a spare column, the spare columns will also be written to when a word is written at that address. When we write a word in a row, the spare columns of the same row will also be written, if enabled. This is verified with functional tests that write random data values at random addresses.

2.3.4 Verilog

The behavioral Verilog model of the OpenRAM SRAM is modified to support spare writes. The parameters *NUM_SPARE_COLS* and *COL_ADDR_WIDTH* are used to define a separate spare memory array. The depth and width of this array are the same as the number of rows and spare columns, respectively.

```
 1  reg [DATA_WIDTH-NUM_SPARE_COLS-1:0] mem [0:RAM_DEPTH-1];
 2  reg [NUM_SPARE_COLS-1:0] spare_mem [0:SPARE_RAM_DEPTH-1];
 3   always @ (negedge clk0)
 4   begin : MEM_WRITE
 5    if (!csb0_reg && !web0_reg )
 6     begin
 7      mem[addr0_reg]
 8            = din0_reg[DATA_WIDTH-1:NUM_SPARE_COLS];
 9      if (spare_web0_reg[0])
10       begin
11        spare_mem[addr0_reg[ADDR_WIDTH-1:COL_ADDR_WIDTH]][0]
12                  = din0_reg[0];
13       end
14     end
15   end
```

Figure 2.13: Verilog code showing writes to spare memory

An example is shown in Figure 2.13. The spare memory array is addressed using the regular word's row address (line 11) and the spare write enables (line 9). The chip select and write enable inputs to the SRAM are active low for spare writes (line 5), since they occur with the regular word writes. Lower bits of the data input are reserved as inputs for spare columns (line 12), while the rest are for the regular word (lines 7 and 8).

2.4 Spare Rows

A spare row replaces an entire regular row. Spare rows must be accessible using the address input to the SRAM, so an extra bit is added to the row address. When regular words are accessed, this bit will be a “0”. Although this means that the memory size can be doubled, we need to add just a few extra rows to the bitcell array. Beyond the last word of the last spare row, the remaining addresses that can be generated with the extra bit are not usable.

2.4.1 Netlist

Extra rows are added to the bitcell array by changing the total number of rows. The port address block is also modified so that it can select the spare rows and drive their wordlines. The words in spare rows will be accessible in the same manner as regular words, so the port data block remains unchanged.

2.4.2 Layout

2.4.2.1 Replica Bitcell Array

The bitcell array is generated by creating a 2D array of bitcells, where every other row is flipped vertically. Dummy arrays are instantiated and rotated or mirrored to form dummy columns/ row-end caps on all four sides of the bitcell array. Though these dummies have their bitlines disconnected, they still need to be aligned and offset with the surrounding cells to avoid DRC issues.

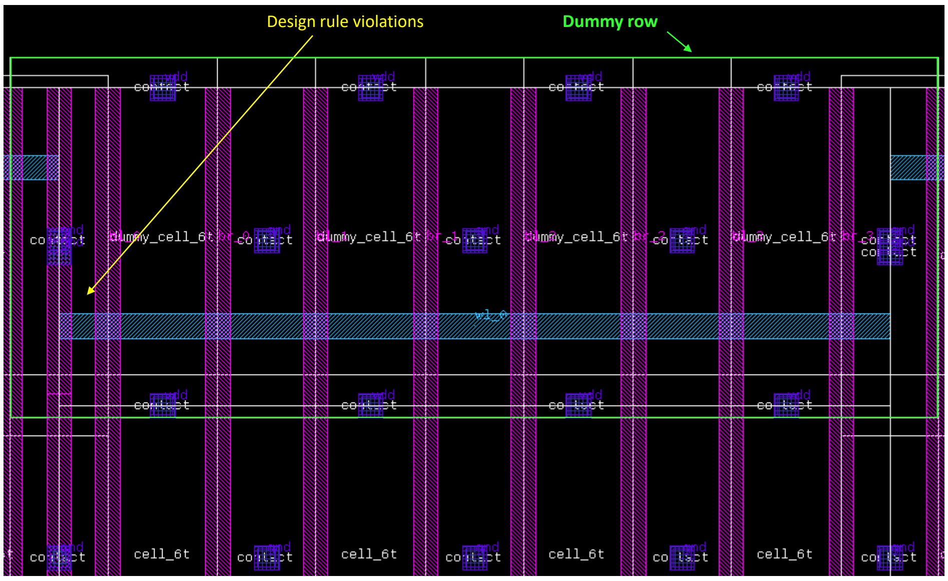


Figure 2.14: Layout of the top portion of replica bitcell array with 7 rows, where the top dummy row is not flipped, causing DRC violations

When spare rows are enabled, extra bitcell rows are added to the top of the array. Since the number of spare rows can be arbitrary, the total number of rows can be odd instead of even, as they were previously. In this case, we see DRC errors, as depicted in Figure 2.14. The dummy array boundaries clash with those of regular 6T cells, and wordlines are broken. This is because, previously, the number of rows has always been even, and so the top dummy array was not flipped for aligning with the bitcell array.

With odd rows, this dummy row needs to be flipped vertically and offset to match the top bitcell row. Then, the layout comes out DRC clean, as in Figure 2.15.

2.4.2.2 Wordline Driver

The wordline driver array is a NAND and INV array to drive all wordlines. It is generated to match the total height of the bitcell, including the spare rows.

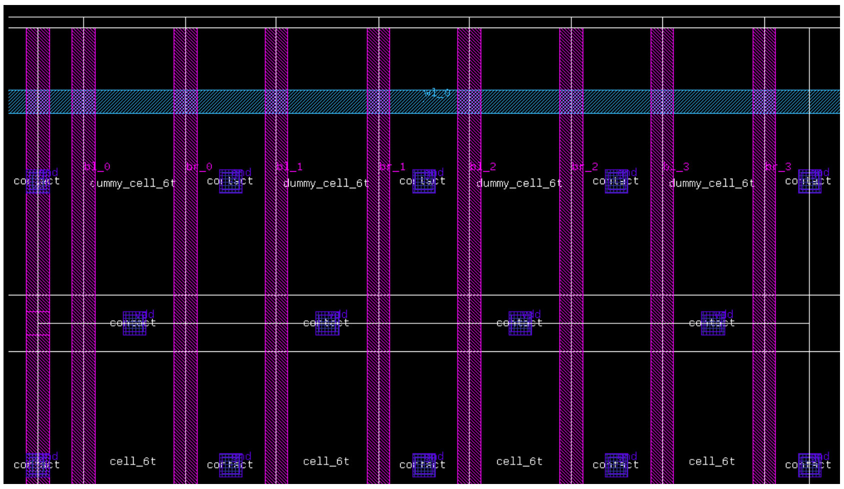


Figure 2.15: DRC clean layout of the top portion of replica bitcell array with 7 rows

2.4.2.3 Hierarchical Row Decoder

The row address decoder is composed of predecoders feeding into AND logic. The inputs to the address decoder are given by vertical *metal2* rails, which connect to row address flip-flops at the SRAM-level (see Figure 2.12). Combinations of two kinds of predecoders – a 3-to-8 predecoder and a 2-to-4 predecoder, are used for the first stage of the decode. An array of AND gates does the final decode to select a wordline.

