Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет __Комп’ютерної інженерії та управління___________________
(повна назва)

Кафедра Автоматизації проектування обчислювальної техніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти____другий (магістерський)____________________
(рівень вищої освіти)

Методи діагностування запам'ятовуючих пристроїв на основі
часових рекурсивних автоматів
(тема)

Виконав: студент 2 курсу, групи СКСм-21-1
__________Корнієнко М.Р.___________
(прізвище, ініціали)
Спеціальність *123 _Комп’ютерна інженерія*
Тип програми ____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)
Освітня програма_ *Спеціалізовані*________
_*комп’ютерні _системи*___________
(повна назва освітньої прграми)

Керівник роботи _доцент Шкіль О. С._
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ___________ _Чумаченко С.В
(підпис) (прізвище, ініціали)

2022 р.

Харківський національний університет радіоелектроніки

Факультет________Комп'ютерної інженерії та управління ________________

Кафедра_______Автоматизації проектування обчислювальної техніки______

Рівень вищої освіти____другий (магістерський)__________________________

Спеціальність_____123_ Комп'ютерна інженерія _______________________

(шифр і назва)

Тип програми ________Освітньо-професійна___________________________

(освітньо-професійна або освітньо-наукова)

Освітня програма ________Спеціалізовані комп'ютерні системи__________

(повна назва

ЗАТВЕРДЖУЮ:

Зав.кафедри_________________

(підпис)

"_02__" ____09___ 2022 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові _______Корнієнко Максиму Романовичу________________

(прізвище, ім'я, по батькові)

Тема роботи (проекту) Методи діагностування запам'ятовуючих пристроїв на основі часових рекурсивних автоматів____________________________

затверджена наказом по університету від "14"____11_____2022 р. № _1478 Ст_____

1. Термін подання студентом роботи до екзаменаційної комісії ____15.12.2022______
2. Вихідні дані до роботи (проекту)____

Мова програмування VHDL ______________________________________

Рекурсивні автомати____

Теорія автоматів____

Теорія діагностування запам'ятовуючих пристроїв____________________

3. Перелік питань, що потрібно опрацювати у роботі _________________

Існуючі методи з діагностування пам'яті

Існуючі несправності пам'яті

Вибір платформи для реалізації проекту

Проектування рекурсивного автомату

Проектування системи тестування пам'яті

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) ___21 слайд

6. Консультанти розділів роботи (проекту)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

7. Дата видачі завдання 27.04.2022

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи (проекту)	Термін виконання етапів проекту (роботи)	Примітка
1	Видача теми проекту, узгодження і затвердження теми	01.09.2022 -27.09.2022	
2	Аналіз існуюючих методів діагностування	28.09.2022 – 04.10.2022	
3	Аналіз існуючих несправностей	05.10.2022 - 13.10.2022	
4	Аналіз та вибір платформи для реалізації	14.10.2022 -24.10.2022	
5	Розробка системи	25.10.2022 -04.11.2022	
6	Розробка та імплементація системи	05.11.2022 -29.11.2022	
7	Оформлення пояснювальної записки	30.11.2022 -06.12.2022	
8	Перевірка виконаного проекту керівником	07.12.2022 -14.12.2022	
9	Захист проекту	15.12.2022 -22.12.2022	

Студент ______________________________
(підпис)

Керівник роботи (проекту) ____________ доцент Шкіль О.С.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка містить 60 сторінки, 28 рисунків, 14 джерел за переліком посилань.

RANDOM-ACCESS MEMORY, РЕКУРСИВНИЙ АВТОМАТ, MARCH C- АЛГОРИТМ, ТЕСТИ MARCH, VHDL, КІНЦЕВИЙ АВТОМАТ, FPGA, SPARTAN6, BIST

Розглянуто існуючі алгоритми тестування RAM. Розглянуто концепцію BIST. Для синтезу алгоритмів і програм тестів діагностування запам'ятовуючих пристроїв запропоновано систему генерації та контролю за виконанням тестів, що містить пристрій управління, матрицю запам'ятовуючих комірок і чотири головки запису (зчитування). Наведено приклад синтезу за допомогою даної системи тесту March C-. Запропоновано реалізацію алгоритму шляхом автоматного проектування, а саме за допомогою моделі часового рекурсивного автомату. Спроектовану систему імплементовано та протестовано на FPGA Xilinx Spartan6.

ABSTRACT

Thesis contains 60 pages, 28 figures, 14 sources according to the list of links.

RANDOM-ACCESS MEMORY, RECURSIVE AUTOMATA, MARCH C-ALGORITHM, MARCH TESTS, VHDL, FINITE AUTOMATA, FPGA, SPARTAN6, BIST

Existing RAM testing algorithms are considered. The concept of BIST is considered. For the synthesis of algorithms and test programs for the diagnosis of memory devices, a test generation and control system is proposed, which includes a control device, a matrix of memory cells, and four recording (reading) heads. An example of synthesis using this system of the March C- test is given. It is proposed to implement the algorithm by means of automatic design, namely by means of a time recursive automaton model. The designed system is implemented and tested on a Xilinx Spartan6 FPGA.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 7
ВСТУП 8
АЛГОРИТМИ ТЕСТУВАННЯ ПАМ'ЯТІ 10
1.1 Моделі напівпровідникової пам'яті 10
1.2 Складність тестування 13
1.3 Несправності секцій пам'яті 14
1.3.1 Одноклітинні помилки 15
1.3.2 Двоклітинні помилки 18
1.3.3 Усі інші помилки 20
1.4 Існуючі алгоритми тестування 22
1.5 Вибір алгоритму для реалізації 26
2 РОЗРОБКА СИСТЕМИ ТЕСТУВАННЯ 29
2.1 Концепція BIST 29
2.2 Моделювання системи 30
2.2 Синтез алгоритму теста MARCH C 34
2.3 Граф переходів автомату 38
3 АВТОМАТНЕ ПРОГРАМУВАННЯ 40
3.1 Теорія автоматів 40
3.2 Рекурсивні автомати 41
3.3 Порівняння рекурсивного та ітеративного підходу 44
3.4 Реалізація рекурсивного автомату на мові VHDL 45
4 ПРОГРАМУВАННЯ ТА ІМПЛЕМЕНТАЦІЯ 48
4.1 Платформа для імплементації 48
4.2 VHDL опис системи 49
4.3 Імплементація засобами Xilinx ISE 52
5 ТЕСТУВАННЯ СИСТЕМИ 53
5.1 Перевірка працездатності окремих станів 53
5.2 Час виконання 56
ВИСНОВКИ 58
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 59

ДОДАТОК А 60
ДОДАТОК Б ....**Ошибка! Закладка не определена.**

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ADF – Address Decoder Fault;

BIST – Built-in self-test;

ISE – Identity Services Engine;

ОЗУ – Оперативна пам'ять;

RAM – Random Access Memory;

VLSI – Very large-scale integration;

DDR – Double data rate;

ПЛІС – Програмно-логічна інтеграційна схема;

DFT – Design for Testability;

SRAM – Static random-access memory;

DRAM – Dynamic random-access memory;

FSM – Finite-State Machine;

IC – Intergrated Circuit;

FFM – Functional Fault Models;

LUT – Look-up Table;

SAF – Stuck-at Fault;

SOF – Stuck-open Fault;

TF – Transition Fault;

CF – Coupling Fault;

NPSF – Neighborhood Pattern Sensitive Faults;

LF – Linked Fault;

FPGA – Field-Programmable Gate Array;

RSM – Recursive State Machine;

VHDL – Hardware Description Language.

ВСТУП

Оскільки системи стають все складнішими, потреба в систематичних стратегіях тестування також зростає. Збій на вищому системному рівні в середньому призведе до підвищення вартості ремонту в десять разів. Тому виробники складних систем надають дуже велике значення високонадійним компонентам.

У наш час роль запам'ятовуючих пристроїв у напівпровідниковій промисловості надвелика. Такі сфери, як комп’ютерна графіка, цифрова обробка сигналів і швидкий пошук величезних обсягів даних, вимагають експоненціального збільшення обсягу пам’яті. Таким чином, постійно зростаючий відсоток області інтегральних схем (IC) присвячений реалізації структур пам’яті.

Постійні дослідження з метою удосконалення історично підштовхують технологію пам’яті до межі, роблячи ці пристрої надзвичайно чутливими до фізичних дефектів і впливу навколишнього середовища, які можуть серйозно скомпрометувати їх правильну роботу. Схеми оперативної пам'яті є одними з високощільних схем VLSI, які виготовляються сьогодні. Оскільки лінії транзисторів розташовані дуже близько одна до одної, схеми оперативної пам’яті страждають від дуже високої середньої кількості фізичних дефектів на одиницю площі мікросхеми порівняно з іншими схемами. Цей факт спонукає розробку ефективних тестових послідовностей оперативної пам’яті, які забезпечують добре покриття несправностей. Тому ефективне та детальне тестування компонентів пам’яті є обов’язковим. Пам'ять не включає логічні вентилі та тригери. У результаті для тестування пам’яті потрібні різні алгоритми тестування.