The changes done to the hierarchical decoder for supporting spare rows can be explained with an example. Figure 2.16 shows a hierarchical decoder for 19 rows, which means 16 regular rows and 3 spare rows. This implies a minimum of 5-bit row address input, which can generate 32 unique row addresses using combinations of the 3-to-8 and 2-to-4 predecode outputs. However, since we need only 19 of them, the AND array is pruned to *decode_18*. This way, by pruning the decoder to an upper limit, we can decode addresses for non-power-of-two rows.

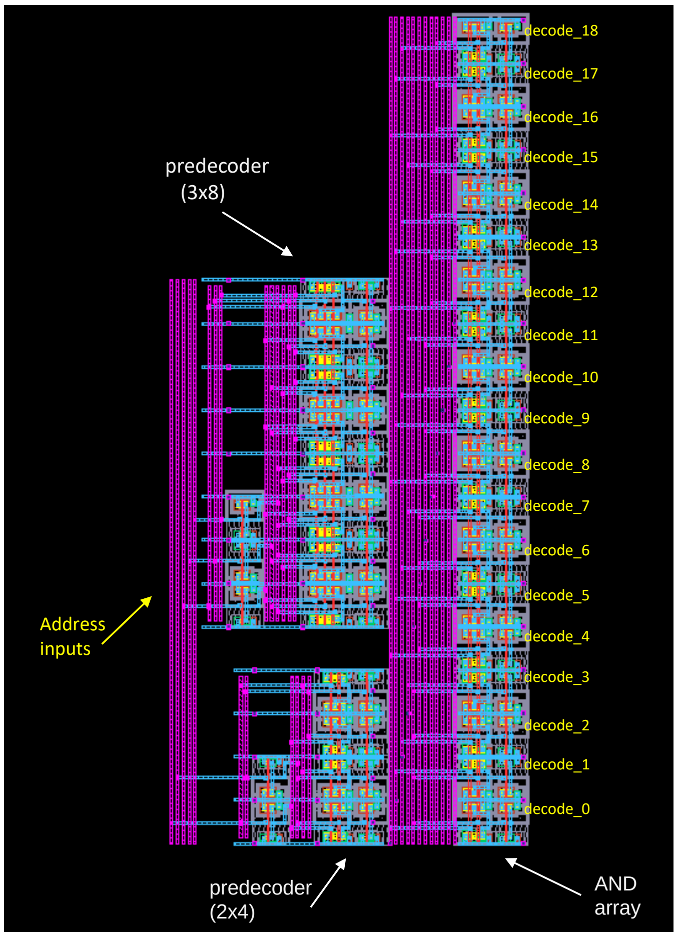


Figure 2.16: SCMOS layout of hierarchical decoder for 19 rows and 5-bit row address width

2.4.3 Verilog

Only the repair wrapper must have access to the spare rows. An upper limit of the acceptable address is set in the top module (*SRAM_BISR*, explained in Section 3.1) so that the extra rows are not directly accessible to the system. Essentially, the most significant bit of the row address must be a "0" in any address inputs coming from the system. If a row is to be remapped to a spare, the BISR wrapper supplies the required row address to the SRAM. This is explained further in Chapter 4.

Chapter 3

Self-Test

Built-in self-test (BIST) techniques are a standard industrial practice, extensively used for memory tests since they can perform checks on the fly, not just after manufacture. For OpenRAM, this is implemented as a synthesizable Verilog test collar around the memory. It requires only the power-on reset and clock signals from the system. The BIST module checks for functional faults in the memory array and sends diagnostic information to the redundancy allocator (BIRA) on boot-up. This chapter covers an overview of the BISR architecture, memory fault models, the test algorithm, and the state machine implemented for BIST.

3.1 BISR Overview

Figure 3.1 shows the repair schema and its sub-modules. The test/repair collar works in two modes of operation – the test mode, where *mode* signal is high, and the normal mode, after the repair has finished and the system wants to use the memory. The top module is named *SRAM_BISR* and instantiates the modified SRAM remapper, BIST, and BIRA controllers. Signals to and from the system are the same as when BISR was not used previously. The BIST controller has access to the memory in the test mode, while Address and Data Remapper has access in the normal mode.

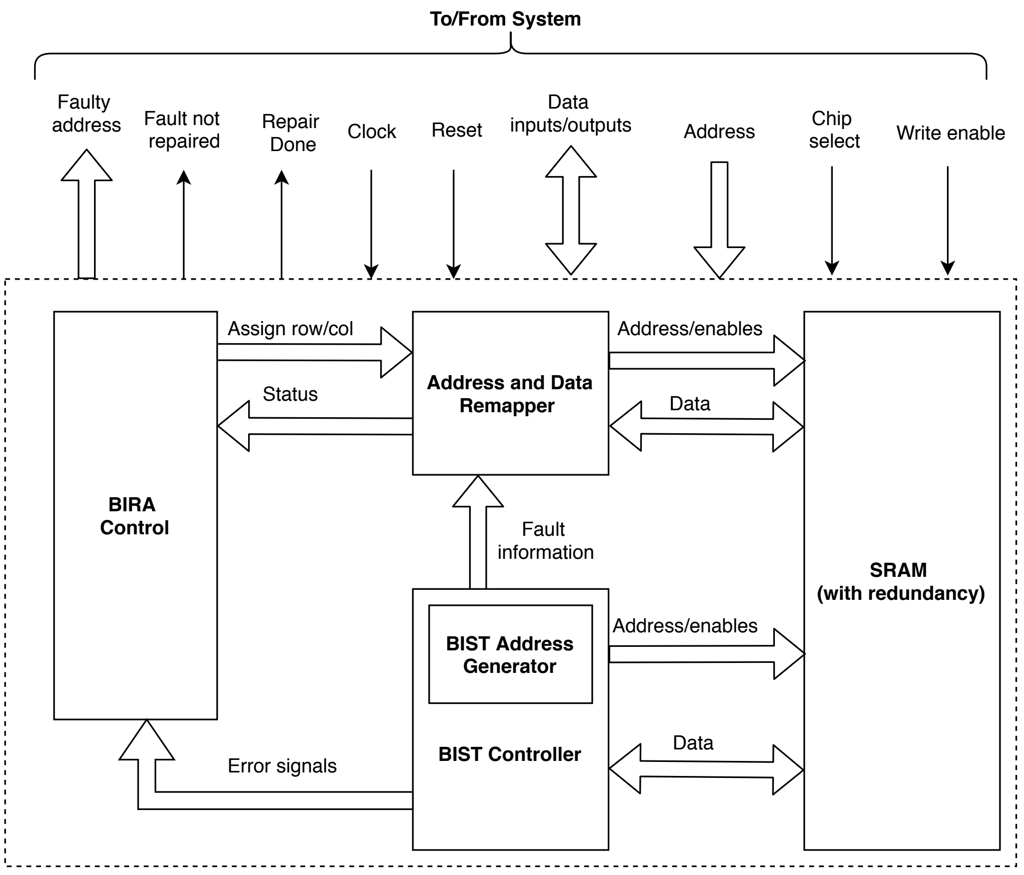


Figure 3.1: BISR Architecture

On reset, the BIST controller starts testing the memory array and gives fault information to the BIRA controller and the remapper. The BIRA controller monitors the remapper's status and accordingly instructs it to store the data in redundant rows/cols. This re-allocation occurs in the test mode itself, so when the entire test sequence is complete, the repair is finished too. If a fault is not reparable, its information is exported using the *faulty address* and *fault not* *repaired* signals to the system. BIRA Controller informs the system about repair end and switches to normal mode. In normal mode, the remapper compares incoming addresses from the system to the ones stored and issues appropriate signals to the SRAM, and also sends the corrected outputs to the system.

3.2 RAM Fault Models

Many physical deformities have logical effects. By modeling these effects as faults, test algorithms are used to verify the operations of circuits. Here, we focus on three memory fault models - stuck-at, transition, and stuck-open, to classify the logical behavior of such defects. Figure 3.2 a. shows the state transitions of a healthy memory cell. The states correspond to the values stored in the cells, while the "w"s are write operations.

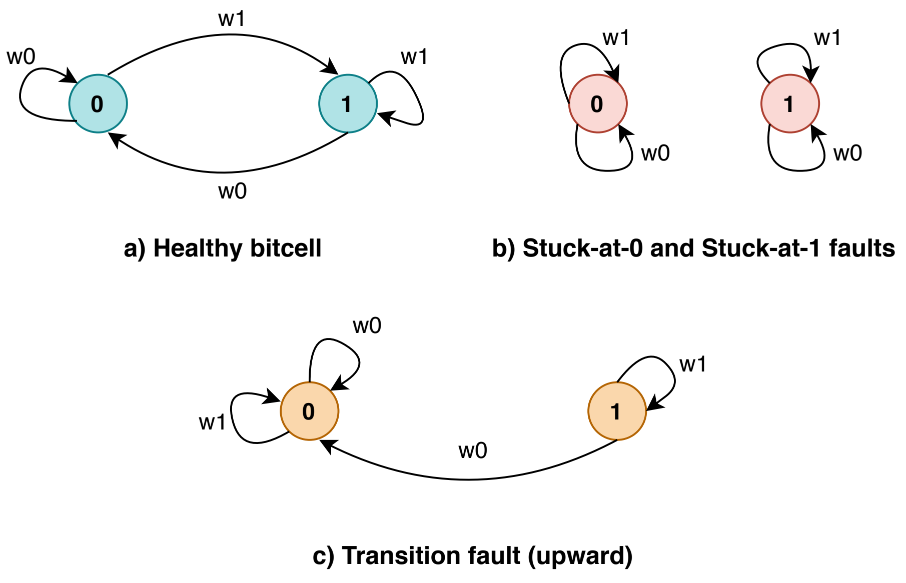

w1
w0
w1
0
1
w0
a) Healthy bitcell
w1
w1
0
1
w0
w0
b) Stuck-at-0 and Stuck-at-1 faults
w0
w1
0
1
w1
w0
c) Transition fault (upward)

Figure 3.2: Fault Models

Stuck-at Faults

Stuck-at faults (Figure 3.2 b.) occur when defects cause opens or shorts in bitlines or drivers. Such faults cause the output of a memory cell to get permanently forced to a particular value. Cells can be stuck-at ones (s-a-1) or zeroes (s-a-0).

Transition Faults

Transition faults (Figure 3.2 c.) are cases where cells fail to transition to a value. For example, if a cell cannot go through upward transitions (‘0’→‘1’), a “1” cannot be stored in it again once a “0” has been written.

Stuck-open Faults

Stuck open faults occur when a memory cell cannot be accessed. For example, if the address decoder is faulty, one or more of the wordlines might not get selected.

3.3 Test Algorithm

In a typical memory BIST approach, test patterns are written into cells, read back, and compared with the expected responses to generate fault signatures and related information. Typical examples of memory test algorithms are the Checkerboard tests and March tests [8]. Even though checkerboard tests are simpler, March tests are popular because they can detect a variety of faults [9]. The test patterns in March tests are applied by marching through the memory in order of ascending or descending addresses. In our implementation, we use the MATS++ test [10], a variant of the march tests.

$$\begin{array}{ccc} \Updownarrow(w0); & \Uparrow(r0,\ w1); & \Downarrow(r1,\ w0,\ r0) \\ M_0 & M_1 & M_2 \end{array}$$

Figure 3.3: The MATS++ Algorithm

The MATS++ test is a sequence of March elements, where each element is an operation (read or write) performed at addresses in a marching fashion. The test is described in Figure 3.3, with the arrows showing the marching order. Although every element here represents an operation at a cell, this can be easily extended for word-oriented memories such as OpenRAM SRAMs – instead of performing an operation at a single cell, it is performed for all bitcells in a word.

M0 is the first element, where the address sequence can start either at the highest or the lowest address, as shown by the double-arrow. For this BIST, we start at the highest address and go in the descending order. For every address, the element's operation is performed, which in case of M0, is writing a "0" at the address. This means every bit in the word is initialized with a "0" as part of the M0 element. Next, the element M1 begins, where we go in ascending order of addresses, read an expected "0"" and write a "1" in its place. Finally, the first operation in element M2 is to read the "1" written previously, followed by writing and reading "0"s. If there is any difference in the read outputs from the expected responses, that cell is deemed faulty.

This algorithm can be used to detect all the faults discussed in Section 3.2. It detects stuck-at and stuck-open faults since we write a complemented value at every cell and read again. It also detects transition faults since we write all "0"s first, followed by "1"s, followed by "0"s again; this procedure covers both the upwards and downwards transitions.

3.4 BIST Architecture

The BIST architecture in Figure 3.4 consists of the BIST Controller and the Address Generator. The spare rows and columns are not tested with BIST and are assumed to be fault-free.