Сучасні мікросхеми напівпровідникової пам'яті типу DDR і QDR можуть за одну транзакцію порахувати або записати пакет даних, що містить кілька слів. При зчитуванні такі мікросхеми видають спочатку перше слово,

потім друге, третє тощо або можуть змінити послідовність видачі слів у будь-якій їх комбінації. Для виконання тестового діагностування сучасних швидкодіючих мікросхем пам'яті доцільно застосовувати тестери, що мають мультипроцесорну структуру, що містять кілька груп операційних процесорів та забезпечують паралельне формування тестових впливів суміжних тактів. Нові функціональні можливості сучасних мікросхем пам'яті потребують розробки засобів моделювання алгоритмів, здатних скоротити трудомісткість синтезу нових програм тестів.

Для забезпечення функціонування зазначених діагностичних пристроїв розпаралелювання алгоритмів тестів необхідно виконувати так, щоб забезпечити синхронну взаємодію окремих операційних процесорів. Для вирішення цього завдання потрібно застосування спеціальних засобів візуалізації алгоритмів тестів, що дозволить суттєво знизити трудомісткість проектних робіт.

Моделювання помилок, ймовірно, є одним із найважливіших елементів тестування пам’яті та аналізу помилок. Оскільки фізичне обстеження пам’яті зазвичай надто складне, інженери-випробувачі вдаються до моделей функціональних несправностей (FFM), щоб створити ефективні алгоритми функціонального тестування.

Не менше важливим ніж моделювання помилок та проектування тестової послідовності є засоби імплементації цих тестових послідовностей, бо цей фактор також впливає на ефективність та оптимальний часові витрати. На питанні засобів реалізації систем тестування пам’яті буде зроблено окремий акцент у цій роботі.

АЛГОРИТМИ ТЕСТУВАННЯ ПАМ'ЯТІ

1.1 Моделі напівпровідникової пам'яті

Типова модель пам'яті складається з комірок пам'яті, з'єднаних у двовимірний масив, і, отже, продуктивність комірки пам'яті потрібно аналізувати в контексті структури масиву. У структурі масиву комірка пам'яті складається з двох основних компонентів: «вузол зберігання» та «вибраний пристрій». Компонент «вибір пристрою» сприяє адресації комірки пам'яті для читання/запису в масиві. На обмеження масштабування пам'яті впливають обидва ці компоненти.

Як показано на рисунку 1.1, декодери рядків і адрес визначають адресу комірки, до якої потрібно отримати доступ. На основі адрес у декодерах рядків і стовпців вибираються відповідні рядок і стовпець, які потім підключаються до підсилювача чутливості. Підсилювач сенсу(sense amplifier) підсилює та надсилає дані.

Так само ми можемо отримати доступ до потрібної комірки, куди потрібно записати дані. Для запису значень у комірку з шини даних використовується спеціальна схема. Для декодерів ми перевіряємо функціональність, чи можуть вони отримати доступ до потрібних клітинок на основі адреси в адресній шині. Для підсилювача та драйвера ми перевіряємо, чи можуть вони правильно передавати значення до клітинок і з них.

На рисунку 1.1 наведено загальну модель мікросхеми динамічної пам'яті (DRAM). Для мікросхеми статичної оперативної пам'яті (SRAM) логіка оновлення буде пропущена. Перш ніж детально обговорювати рисунок 1, слід ознайомитися з різницею у зовнішній і внутрішній організації масиву комірок пам'яті D. Мегабітний чіп може виглядати зовні як такий, що має мільйон адрес слів шириною в 1 біт. Всередині комірки пам'яті (кожна з яких містить один біт даних) організовані у вигляді матриці або кількох матриць.

Наприклад, він може бути організований як одна матриця 1K*1K біт або чотири матриці 512*512, або 8 матриць 128*1K. Для кожної операції читання та запису рядок розміром 1 Кбіт (або менше) зчитується або записується всередину, тоді як лише 1 біт стає видимим для зовнішнього світу.

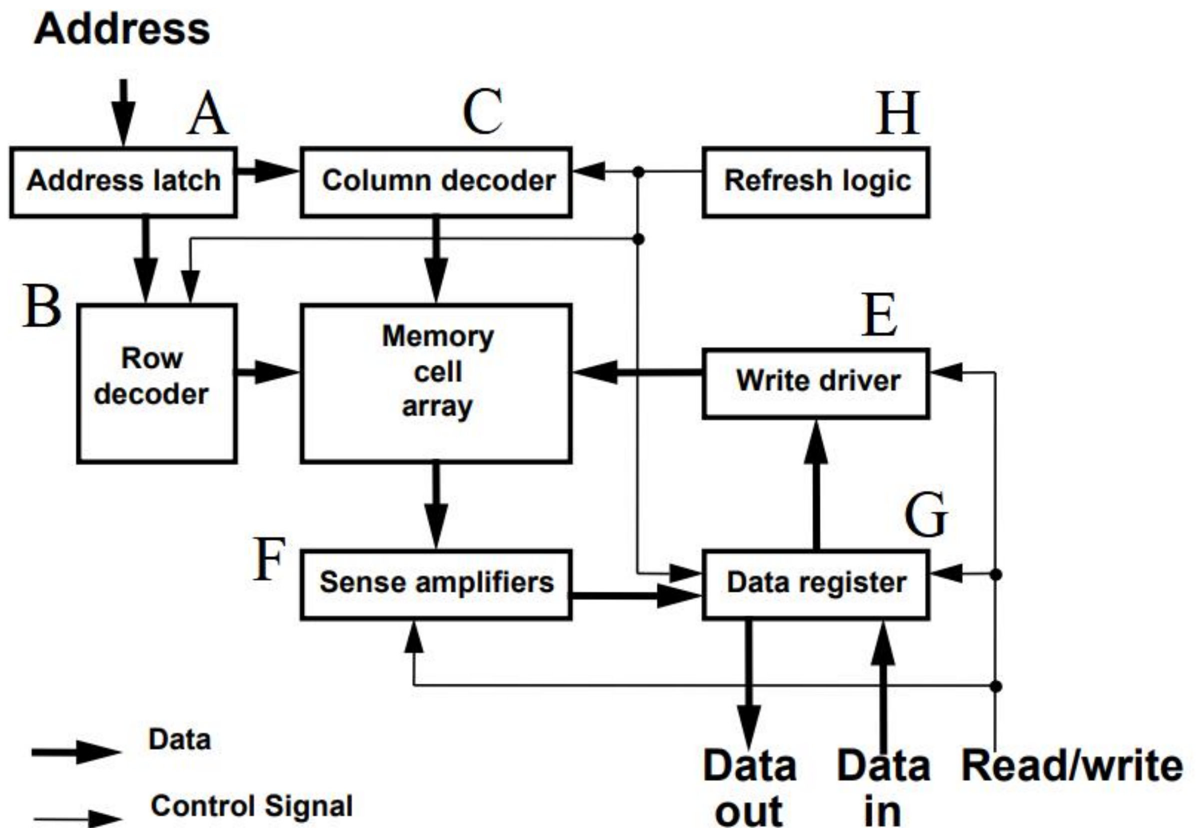


Рисунок 1.1 – Архітектура пам’яті

Пам’ять розрізняють за типу комірок. Розріняють статичні RAM (SRAM) і динамічні RAM (DRAM); у SRAM бістабільний електричний компонент використовується для зберігання двійкового значення. Коли комірку встановлено в один із двох можливих станів («0» або «1»), вона залишається в ньому, доки пристрій не вимкнеться. Комірки DRAM натомість складаються з простих ємнісних елементів, які зберігають електричні заряди для відображення певного логічного рівня, і характеризуються поступовою втратою збереженої інформації через фізичне явище, відоме як струм витоку транзистора. Як наслідок, комірки DRAM

потрібно періодично «перезаряджати» («оновлювати», «перезаписувати»), щоб не втратити збережене значення. На рисунку 1.2 показано приклад структурної моделі рівня пристрою: (A) комірки SRAM і (B) комірки DRAM.