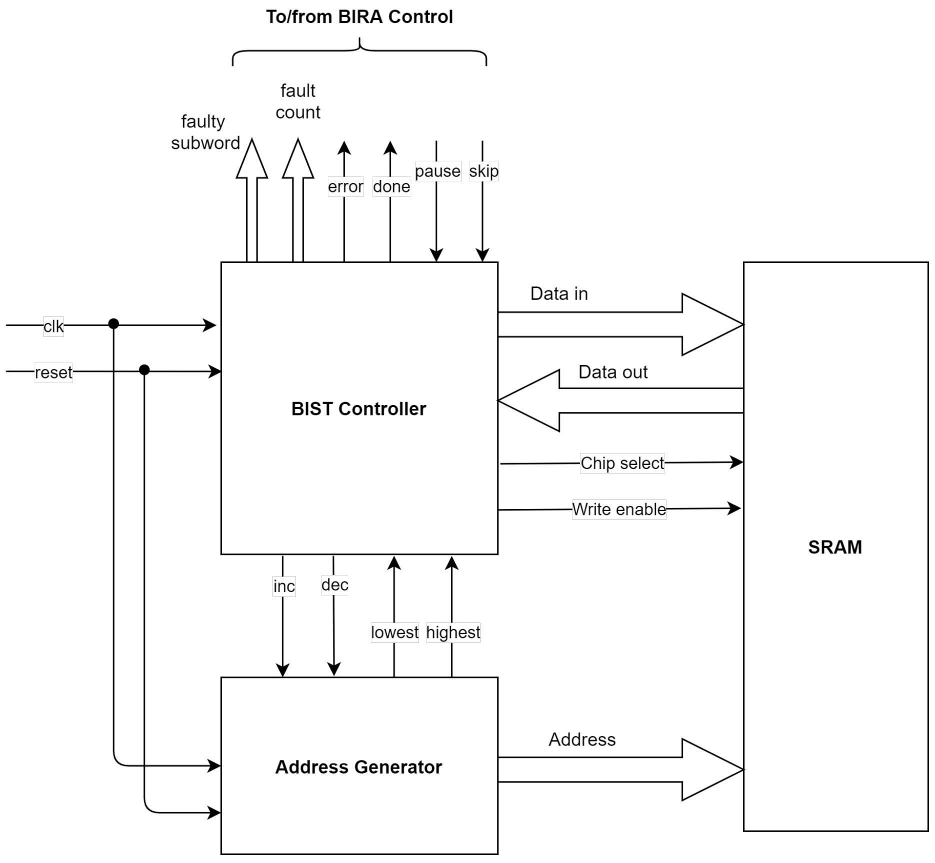


Figure 3.4: BIST Top

The address generator generates the marching addresses and also sends the fault's address to BIRA. It initializes to the highest address on the reset pulse, and increments or decrements an address when the controller sends *inc* or *dec* signals. If both signals are low, the address value is held constant. It also informs the controller when the lowest or highest address is reached, so that the controller can go to the appropriate state in the next clock cycle.

The BIST controller handles data in/out and enable signals to the SRAM and also communicates with the BIRA controller (discussed further in Section 4.2). When an error is detected, the *error* signal to BIRA goes high. The *pause* is an active-high input from BIRA which indicates that the remapper is working on storing the fault information, and so all output values to BIRA should be held constant until it goes low. The *skip* input tells the BIST to move to the next address and forces *error* to low.

Once BIST is finished, *done* goes high, and the repair wrapper switches to normal mode.

3.5 BIST State Machine

The BIST state machine in Figure 3.5 has a total of eight states. Each state corresponds to one march element in the test. The BIST controller stays in the same state until the element's operations are performed on all memory cells.

On reset, it transitions to the first state, ***write_descend_zeroes***, where zeroes are written into all addresses starting with the highest address (excluding the spare rows) and going to the lowest. Next, reads are done in an ascending order. The ***read_ascend_zeroes*** state sets up the read by providing necessary enable signals, the next state ***write_ascend_ones*** sees the actual read output and also writes ones at the same address. Similarly, the next march element operations are carried out in ***read_descend_ones***, ***write_zero_last*** and ***read_zero_last*** states.

The state machine enters the ***error*** state if a fault is detected, where fault-related information is sent to BIRA. When pause goes low, it exits the error state, skips the address, goes back to the previous state, and continues testing from there.

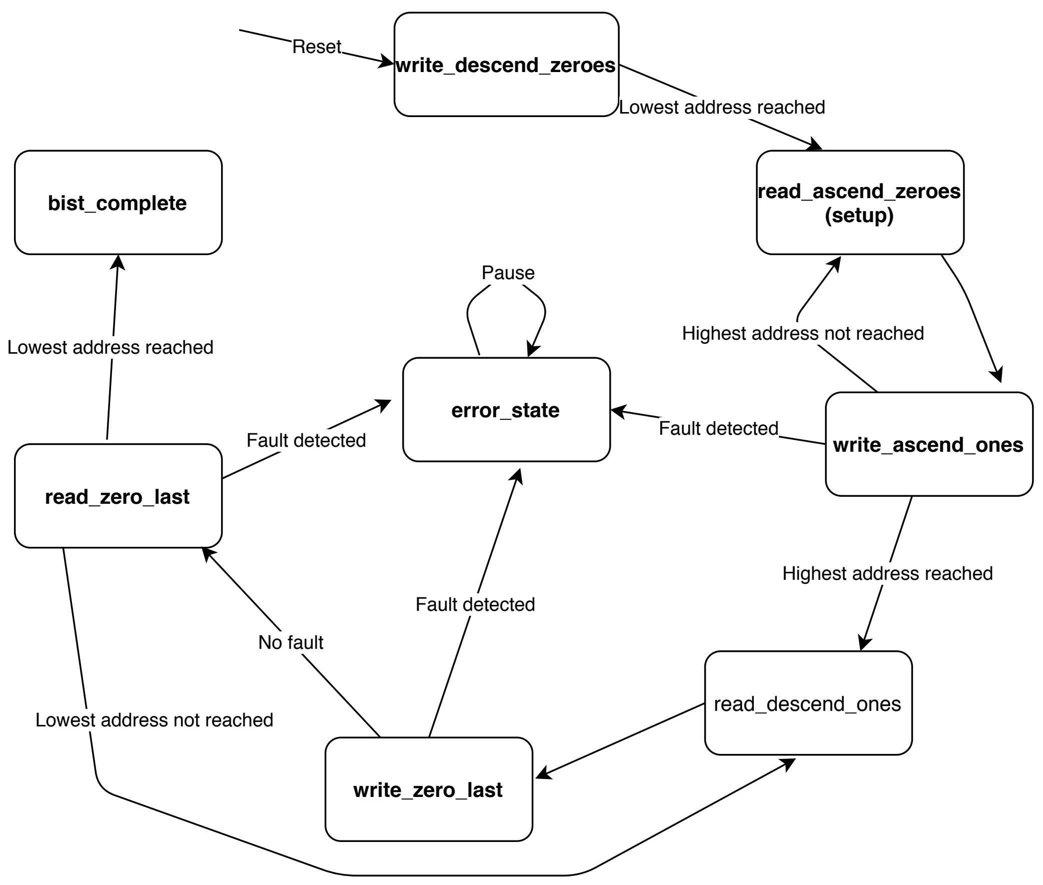


Figure 3.5: BIST State Machine

When the March test is finished (***bist_complete***), BIST controller sends a *bist_done* signal to the redundancy allocator.

Chapter 4

Self-Repair

While BIST can be used to diagnose faults and improve fabrication processes to avoid them, some repair capability is crucial for enhancing production yields in high volume manufacturing. This is even more important for memories, since memory yields determine overall SOC yields to a large extent.