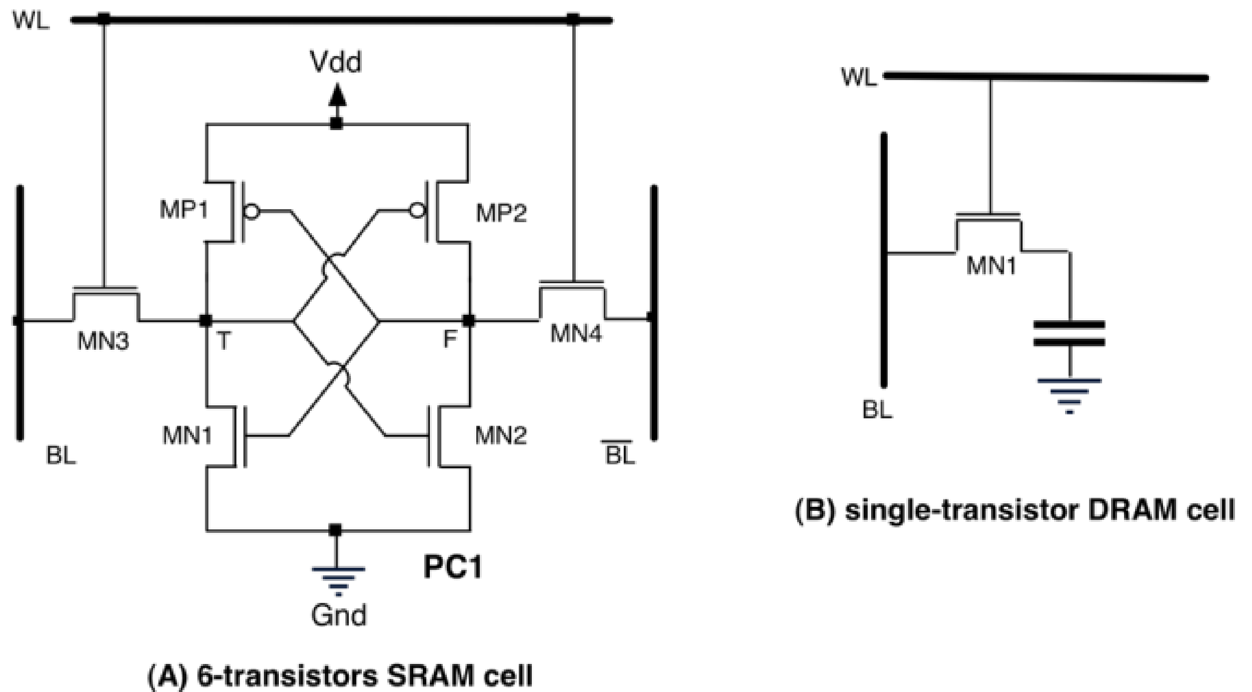


Рисунок 1.2 – (A) SRAM і (B) структурна модель DRAM на рівні пристрою

Під час тестування мікросхеми ніхто не зацікавлений у виявленні несправності, тому що мікросхему неможливо відремонтувати. Люди зацікавлені лише у виявленні несправності. З цієї причини модель на рисунку 1.1 можна спростити без втрати інформації. Для функціонального тестування використовується модель, яка містить лише три блоки: дешифратор адреси, масив комірок пам'яті та логіку читання/запису; дивіться рисунок 1.3. Фіксатор адреси A, рядок B і декодер стовпця C на рисунку 1.1 об’єднані в декодер адреси; це робиться тому, що всі вони стосуються адресації потрібної клітинки чи слова. Драйвер запису E, підсилювач F сенсора і регістр даних G об'єднані в логіку читання/запису; це зроблено тому, що всі вони стосуються транспортування даних з і до масиву пам'яті. Частина оновлення H пропущена, оскільки вона належить до динамічної моделі, яка не обговорюється в цьому документі.

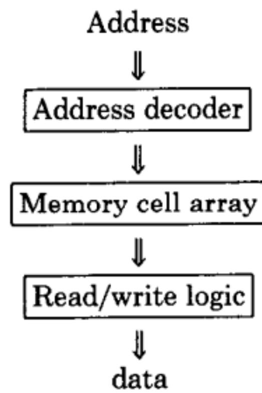


Рисунок 1.3 – Зменшена функціональна модель

1.2 Складність тестування

Для тестування сучасної пам’яті високої щільності традиційні алгоритми потребують занадто багато часу. Наприклад, для GALPAT і WALKING I/O потрібен час тестування порядку n^2 і n^3/2 (де n – кількість бітів у мікросхемі). За такої швидкості, припускаючи час циклу 100 нс, для тестування 16-мегабітного чіпа знадобиться 500 годин для тесту n^2 і 860 секунд для тесту порядку n^3/2. Інші старіші тести, такі як Zero-One і Checkerboard, мають порядок n, але вони мають погане покриття помилок. Таблиця 1 показує час тестування пам’яті як функцію розміру пам’яті.

Таблиця 1.1 – Час перевірки як функція від обсягу пам'яті

Size	Complexity			
n	n	$n \log n$	$n^{3/2}$	n^2
1K	0.0001s	0.001s	0.0033s	0.105s
4K	0.0004s	0.0048s	0.0262s	1.7s
16K	0.0016s	0.0224s	0.21s	27s
64K	0.0064s	0.1s	1.678s	7.17m
256K	0.0256s	0.46s	13.4s	1.9h
1M	0.102s	2.04s	1.83m	1.27d
4M	0.41s	9.02s	14.3m	20.39d
16M	1.64s	39.36s	1.9h	326d
64M	6.56s	2.843m	15.25h	14.3y
256M	26.24s	12.25m	5.1d	229y
1G	1.75m	52.48m	40.8d	3659y

Протягом багатьох років досліджувалися та пропонувалися різні підходи до перевірки пам'яті. Найбільш традиційний підхід полягає в тому, щоб просто застосувати послідовність тестових шаблонів до контактів вводу/виводу та перевірити функціональність пам'яті. Шахова дошка, Zero-one (MSCAN) [Breuer & Friedman, 1976], ходьба , галоп , тести MARCH – тест Marching 1/0 [Breuer & Friedman, 1976], тест MATS [Nair] , Thatte & Abraham, 1979], MATS+ Test [Abadir & Reghbati, 1983], MATS++ [Goor, 1991], MARCH C [Marinescu, 1982], MARCH C- [Goor , 1991], MARCH A [Suk & Reddy, 1981], MARCH B [Suk & Reddy, 1981], метелик, рухома інверсія (MOVI) [De Jonge & Smeulders, 1976], об'ємне звучання disturb є деякі традиційні шаблонні тести. Більш сучасний підхід полягає в переробці та розширенні периферійної схеми, що оточує схему оперативної пам'яті, щоб покращити тестоздатність, яку в народі називають дизайном для тестування. Інші підходи пропонують додати ще більше додаткового обладнання до схеми пам'яті для реалізації вбудованого самотестування (BIST). Але переваги цих модифікацій конструкції пам'яті часто компенсуються накладними витратами, які вони вводять. Ці накладні витрати є функцією розміру пам'яті з точки зору додаткового обладнання, тому ми можемо обґрунтувати його наявність для великого обсягу пам'яті. Основною метою цих підходів є скорочення часу тестування пам'яті, який експоненціально зростає з розміром пам'яті.

1.3 Несправності секцій пам'яті

Моделювання помилок, ймовірно, є одним із найважливіших елементів тестування пам'яті та аналізу помилок. Оскільки фізичне дослідження пам'яті зазвичай надто складне, інженери-випробувачі вдаються до моделей функціональних несправностей (FFM), щоб побудувати ефективні алгоритми функціонального тестування. Функціональне тестування робить доказ повноти тесту та відсутності надлишковості логічною проблемою, якою

легко керувати та автоматизувати. Моделювання помилок пам'яті має довгу історію. Рання робота над FFM для RAM була виконана Thatte і Abrahm [Thatte et al., 1977], а потім Ван де Гуром на початку 1980-х років [Ven de Goor et al., 1990]. Ці публікації є основою для більш ніж 20 років роботи з визначення різних типів FFM і відповідних алгоритмів тестування. Цей величезний обсяг роботи був мотивований безперервним вдосконаленням кремнієвої технології, що призвело до підвищеної чутливості до дефектів виготовлення та стресу навколишнього середовища. Враховуючи цю передумову, повний опис усіх запропонованих FFM та пов'язаних з ними механізмів несправностей був би надто довгим, щоб поміститися в одну главу книги. Більше того, постійне вдосконалення технології призвело б до того, що вона за короткий час застаріла. З цієї причини цей розділ зосереджуватиметься на основних концепціях і теоріях, необхідних для розуміння та роботи з FFM пам'яті, представляючи лише скорочений набір добре відомих FFM. Усі несправності було поділено на такі категорії.

1. Одноклітинні:
- Stuck-At fault (SAF);
- Stuck-open fault (SOF);
- Transition fault (TF);
- Address decoding fault (ADF).
2. Двоклітинні - Coupling fault (CF).
3. Data retention fault (DRF).
4. N-клітинні - Neighborhood pattern sensitive fault (NPSF).
5. Linked faults (LFs).

1.3.1 Одноклітинні помилки

Stuck-At fault (SAF). Помилка застрягання може бути описана таким чином: логічне значення комірки або рядка із застряглою коміркою завжди дорівнює 0 або 1. Вона завжди перебуває в стані 0 або в стані 1 і не може

бути змінена на протилежний стан. Тест, який має виявляти та знаходити всі застряглі несправності, має задовольняти таку вимогу: з кожної комірки чи рядка мають бути прочитані 0 та 1.

Stuck-open fault (SOF). Ця несправність означає, що до комірки неможливо отримати доступ, можливо, через відкриту лінію слова. Коли операція зчитування виконується на комірці, диференціальний підсилювач вимірювання (SA) повинен визначити різницю напруги між бітовими лініями цієї комірки. У випадку SOF обидві бітові лінії матимуть однаковий рівень напруги; отже, вихідне значення, створене підсилювачем чутливості, залежить від способу його реалізації. Якщо SA є комбінаційним або якщо SA має лише один вхід, він передасть визначене логічне значення на вихідний висновок. У цьому випадку несправність із зависанням у відкритому стані буде виявлено так, ніби це була несправність із застряганням. Однак деякі конструкції SA включають засувку на шляху зчитування. Тоді SOF може призвести до того, що засув не оновлюється, оскільки різниця напруги між бітовими лініями надто мала. Попереднє вихідне значення створюється як вихідне значення для SOF. Було припущено, що повинен існувати елемент, у якому значення x і значення ~x зчитуються з клітинки, і інший, або, можливо, той самий елемент, де значення ~x і значення x зчитуються з клітинки.

Transition fault (TF). Окремим випадком SAF є перехідна помилка (TF). Комірка або лінія, яка не може виконати 0 → 1 або 1 → 0 перехід, коли виконується запис, називається такою, що містить помилку переходу вгору. TF можна розглядати як тригер типу встановлення/скидання (S/R) із SAF на вході налаштування чи скидання (Рисунок 1.4). Однак помилки переходу не можна розглядати як помилки SAX, оскільки інші помилки, такі як помилки зв’язку, можуть повернути комірку в стан X. Таким чином, тестування для TF передбачає дещо складніший алгоритм.

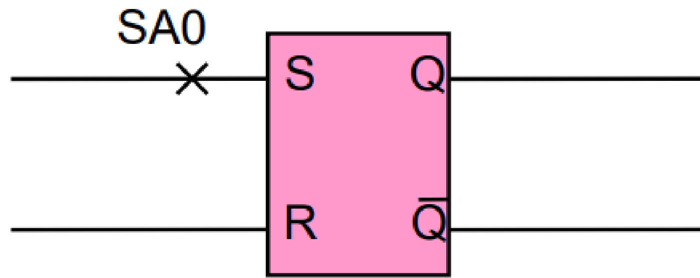


Рисунок 1.4 – Тригер як модель перехідної несправності

Кожна комірка повинна пройти зміну стану 0 → 1 та 1 → 0, після яких комірка повинна бути прочитана перед виконанням будь-яких подальших переходів.

Несправності декодера адреси (ADF). Функціональні несправності, які виникають у дешифраторі адреси, можуть бути такими:

1. З певною адресою доступ до жодної комірки не буде здійснюватися;
2. Певна комірка буде недоступна;
3. З певною адресою здійснюється доступ до кількох комірок одночасно;
4. До певної комірки можна отримати доступ за кількома адресами.

Оскільки комірок стільки, скільки адрес, жодна з перерахованих вище помилок не може виникати окремо. Коли виникає помилка 1, має виникнути або помилка 2, або помилка 3. З несправністю 2 виникає щонайменше помилка 1 або помилка 4; з несправністю 3, принаймні несправністю 1 або 4; з несправністю 4, несправністю 2 або 3. Ці чотири комбінації несправностей показані на рисунку 1.8.

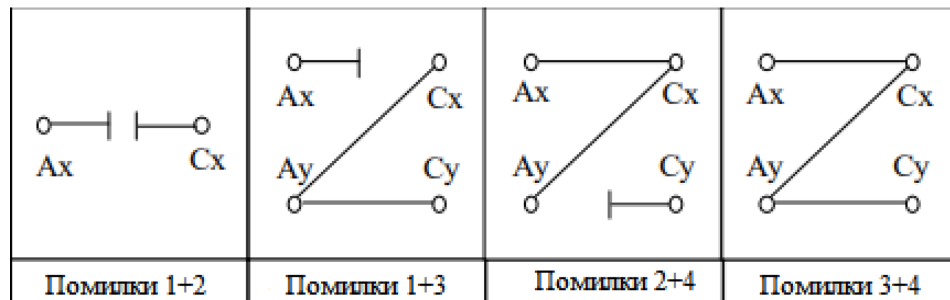


Рисунок 1.5 – Комбінації несправностей декодера адреси

1.3.2 Двоклітинні помилки

Несправність зчеплення (CF) є двоклітинної несправністю. Воно залучає дві клітини і визначається таким чином: операція запису, яка генерує перехід 0 -> 1 або 1 -> 0 в одній клітинці, змінює вміст іншої клітинки. У свою чергу ця категорія поділяється на такі категорії.

1. Інверсійна несправність (CFin): перехід в одній клітинці інвертує вміст другої клітинки. Це можна розглядати як тригер D-типу з додатковим входом тактової частоти Cc (через несправність зв'язку інверсії) на додаток до нормального входу синхронізації Cn і виходу ~Q, прив'язаного до входу D, як показано на рисунку 1.5. Помилка зв'язку інверсії призведе до помилкового введення тактової частоти тригера через Cc, що спричиняє те, що вихід тригера приймає зворотнє значення.

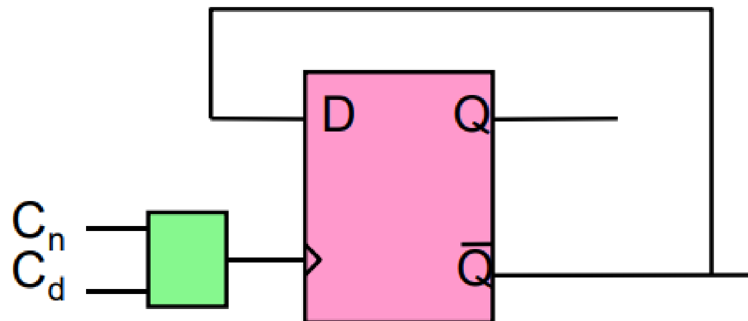


Рисунок 1.6 – Тригер як модель несправності інверсійного зв'язку

2. Помилка ідемпотентного зв'язку (CFid): перехід в одній комірці змушує вміст другої комірки досягати певного значення, 0 або 1. Несправність ідемпотентного зв'язку можна розглядати як тригер типу S/R з вентилем АБО в рядку Set або Reset, як зображено на рисунку 1.6. Sn – це нормальний вхідний сигнал, тоді як Sc – це небажаний вхідний сигнал через з'єднання з одним або кількома іншими тригерами. Якщо не зазначено інше, ідемпотентна 2-зв'язкова помилка мається на увазі, коли згадується помилка зв'язку або CF. Припускається, що ідемпотентний збій зв'язку більш

ймовірний, ніж інверсійний збій зв’язку, оскільки примусове встановлення певного значення більш імовірне, ніж примусове інверсування.

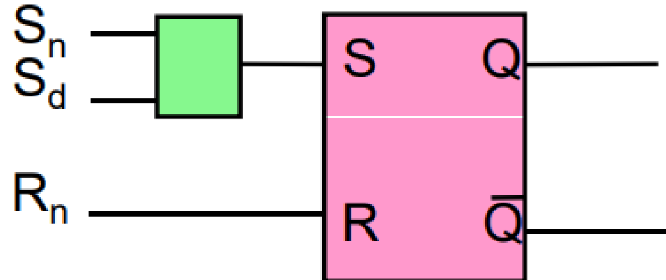


Рисунок 1.7 – Тригер як модель ідемпотентної несправності зв’язку

3. Помилка з’єднання стану (CFst): стан 0 або 1 в одній комірці змушує вміст другої комірки мати певне значення 0 або 1. Його можна розглядати як тригер D-типу з вентилем АБО/І в лінії даних (D) як зображено на воротах у лінії даних (D), як зображено на наступному рисунку 1.7. Dn – це звичайний набір вхідних сигналів, тоді як Dd – небажаний набір inp pg ut через з’єднання з одним чи кількома іншими тригерами.