The repair feature added to OpenRAM is designed as a synthesizable Verilog wrapper, similar to how BIST is implemented. The repair logic works with the self-test logic to allocate spare cells to defective cells during the repair phase and also performs data/address remapping after the repair is complete. This chapter discusses how the redundancy is organized and how repair and remapping are done.

4.1 Fault Replacement

Figure 4.1 shows how the redundancy replaces faults. This redundancy organization and repair architecture is based on the BISR scheme proposed by J. F. Li et al. [11] in their paper on 2D redundancy repairs. Every word is logically divided into sub-words; for example, a sub-word may have 2 bits if the word size is 8 bits. Although spare columns can be individually enabled, as explained in Section 2.3.2.1, they are divided into groups. One spare column group replaces a sub-word for a range of rows. The number of spare columns should be a multiple of the number of bits in every sub-word. The number of spare rows can be arbitrary, and one such row is meant to replace an entire regular row if the spare columns are not sufficient. Faults are always assigned to spare columns first; row allocation is done only if the column groups are full. Fault coverage depends on the number of spare columns and rows.

Having sub-words instead of bit-level replacements reduces the area overhead of the repair logic since a smaller number of registers is needed to keep track of the remapping. It also allows the logic to handle multiple bitcell repairs at once. If faults are close by, they will likely fall in the same sub-word and hence can be repaired in one go.

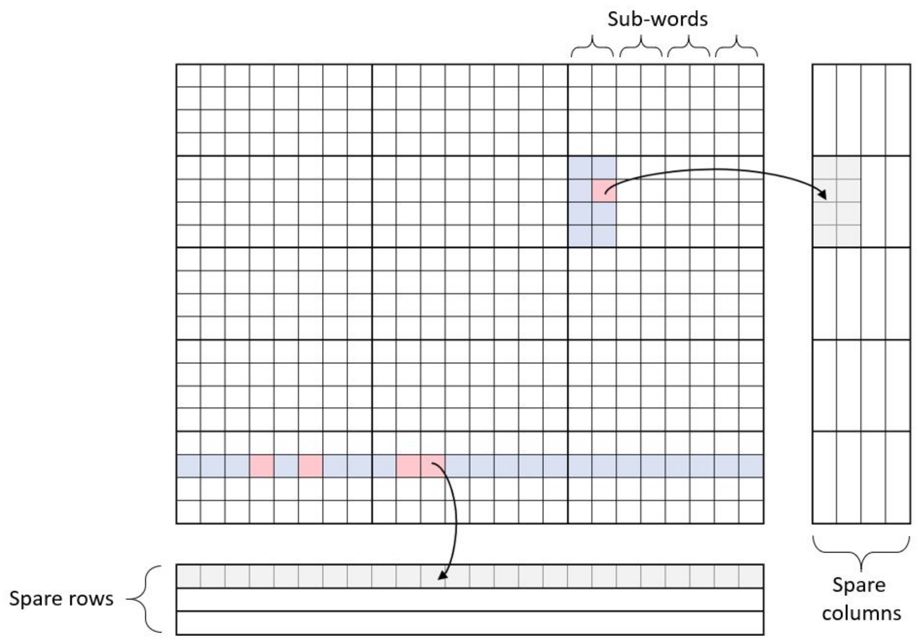


Figure 4.1: Fault replacement of defective cells (in red) and regular cells in (in blue) with the spare cells (in grey)

4.2 BIRA Controller

The controller is a finite state machine that instructs the remapper to store fault information. It enters the monitor state on power-on reset, where it looks out for the error signal coming from the BIST module. It receives the fault address and faulty sub-word inputs from the BIST. The faulty sub-word is an array input that shows which of the sub-words in a word are faulty (indicated by a "1"). When BIST informs the controller about an error, it decides to allocate a column group or an entire row for the fault according to the remapper's status. If no spares are available, the controller exports the fault's address to the system. The controller and remapper are shown in Figure 4.2.

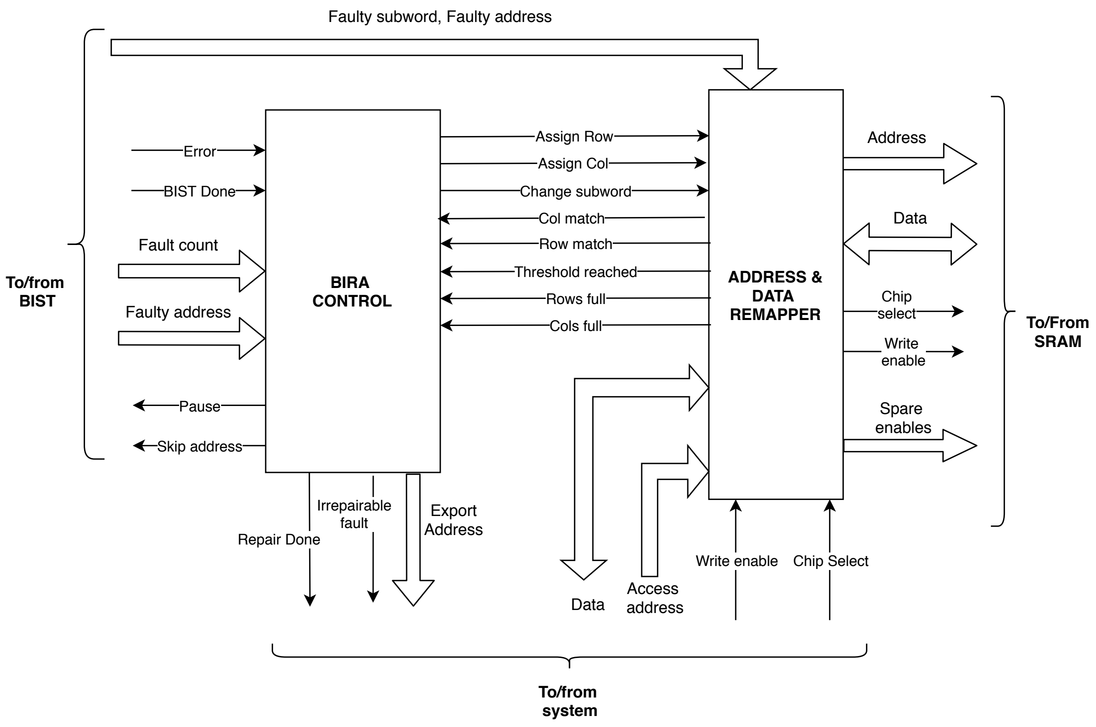


Figure 4.2: BIRA Control and Remapper

4.2.1 State Machine

The BIRA state machine in Figure 4.3 has eight states. On reset, it enters the ***monitor*** state and waits for the error signal from BIST.