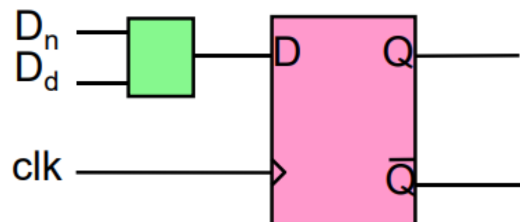


Рисунок 1.8 – Тригер як модель помилки з’єднання стану

4. Помилка сполучення (CFds): операція (запис або читання), виконана на комірці-агресорі, змушує комірку-жертву перейти в заданий логічний стан. Будь-яка операція, виконана на агресорі, вважається операцією сенсибілізації (читання, перехідний запис або запис без переходу).

5. Transition coupling fault (CFtr): стан комірки-агресора спричиняє збій операції запису переходу, виконаної на комірці-жертві. Ця помилка

сенсибілізується операцією запису на клітинку-жертву, поки агресор знаходиться в заданому стані.

6. Помилка руйнівного зв’язку при записі (CFwd): операція запису без переходу, яка виконується на клітині-жертві, коли клітина-агресор перебуває в заданому стані, призводить до переходу самої клітини.

7. Помилка руйнівного зв’язку зчитування (CFrd): операція зчитування, що виконується на клітині-жертві, коли клітина-агресор перебуває в заданому стані, знищує дані, що зберігаються в клітині-жертві.

8. Неправильна помилка зв’язку читання (CFir): операція читання, виконана на клітинці-жертві, повертає неправильне логічне значення, поки агресор перебуває в заданому стані.

9. Оманлива помилка зчитування з деструктивним зв’язком (CFdr): операція читання, виконана на клітинці-жертві, повертає правильне логічне значення та змінює вміст жертви, поки агресор перебуває в заданому логічному стані.

1.3.3 Усі інші помилки

Data retention fault (DRF). Помилка збереження даних (DRF) виникає, коли комірка не зберігає своє логічне значення через деякий період часу. DRF може бути викликана зламаним (відкритим) підтягуючим пристроєм у комірці. Тоді струми витоку призведуть до того, що вузол із зламаним підтягуючим пристроєм втратить свій заряд, що спричинить втрату інформації, якщо логічне значення було збережено в комірці, яка потребувала високої напруги у відкритому вузлі. Можна розпізнати дві різні DRF (обидва можуть одночасно бути присутніми в одній комірці): 1 -> 0 та 0 ->1. Коли обидва присутні в одній клітинці, клітинка поводиться так, ніби вона містить SOF. Таким чином, тестові розширення для SOF, якщо SA є непрозорим для SOF, повинні бути частиною тесту для DRF.

N-клітинні несправності. Несправність, чутлива до шаблону сусідства (NPSF). N-зв'язані несправності представляють моделі несправностей, де n різних комірок пам'яті задіяні в механізмі несправностей (f-клітин=n). Їх зазвичай називають дефектами, чутливими до шаблону (PSF). Загалом на вміст комірки i (або на її здатність змінювати свій стан) впливає вміст усіх інших комірок пам'яті або операції, що над ними виконуються. Помилка, чутлива до шаблону, є найзагальнішим визначенням несправності n-зв'язку, у якій n дорівнює розміру пам'яті.

У більш реалістичній ситуації зазвичай розглядаються так звані шаблони сусідства, чутливі до дефектів (NPSF), у яких зменшений набір клітин, просторово розташованих у суміжних положеннях, відповідає за механізм дефектів. Околиця – це загальна кількість клітинок у цьому наборі. Традиційно клітинку-жертву в цьому контексті називають базовою клітинкою, тоді як клітинки-агресори називають видаленою околицею.

У PSF околиця може бути будь-де в пам'яті, тоді як у NPSF околиця має бути в одній позиції навколо базової комірки. Ці типи моделей несправностей особливо вказуються при роботі з DRAM високої щільності через зменшену ємність комірки пам'яті.

NPSF можна класифікувати за трьома підтипами щодо різної поведінки несправностей: статичний NPSF (SNPSF), пасивний NPSF (PNPSF) і активний NPSF (ANPSF). Детальна поведінка помилок трьох підтипів підсумовується таким чином: SNPSF – стан базової комірки примусово змінюється на 0 або 1, поки видалена околиця перебуває в певному стані; PNPSF – базова комірка не може зазнати переходу від 0 до 1 або з 1 до 0, поки видалена околиця перебуває в певному стані; ANPSF – стан базової комірки примусово змінюється на 0 або 1, поки одна комірка видаленої околиці зазнає переходу або, а інші клітинки видаленої околиці перебувають у певному стані. Якщо розглядається лише виявлення NPSF, тестовий алгоритм повинен записати кожну базову комірку з 0 (1) і застосувати всі

можливі шаблони у відповідному видаленому околі, а потім зчитати базову комірку.

Загалом розглядаються два типи шаблонів сусідства: Тип 1, що включає чотири видалені клітини сусідства, і Тип 2, включаючи 8 видалених клітинок сусідства. Модель типу 2 є більш складною та дозволяє моделювати ефекти діагонального зв'язку в матриці пам'яті. Рисунок показує два типи сусідства.

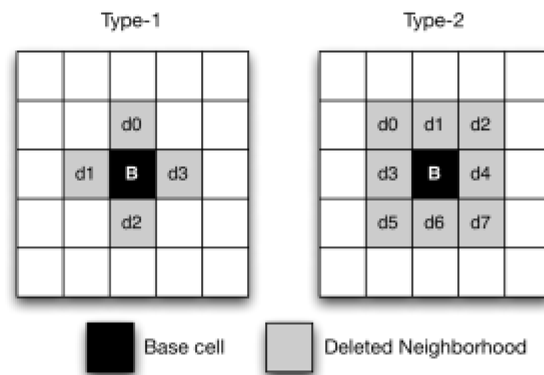


Рисунок 1.9 – Тип 1 та тип 2 NPSF

Linked Faults (LFs). Пов'язані несправності можуть виникати між несправностями одного типу або між несправностями різних типів. SAF завжди виявляється, навіть якщо TF або CF пов'язані з SAF. Отже, ми повинні розглянути наступні пов'язані помилки в масиві пам'яті: TF-CF, CFid-CFid, CFid-CFin, CFin-CFin. Ці пов'язані розломи можуть включати не лише два, але будь-яку кількість розломів цих типів.

1.4 Існуючі алгоритми тестування

Використані позначення:

- rx: операція читання x;
- wx: операція запису x;
- ⇑ : збільшення послідовності адресації (від 0 до n);
- ⇓ : послідовність адресації, що зменшується (від n-1 до 0);

- $\Updownarrow$: або збільшення, або зменшення послідовності адресації.

У цьому розділі розглядатимуться деякі класичні або застарілі алгоритми тестування пам'яті.

GALPAT (патерн галопу), який іноді називають патерном пінг-понгу є тестом складності 4n^2. Перевірка галопу є сильним тестом для більшості недоліків; це повний тест для виявлення та визначення місцезнаходження всіх SAF, TF, ADF та CF. Алгоритм передбачає написання базової клітинки з її доповненням, а потім перевірку кожної іншої клітинки на цілісність даних і повторення алгоритму з інверсією даних.

WALPAT(шляховий шаблон) подібний до GALPAT, за винятком того, що тестова клітинка зчитується один раз, а потім усі інші клітинки. Щоб створити шаблон кроку з програми GALPAT, пропустіть другу операцію читання в тестовому стенді. Шляховий шаблон має час виконання, пропорційний 2N^2. Він перевіряє пам'ять на відкриття та замикання комірки та унікальність адреси.

Тест MARCH складається зі скінченної послідовності елементів March, а елемент March – це скінченна послідовність операцій, що застосовуються до кожної комірки в масиві пам'яті перед переходом до наступної комірки. Операція може складатися із запису 0 у клітинку (w0), запису 1 у клітинку (w1), читання очікуваного 0 із клітинки (r0) і читання очікуваного 1 із клітинки (r1). Після того, як усі операції елемента march були застосовані до даної комірки, вони будуть застосовані до наступної комірки. Адреса наступної комірки визначається декодером адреси.

MARCHING 1/0 – тест складності 14n. Це повний тест для ADF, SAF і TF, але має здатність виявляти лише частину CF. Послідовність випробувань представлена у формулі (1.1).

$$\Uparrow(w0);\Uparrow(r0,w1,r1);\Downarrow(r1,w0,r0);\qquad(1.1)$$

$$\Uparrow(w1);\Uparrow(r1,w0,r0);\Downarrow(r0,w1,r1)$$

MATS – модифікована алгоритмічна тестова послідовність. MATS є найкоротшим тестом MARCH для незв'язаних SAF у масиві комірок пам'яті та логічній схемі читання/запису. Алгоритм може виявити всі помилки для технології типу АБО, оскільки результат читання кількох клітинок розглядається як функція АБО вмісту цих клітинок. Цей алгоритм також можна використовувати для ADF типу AND з використанням тестової послідовності MATS-AND, наведеної нижче. Алгоритм MATS має складність 4n з кращим покриттям помилок порівняно з еквівалентними тестами нуль-один і шаховою дошкою. Алгоритм MATS-OR у формулі (1.2). Алгоритм MATS-AND у формулі (1.3).

$$\Updownarrow(w0); \Updownarrow(r0, w1); \Updownarrow(r1) \quad (1.2)$$

$$\Updownarrow(w1); \Updownarrow(r1, w0); \Updownarrow(r0) \quad (1.3)$$

Тестова послідовність MATS+ виявляє всі SAF і ADF, її часто використовують замість MATS, коли тестована технологія невідома. Алгоритм MATS+ має тестову складність 5n. Алгоритм у формулі (1.4).

$$\Updownarrow(w0); \Uparrow(r0, w1); \Downarrow(r1, w0) \quad (1.4)$$

Тестова послідовність MATS++ – це повна, ненадлишкова та оптимізована тестова послідовність. Він подібний до тесту MATS+, але дозволяє покривати несправності для TF. Рекомендований тест складності тесту 6n для незв'язаних SAF і TF. Алгоритм у формулі (1.5).

$$\Updownarrow(w0); \Uparrow(r0, w1); \Downarrow(r1, w0, r0) \quad (1.5)$$

Тест MARCH C підходить для ADF, SAF, TF і всіх CF. Це тест 11n складності. Алгоритм у формулі (1.6).

$$\Updownarrow(w0);\Uparrow(r0,w1);\Uparrow(r1,w0);$$
$$\Updownarrow(r0);\Downarrow(r0,w1);\Downarrow(r1,w0);\Updownarrow(r0) \quad (1.6)$$

MARCH C- це тестова послідовність, яка є модифікацією тесту MARCH C, реалізована з метою усунення присутньої в ньому надмірності. Виявляє незв'язані ADF, SAF, TF і всі CF. Цей тест складності 10n. Алгоритм у формулі (1.7).

$$\Updownarrow(w0);\Uparrow(r0,w1);\Uparrow(r1,w0);$$
$$\Downarrow(r0,w1);\Downarrow(r1,w0);\Updownarrow(r0) \quad (1.7)$$

Тест MARCH A є найкоротшим тестом для ADF, SAF, пов'язаних CFid, TF, не пов'язаних із CFid, і певних CFin, пов'язаних із CFid. Це повний і безрезервний тест складності 15n. Алгоритм у формулі (1.8).

$$\Updownarrow(w0);\Uparrow(r0,w1,r1,w0,r0,w1);\Uparrow(r1,w0,w1);$$
$$\Downarrow(r1,w0,w1,w0);\Downarrow(r0,w1,w0) \quad (1.8)$$

Тест MARCH B є розширенням тесту MARCH A. Це повний і незайвий тест, здатний виявляти ADF, SAF, пов'язані CFid або CFin. Цей тест складності 17n. Алгоритм у формулі (1.9).

$$\Updownarrow(w0);\Uparrow(r0,w1,w0,w1);\Uparrow(r1,w0,w1);$$
$$\Downarrow r1,w0,w1,w0);\Downarrow(r0,w1,w0) \quad (1.9)$$

Тест MARCH X називається так, оскільки він використовувався без публікації. Цей тест виявляє незв'язані SAF, ADF, TF і CFin. Тест MARCH C – тест складності 6n. Алгоритм у формулі (1.10).

$$\Updownarrow(w0);\Uparrow(r0,w1);\Downarrow(r1,w0);\Updownarrow(r0) \quad (1.10)$$

1.5 Вибір алгоритму для реалізації

Раніше було запропоновано багато типів тестів для ОЗУ. Наразі одне сімейство тестів, яке називається March tests, довело перевагу за часом тестування та простотою алгоритмів. Тести MARCH сьогодні широко використовуються для функціонального тестування технологій SRAM і DRAM. Вони ефективніші ніж старіші тести на основі класичних шаблонів із кращим покриттям помилок (табл.1.2, рис.1.2). Із збільшенням щільності напівпровідникової пам'яті тривають дослідження кращих послідовностей шаблонів і альтернативних стратегій, таких як DFT і BIST. Як видно з рисунку 1.10 TLSNPSFlG2 (тест для виявлення помилок, чутливих до статичних шаблонів сусідства, у сусідстві type1, використовуючи метод двох груп) може виявляти SAF та помилки, чутливі до шаблонів статичного сусідства. На рис. 6 показано, що це невдала модель несправності для SRAM.

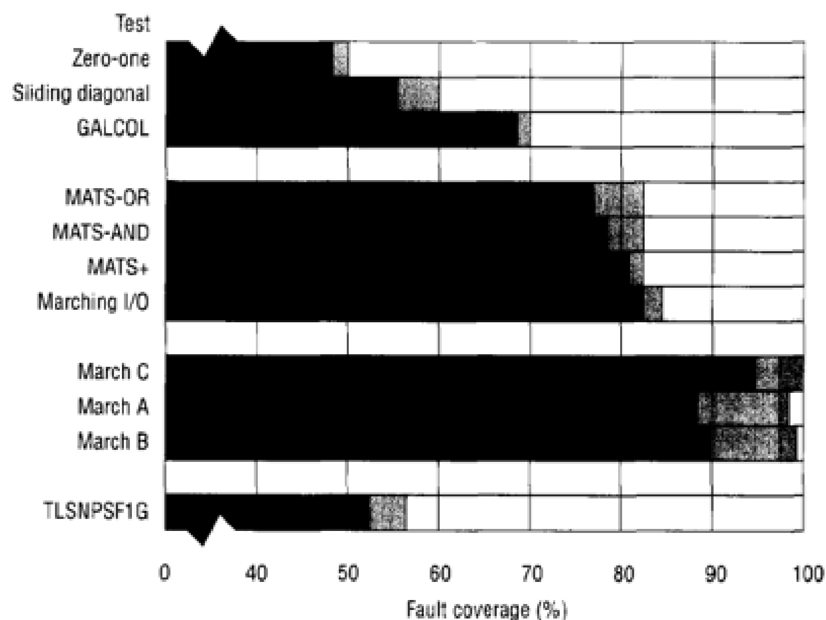


Рисунок 1.10 – Покриття несправностей для різних тестів SRAM

Таблиця 1.2 – Покриття несправностей для MARCH тестів

Fault	MATS++	MARCHX	MARCHY	MARCH C-
SAF's	100%	100%	100%	100%
TF's	100%	100%	100%	100%
SOF's	100%	0.2%	100%	0.2%
AF's	100%	100%	100%	100%
CFin's	75.0%	100%	100%	100%
CFid's	37.5%	50.0%	50.0%	100%
CFst's	50.0%	62.5%	62.5%	100%

Виходячі з даних представлених у таблиці 1.1 можна зробити висновок що алгоритми з часом виконання n^3/2 та n^2 абсолютно непригодні для практичного використання через надмірні часові затрати, алгоритми з часом виконання n*log(n) вкрай ситуативні, бо теж непригодні для тестування великих обсягів пам'яті, тому вважаю доцільним тільки зробити фокус на алгоритмах які виконуються за n тактів, а саме на алгоритмі March C-. Алгоритм покриває більшість базових несправностей (рисунок 1.2), які можуть виникати у SRAM. Зв'язані несправності (LF) складають дуже маленький процент від усіх помилок, та для покриття кожної з них потрібно окремо конструювати алгоритм. Для покриття таких несправностей як DRF та SOF алгоритм March C- може бути без проблем модифікований.

March C- (формула 1.7) можна модифікувати для виявлення SOF шляхом розширення операції (r0,w1) за допомогою операції «r1» і операції (r1,w0) за допомогою операції «r0». Тоді вони матимуть вигляд (r0,wl,r1) і (rl,w0,r0).

Будь-який маршовий тест можна розширити, щоб охопити також DRF. Виявлення DRF вимагає переведення комірки пам'яті в один із її логічних станів. Повинен пройти певний час, поки DRF розвивається (струми витоку повинні розрядити відкритий вузол комірки SRAM). Після цього вміст комірки перевіряється. Цей тест потрібно повторити з інверсним логічним значенням, збереженим у комірці, щоб перевірити наявність DRF через

відкрите з’єднання в іншому вузлі комірки. Час очікування залежить від кількості заряду, що зберігається в конденсаторі вузла, і величини струму витоку (який важко визначити). Емпіричні результати показують, що час очікування (званий затримкою) 100 мс є достатнім для досліджуваних клітин SRAM.