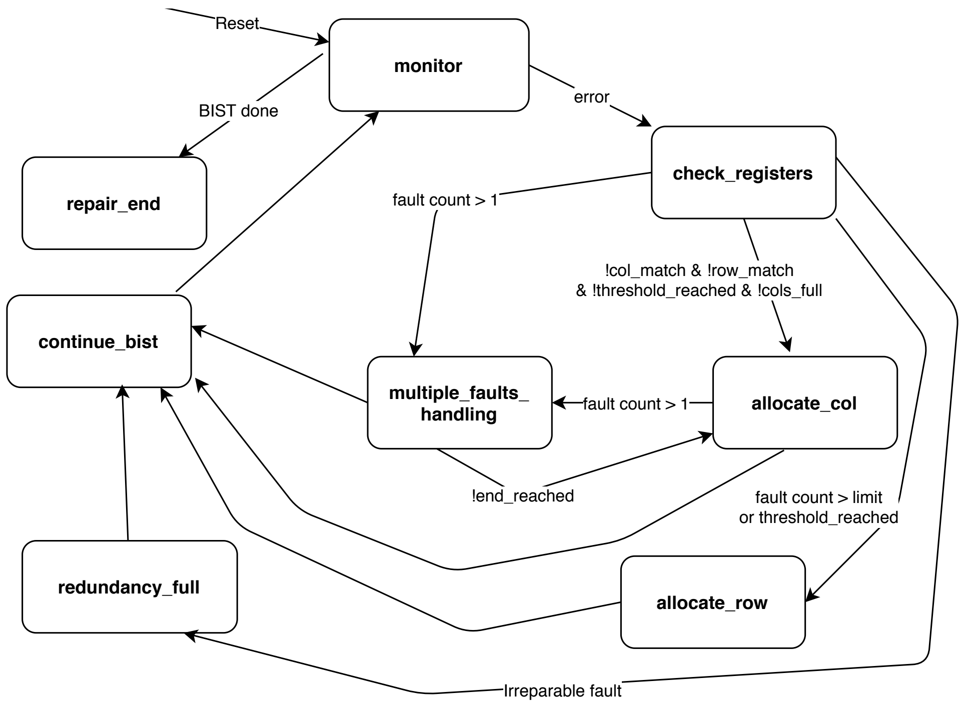


Figure 4.3: BIRA Control State Machine

When BIST detects a fault, the BIRA state machine asks it to pause and then goes to the ***check_registers*** state so that it can decide what to do in the next clock cycle. Here, the status of remapper's registers is observed through five input signals - *column match*, *row match*, *threshold reached*, *rows full* and *cols full*. The match signals are sent by the remapper if the incoming fault has been stored before. The full signals indicate whether all spare columns and rows have been used up. Also, *threshold reached* indicates if all spare col groups for the fault's row address are full.

One of the following is possible in the *check_registers* state:

- If there is a row match or a col match, this means the fault has been handled already, so it goes to the ***continue_bist*** state directly.
- If the threshold has not been reached, the fault is assigned to the first available column group, so it goes to the ***allocate_col*** state.
- If the threshold has been reached, this means the fault needs to be assigned to a row, so it goes to the ***allocate_row*** state.
- If the threshold has been reached, but all spare rows are full, this means the fault cannot be repaired, so it transitions to the *redundancy_full* state.
- If multiple sub-words within a word are faulty, they have to be assigned one by one, so it goes to the *multiple_faults_handling* state which takes care of this case.

Signals *assign_row* and *assign_col* are used to tell the remapper to store the fault into its registers at the positive edge of clock.

In the ***multiple_faults_handling*** state, we check the number of ones in the faulty sub-word input and assign them one by one to the available column groups. For example, if the faulty sub-word is "0110", then the sub-words "0010" and "0100" are stored into two different column groups. For doing this, the state machine goes to the *allocate_col* state, then back to the *multiple_faults_handling* state as many times as required to check and store all faulty sub-words in the word.

If there are too many faults in the same word (shown as fault count > limit in Figure 4.3), we go directly to the *allocate_row* state to assign a spare row to the fault's row.

In the ***redundancy_full*** state, the controller exports the faulty address and asserts a signal to inform the system about the fault. These irreparable faults need to be handled by the system through software-based remapping, so that non-usable addresses are masked.

After one of the above decisions is taken, the state machine goes back to the *monitor* state through the *continue_bist* state. In the ***continue_bist*** state, the *pause* signal to memory is de-asserted and *skip* goes high, so that the address generator can skip this address when it goes back to testing. When BIST is done, the repair phase also ends (***repair_end*** state) and the remapper takes control of the memory. The BIST and BIRA controllers are inactive during the normal mode of operation.

4.3 Address and Data Remapper

The address and data remapper is the most important part of this architecture. It is responsible for storing fault information in the test mode, and deciding inputs to the SRAM when switched to the normal mode. It also provides data outputs from the SRAM to the system in the normal mode.

4.3.1 Test/Repair Mode

In the test mode, registers are used to store allocations to spare cells at the positive clock edge. The registers are numbered such that every register corresponds to a specific spare row or spare column group. A valid bit corresponding to the register number is also set when allocating spares.

- If a row has to be allocated, the row address of the current fault is stored in the first non-valid ***row_info*** register.

- If a spare column group has to be allocated, the column address of the current fault and the faulty sub-word is stored in the first available ***col_info*** and ***sub-word_info*** registers.

4.3.2 Normal Mode

In the normal mode, the remapper compares incoming addresses with the stored ones to decide the data and address inputs to the SRAM. The incoming address from the system will be referred to as the *access_address*.

- If the access address matches with one of the *row_info* fields, then the row address to the SRAM is changed to the corresponding spare row's address.
- For the range of rows the access address falls under, if the column address matches with one of the *col_info* fields:
  - Read operation: The faulty sub-word in the data output from SRAM is replaced with the sub-word stored in the spare column. Then this new word becomes the data output to the system, while the extra output bits from the spares are not visible.
  - Write operation: The data input bits at the locations of the faulty sub-words are copied and written into the spare column groups which were allocated for those sub-words.
- If the access address has been stored in row as well as column registers, then priority is given to the row allocation. Since the row has already been assigned, we do not need to read or write anything to the spare columns.
- If the access address has not been stored in any row or column information registers, then the spare columns are not enabled and the address input to SRAM is not changed.

Chapter 5

Results and Conclusions

This thesis discussed the need for self-repair in memories and how this feature has been added to the OpenRAM memory compiler. In this chapter, we present the results of this work, followed by a summary of what was achieved.

5.1 Simulation Results

Functional faults in the SRAM are simulated by modifying its Verilog model. We perform tests by injecting stuck-at faults at random addresses. All sub-modules of the wrapper are validated with testbenches. The top module is also verified with testbenches by doing random reads and writes to the SRAM and comparing the outputs with the expected responses. The following subsections show the waveform simulation results for the BIST, BIRA, and SRAM_BISR modules. For each all timing diagrams, the following configuration is assumed:

- Row address size = 5 bits
- Column address size = 2 bits
- Word size = 8 bits

- Sub-word size = 2 bits
- Range of rows in a column group = 4
- Number of spare rows = 3
- Number of spare columns = 4

5.1.1 BIST

The BIST top module is tested with the SRAM in this section. Figure 5.1 shows how stuck-at-1 faults are detected by BIST. In this figure, a stuck-at-1 fault is injected at the first 4 MSB bits of the word at address 0000001. Previously, the BIST was in the first march element, writing "0"s at all addresses. When the address is 0000001, the *read_ascend_zeroes* state sets up the read at time t = 660ns expecting all zeroes in the output. In the next clock cycle *write_ascend_ones* state detects an error. The error signal goes high and the state machine transitions to the *error* state ("111") at t = 680ns. The faulty sub-word is sent to BIRA, along with the fault count, which is 2 ("010"), implying two sub-words are faulty. When pause goes low at t = 730ns, test is resumed from the next address.