Недоліком цієї процедури для алгоритму March C- є те, що може виникнути маскування несправності; наприклад, CF може не бути виявлено, коли пов’язана комірка також має DRF, оскільки DRF може маскувати CF.

2 РОЗРОБКА СИСТЕМИ ТЕСТУВАННЯ

2.1 Концепція BIST

March тести є кращим методом тестування оперативної пам’яті за допомогою зовнішніх тестерів або за допомогою вбудованих рішень для самоперевірки (BIST).

Вбудоване самотестування, або BIST – це техніка розробки додаткових функцій апаратного та програмного забезпечення в інтегральних схемах, щоб дозволити їм виконувати самотестування, тобто тестування власної роботи (функціонально, параметрично або обох) за допомогою власних схем, таким чином зменшення залежності від зовнішнього автоматизованого випробувального обладнання. BIST – це техніка Design-for-Testability (DFT), оскільки вона робить електричне тестування мікросхеми простішим, швидшим, ефективнішим і менш дорогим. Концепція BIST застосовна практично до будь-якого типу схеми, тому її реалізація може варіюватися настільки ж широко, як і різноманітність продуктів, які вона обслуговує. Основна ідея BIST, у своїй найпростішій формі, полягає в тому, щоб спроектувати схему таким чином, щоб схема могла перевірити себе та визначити, чи є вона безвідмовною чи несправною. Це зазвичай вимагає, щоб додаткові схеми та функціональні можливості були включені в дизайн схеми для полегшення функції самотестування. Ця додаткова функціональність повинна бути здатна генерувати тестові шаблони, а також забезпечувати механізм для визначення, чи відповідають вихідні відповіді схеми, що перевіряється, тестовим шаблонам відповіді.

Загалом, два підходи BIST існують для RAM: на основі FSM та на основі ПЗУ. Підход на основі FSM легше описується, конфігується та модифікується тому він був обран для реалізації.

Модель представлена на рисунку 2.1. Базові компоненті BIST включають у себе:

- контролер: генерує керуючі сигнали для генератора тестових шаблонів і пам'яті, що тестується;
- генератор тестових шаблонів: генерує необхідні тестові шаблони та сигнали читання/запису;
- компаратор: порівняння реакції RAM на тест з еталонною.

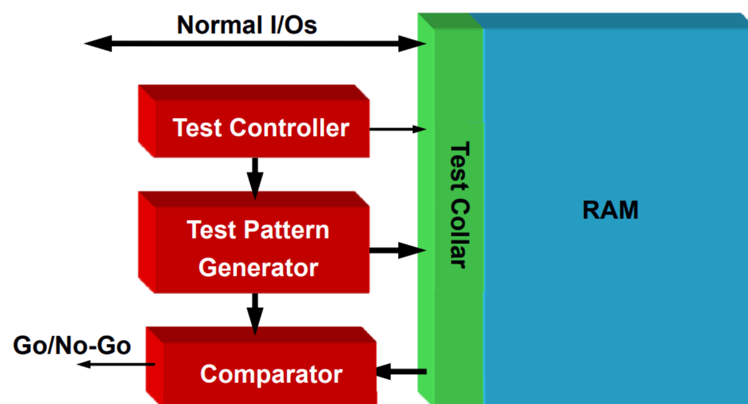


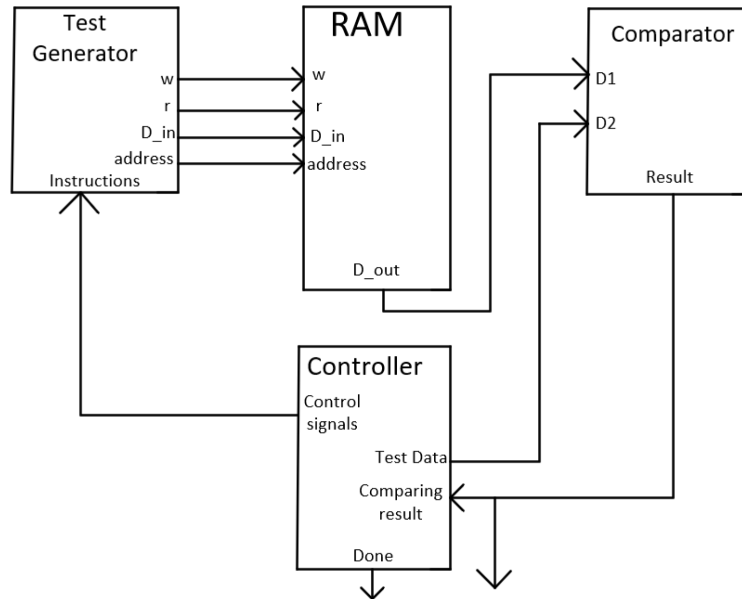
Рисунок 2.1 – Модель BIST

2.2 Моделювання системи

Спираючись на представлену модель RAM (рисунок 1.1) та на наведену модель BIST (рисунок 2.1) можно побувати більше детальну схему взаємодії між компонентами системи, яка повинна сигнали, які будуть передаватись між цими компонентами. Опис схеми наведеної у рисунку 2.2.

1. Генератор тестів. Вхід: інструкції від контроллеру. Вихід: сигнал читання/запису, дані для запису у RAM, адреса комірки у RAM.
2. RAM. Вхід: сигнал читання/запису, дані для запису, адреса комірки. Вихід: дані для зчитування.
3. Компаратор. Вхід: дві строки даних для порівняння (D1,D2). Вихід: Результат порівняння(Result).

4. Контролер. Вхід: результат порівняння (Comparing result). Вихід: результат відпрацювання тестів (Done), результат порівняння (Comparing result), еталонні данні для порівняння (Test Data), вказівки для генератора тестів (Control signals).



Рисунку 2.2 – Поєднання моделі BIST та RAM

Для формалізації поняття алгоритмів використовують автомати Тьюринга та Посту, що дозволяють виділити елементарні операції. Строго виконуючи ці операції та спостерігаючи процес їх виконання за допомогою емуляторів даних машин, можна отримати очікуваний результат за кінцеве число кроків. Однак машини Т'юрінга і Поста мають дуже обмежений набір команд, відсутні основні арифметичні операції і, крім того, вони практично нереалізовані внаслідок нескінченних стрічок, оскільки пам'ять обмежена. У цих машинах головку чи каретку можна переміщати ліворуч чи праворуч лише одну позицію, що ускладнює моделювання алгоритмів тестів. Існують також багатострічкові машини Тьюринга, що мають кілька стрічок і кілька головок, кожна з яких оглядає тільки свою стрічку, але стрічки, що застосовуються, нескінченні вправо, тому їх неможливо реалізувати практично.

До недоліків програмування алгоритмів на обчислювальних машинах з архітектурою фон Неймана можна віднести той факт, що спочатку необхідно запрограмувати код адреси, потім код даних та код операції для пристрою, що тестується, при цьому частота звернення до об'єкта зменшується та знижується ефективність тестування.

Для скорочення трудомісткості синтезу та налагодження алгоритмів та програм тестів діагностування мікросхем напівпровідникової пам'яті пропонується система тестового генератора (рис. 2.2), яка містить пристрій управління УУ і чотири головки запису/зчитування Г0-Г3. Головки можуть записувати або зчитувати з осередків символи алфавіту А і переміщатися по осі X або У, або по осях X і У одночасно вліво або вправо на 1,2, ..., р позицій. Переміщення головок незалежні, вони можуть оглядати різні осередки або один і той самий осередок.

В якості пам'яті пропонується квадратна матриця розміром 256 комірок, по 1 слову інформації розміром 1 байт в кожній, з шиною даних розміром 1 байт. Символи, що записуються або зчитуються з осередків пам'яті, належать алфавіту, який вибирає користувач. Для матриці осередків пам'яті встановлено початковий $a_g = 0$ і кінцевий $a_n = 255$ адреси, з яких зазвичай починається тестування та закінчується виконання тесту. Пристрій управління системи має кінцеве число станів, що визначається складністю алгоритму, що реалізується.