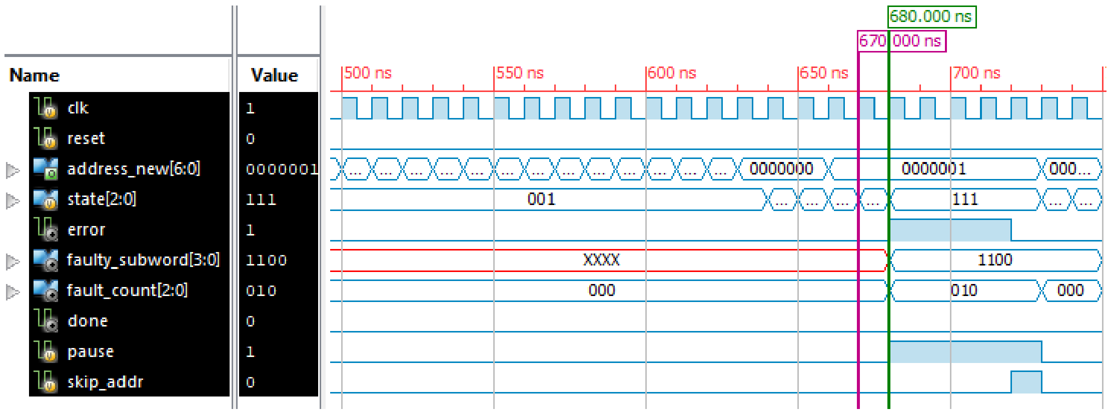


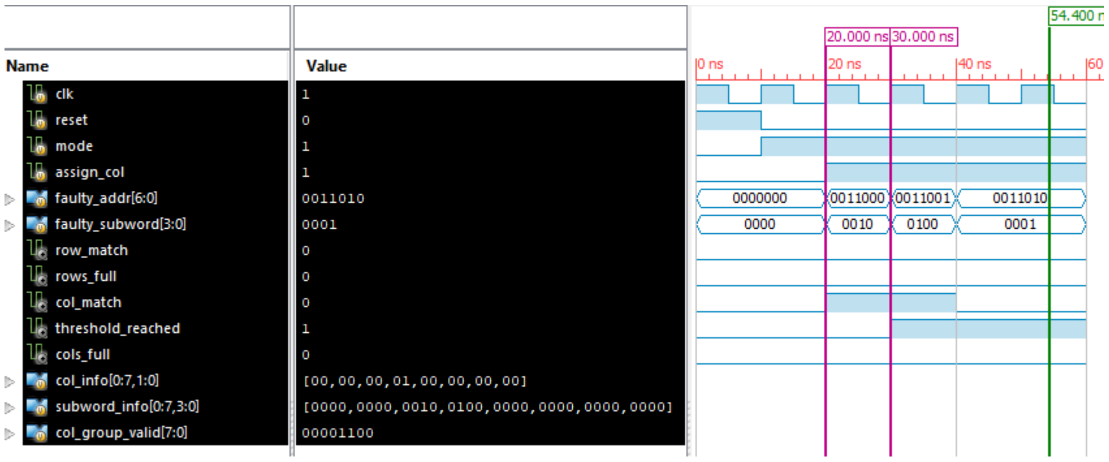
Figure 5.1: BIST simulation for multiple stuck-at-1 faults at an address

5.1.2 Address Remapper

This section details the timing diagram results of the address remapper under different input conditions. For initial verification, it is tested independent of the BIRA control logic.

5.1.2.1 Allocating Spare Columns

The Figure 5.2 shows how spare column groups are allocated for defective cells. In this case, the row address input is for the 7th row. Hence, we see from the valid register that column groups 2 and 3 are responsible for this range of rows. On the *assign_col* input, if the threshold has not been reached, a column group is assigned to the sub-word by storing its information in the corresponding index.



Name	Value
clk	1
reset	0
mode	1
assign_col	1
faulty_addr[6:0]	0011010
faulty_subword[3:0]	0001
row_match	0
rows_full	0
col_match	0
threshold_reached	1
cols_full	0
col_info[0:7,1:0]	[00,00,00,01,00,00,00,00]
subword_info[0:7,3:0]	[0000,0000,0010,0100,0000,0000,0000,0000]
col_group_valid[7:0]	00001100

Figure 5.2: Allocating spare columns to faults in the remapper

At t = 20ns, the sub-word "0010" and column address "00" are stored in the information registers for column group 2. At t = 30ns, the fault information is stored for column group 3. Since there are 2 column groups per row, the threshold has been reached in storing the above two sub-words. So, at t = 40ns, the *assign_col* input does not affect any registers. The *threshold_reached* signal also goes high to inform BIRA.

5.1.2.2 Allocating Spare Rows

Figure 5.3 shows how row allocation works. On the active high *assign_row* input, the fault's row address is stored in the first available *row_info* register. The *row_valid* bit for this register is also set. Faulty sub-word input is ignored since the entire row is replaced. BIRA is informed if rows are full (*rows_full*).

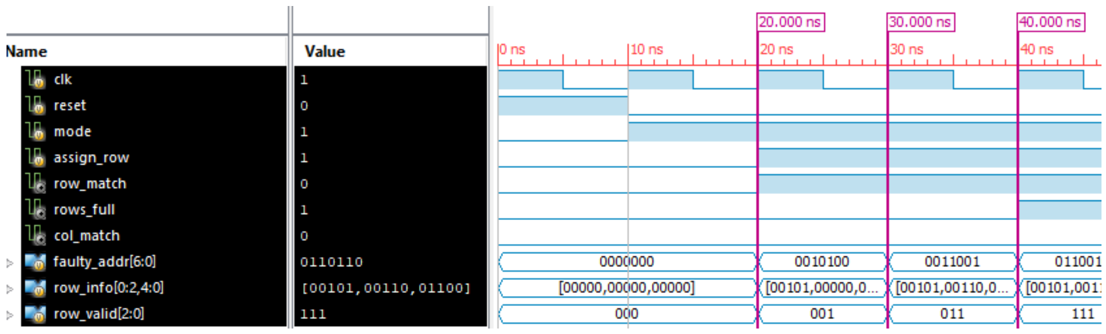


Figure 5.3: Allocating spare rows to faults in the remapper

5.1.2.3 Writing to the SRAM

After repair ends, the address remapper is in the normal mode (*mode* 0). Figure 5.4 shows three cases of writes. In the first case, at t=80ns, the access address from system matches with a column allocated fault, which was allocated at t=30ns. So, the data bits at the faulty sub-word are also written to the spare columns. In the second case at t=110ns, the address to SRAM is changed to that of a spare row. In the third case at t=140ns, the address has not been stored as faulty, so inputs to the SRAM are the same as those from the system.

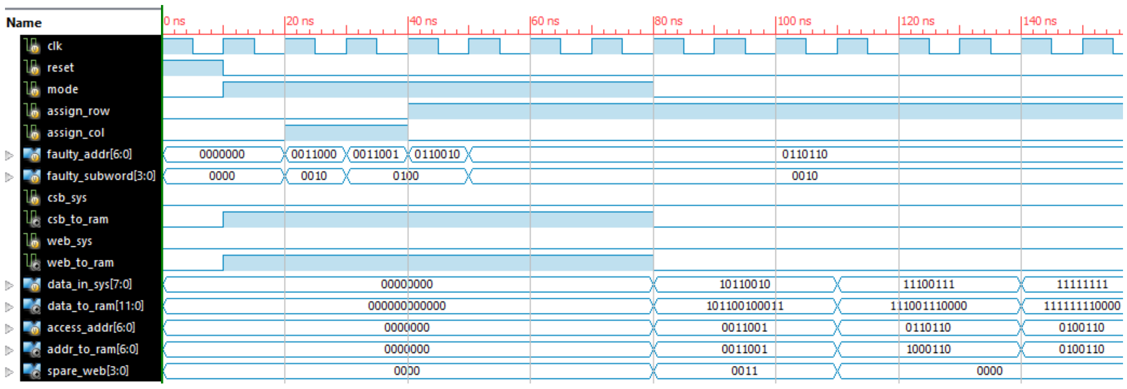


Figure 5.4: Writing to SRAM after repair

5.1.3 BISR

Figure 5.5 shows the normal mode operation of the top module (SRAM_BISR) once repair is finished. Stuck-at-1 faults were introduced at the 3 least significant bits LSB of the word, at the access address shown. At 5000ns, data input from the system was written to this address. At the time marked by the label, data output is read back from this address. Had the fault not been repaired, the three LSBs of the output would be a "1".

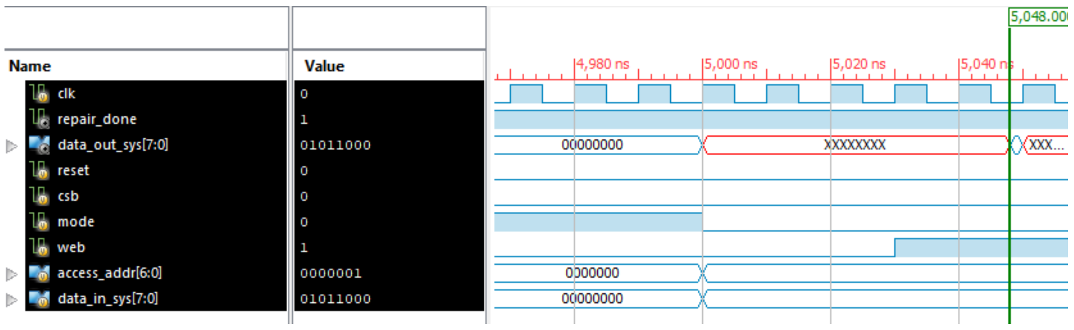


Figure 5.5: SRAM_BISR simulation for repairing multiple bit faults

5.2 Area Costs

We estimate the wrapper's area cost by synthesizing the design using 45nm cell libraries. Moreover, appending extra bitcells to the SRAM introduces additional overheads.

Figure 5.6 shows the area results for the configurations mentioned in Section 5.1. The overhead percentages are given relative to the no-spares SRAM. The no-repairs configuration is a 512-bit memory, to which we add 172 spare bitcells, along with the extra read/write logic described in Chapter 2. As shown in the figure, the SRAM area increases by 50%, while the wrapper takes up 31.5% of the new SRAM's area. Since the wrapper uses flip-flops for storing information about re-allocations, its area also depends on the redundancy.

	No repair	Repair with 4 spare columns and 3 spare rows		
	8b x 64 SRAM	SRAM	BISR	Total area
Area (in μm^2)	9421.19	14189.75	4475.24	18664.99
Overhead		50.60%	47%	98.11%

Figure 5.6: FreePDK 45nm area for SRAM and wrapper

In this design, the maximum critical path is between the address output of the address generator and the state flip-flop of BIRA controller. Similar results can be derived for other configurations of repair. The obvious approach to optimize for area is by reducing the number of spares. However, because they decide the number of reparable faults, a tradeoff exists between the desired repair efficiency and area constraints.

5.3 Conclusion

This thesis is an effort to expand the capabilities of the OpenRAM memory compiler. A synthesizable wrapper was designed and memory repair using redundancies was explored. The OpenRAM SRAM was modified to support parametrized number of rows and columns, granting flexibility in choosing SRAM sizes.

Future work involves estimating the wrapper's repair rates and power overheads, and optimizing the design for them. Advanced march tests for word-oriented memories can also be implemented to detect more varieties of faults.

Bibliography

[1] L. H. Goldstein and E. L. Thigpen, "SCOAP: Sandia Controllability/Observability Analysis Program," in *17th Design Automation Conference*, pp. 190–196, 1980.

[2] M. R. Guthaus, J. E. Stine, S. Ataei, Brian Chen, Bin Wu, and M. Sarwar, "OpenRAM: An open-source memory compiler," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–6, 2016.

[3] H. Nichols, M. Grimes, J. Sowash, J. Cirimelli-Low, and M. R. Guthaus, "Automated Synthesis of Multi-Port Memories and Control," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 59–64, 2019.

[4] "NCSU EDA FreePDK45." `https://www.eda.ncsu.edu/wiki/FreePDK45:Contents`. 2020.

[5] "MOSIS SCMOS design kit." `https://www.mosis.com/pages/design/flows/design-flow-scmos-kits`. 2020.

[6] "SkyWater SKY130 PDK." `https://github.com/google/skywater-pdk`. 2020.

[7] J. Sowash, "Design of a RISC-V Processor with OpenRAM Memories." *UC Santa Cruz*, 2019.

[8] Suk and Reddy, "A March Test for Functional Faults in Semiconductor Random Access Memories," *IEEE Transactions on Computers*, vol. C-30, no. 12, pp. 982–985, 1981.

[9] Von-Kyoung Kim and T. Chen, "On comparing functional fault coverage and defect coverage for memory testing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 11, pp. 1676–1683, 1999.

[10] van de Goor, A. J., *Testing Semiconductor Memories: Theory and Practice*. John Wiley Sons, Inc., 1991.

[11] Jin-Fu Li, Jen-Chieh Yeh, Rei-Fu Huang, and Cheng-Wen Wu, "A built-in self-repair scheme for semiconductor memories with 2-d redundancy," in *International Test Conference, 2003. Proceedings. ITC 2003.*, vol. 1, pp. 393–402, 2003.