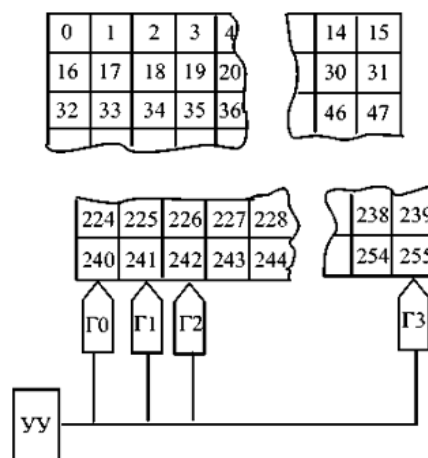


Рисунок 2.2 – Структура системи тестового генератора

Формат команд системи представимо у такому вигляді:

$$Q_i , S_r^i \rightarrow S_r^j , C_r , p_r , Q_j \qquad (2.1)$$

де Q_i - поточний стан системи; S_r^i - символ що зберігався в комірці, яку оглядає r-я головка в даний момент часу; $r = \overline{0, l-1}$ - номер голівки; l - число головок запису (зчитування); S_r^j - новий символ, що записується в комірку головкою; $C_r\ e\{W_r, R_r, A_r,\}$ - коди операцій для кожної голівки; W_r і R_r - операції запису символу в комірку і зчитування з комірки, що сканується r головкою; A_r - операція порівняння символу, зчитаного з комірки, з еталонним значенням; p_r - константа зміни позиції r-ї головки; Q_j - новий стан системи.

Алфавіт може містити будь-який символ з символів клавіатури:

$$S_r^i = S_r^i = \{U, X, 0,1 \dots ,255, Z\} \qquad (2.2)$$

де U - невизначений стан; X - байдужий стан; Z - символ з клавіатури, в комірку матриці заноситься його ASCII-код.

При зміні позицій головками може виконуватися одна мікрооперація з наступного набору:

$$a_r := a_r + p;\ a_r := a_r - p;\ a_r := a_g;\ a_r := a_n \qquad (2.3)$$

де p - Константа зміни коду адреси. Таким чином, адреса комірки, яку оглядає головка, може збільшуватися або зменшуватися на позицій.

2.2 Синтез алгоритму теста MARCH C

Спочатку всі головки встановлюємо в положення, коли вони оглядають одну і ту ж комірку з початковою адресою, потім переміщуємо головку з номером 1 на одну позицію вправо, з номером 2 - на дві позиції, з номером 3 - на три позиції (формула 2.4).

$$Q_0, U_r, \rightarrow \forall r, r = \overline{0,3}\ \ a_r := a_g, Q_1, \quad (2.4)$$
$$Q_1, U_r, \rightarrow \{a_1 \coloneqq a_1 = 1, a_2 \coloneqq a_2 + 2, a_3 \coloneqq a_3 + 3\}, Q_2.$$

Записуємо фон нулів у всі осередки, які скануємо зі зростанням коду адреси (формула 2.5).

$$Q_2, U_r \rightarrow \forall r, r = \overline{0,3} \begin{cases} if\ a_3 \neq a_n\ then\ W_r(0), a_r \coloneqq a_r + 4, Q_2; \\ else\ W_r(0), a_r \coloneqq a_r + 4, Q_3; \\ end\ if. \end{cases} \quad (2.5)$$

Встановлюємо всі головки у вихідне положення та переміщуємо дві головки на одну позицію праворуч (формула 2.6).

$$Q_3, 0_r, \rightarrow \forall r, r = \overline{0,3}\ \ a_r := a_g, Q_4, \quad (2.6)$$
$$Q_4, 0_r, \rightarrow \{a_2 \coloneqq a_2 + 1, a_3 \coloneqq a_3 + 1\}, Q_5.$$

Відповідно до алгоритму, головки, що мають номери 0 і 2, зчитують символи з осередків матриці і порівнюють їх з еталонними значеннями, а головки з номерами 1 і 3 записують в комірки число 255. Потім всі головки переміщуються праворуч на дві позиції (формула 2.7).

$$Q_5, 0_r \rightarrow \begin{cases} if\ a_3 \neq a_n\ then\ F_5(Q_5, 0),\ Q_5; \\ else\ F_5(Q_5, 0),\ Q_6; \\ end\ if. \end{cases} \quad (2.7)$$

Відповідно до функції $F_5(Q_5,0)$ виконуються такі дії (формула 2.8).

$$F_5(Q_5,0) = \begin{cases} R_0(S_0);\ if\ S_0 = 0\ then\ a_0 \coloneqq a_0 + 2; \\ else\ a_0 \coloneqq a_0 + 2, Q_{stop}; \\ end\ if; \\ W_1(255), a_1 \coloneqq a_1 + 2; \\ R_2(S_2);\ if\ S_2 = 0\ then\ a_2 \coloneqq a_2 + 2; \\ else\ a_2 \coloneqq a_2 + 2, Q_{stop}; \\ end\ if; \\ W_3(255), a_3 \coloneqq a_3 + 2. \end{cases} \tag{2.8}$$

Встановлюємо всі головки в положення, коли вони оглядають комірку з початковою адресою, потім дві головки з номерами 2 і 3 зміщуємо на одну позицію праворуч (формула 2.9).

$$\begin{gathered} Q_6, 255_r \rightarrow \forall r, r = \overline{0{,}3}\ \ a_r := a_g, Q_7, \\ Q_7, 255_r \rightarrow \{a_2 \coloneqq a_2 + 1, a_3 \coloneqq a_3 + 1\}, Q_8. \end{gathered} \tag{2.9}$$

Головки виконують дії згідно з алгоритмом тесту (формула 2.10).

$$Q_8, 255_r \rightarrow \begin{cases} if\ a_3 \neq a_n\ then\ F_8(Q_8, 255), Q_8; \\ else\ F_8(Q_8, 255),\ Q_9; \\ end\ if. \end{cases} \tag{2.10}$$

Відповідно до функції F8(Q8,255) виконуються такі дії (формула 2.11).

$$F_8(Q_8,255) = \begin{cases} R_0(S_0);\ if\ S_0 = 255\ then\ a_0 \coloneqq a_0 + 2; \\ else\ a_0 \coloneqq a_0 + 2, Q_{stop}; \\ end\ if; \\ W_1(0), a_1 \coloneqq a_1 + 2; \\ R_2(S_2);\ if\ S_2 = 255\ then\ a_2 \coloneqq a_2 + 2; \\ else\ a_2 \coloneqq a_2 + 2, Q_{stop}; \\ end\ if; \\ W_3(0), a_3 \coloneqq a_3 + 2. \end{cases} \tag{2.11}$$

Для сканування комірок із зменшенням коду адреси встановлюємо головки в позиції, коли вони оглядають комірку з кінцевою адресою. Зміщуємо дві головки з номерами 2 та 3 на одну позицію вліво (формула 2.12).

$$Q_9, 0_r, \rightarrow \forall r, r = \overline{0{,}3}\ \ a_r := a_n, Q_{10}. \tag{2.12}$$
$$Q_{10}, 0_r, \rightarrow \{a_2 := a_2 - 1, a_3 := a_3 - 1\}, Q_{11}.$$

Головки виконують дії згідно з алгоритмом зі зменшенням коду адреси (формула 2.13).

$$Q_{11}, 0_r \rightarrow \begin{cases} if\ a_3 \neq a_n\ then\ F_{11}(Q_{11}, 0), Q_{11}; \\ else\ F_{11}(Q_{11}, 0),\ Q_{12}; \\ end\ if. \end{cases} \tag{2.13}$$

Відповідно до функції F_{11} (Q_{11},0) виконуються такі дії (формула 2.14).

$$F_{11}(Q_{11}, 0) = \begin{cases} R_0(S_0);\ if\ S_0 = 0\ then\ a_0 := a_0 - 2; \\ else\ a_0 := a_0 - 2, Q_{stop}; \\ end\ if; \\ W_1(255), a_1 := a_1 - 2; \\ R_2(S_2);\ if\ S_2 = 0\ then\ a_2 := a_2 - 2; \\ else\ a_2 := a_2 - 2, Q_{stop}; \\ end\ if; \\ W_3(255), a_3 := a_3 - 2. \end{cases} \tag{2.14}$$

Знову встановлюємо головки положення, коли всі вони оглядають комірку з кінцевою адресою, і зміщуємо головки з номерами 2 і 3 на одну позицію вліво (формула 2.15).

$$Q_{12}, 255_r, \rightarrow \forall r, r = \overline{0{,}3}\ \ a_r := a_n, Q_{13}. \tag{2.15}$$
$$Q_{13}, 255_r, \rightarrow \{a_2 := a_2 - 1, a_3 := a_3 - 1\}, Q_{14}.$$

Виконуються дії згідно з алгоритмом зі зменшенням коду адреси (формула 2.16).

$$Q_{14}, 255_r \rightarrow \begin{cases} if\ a_3 \neq a_n\ then\ F_{14}(Q_{14}, 255), Q_{14}; \\ else\ F_{14}(Q_{14}, 255),\ Q_{15}; \\ end\ if. \end{cases} \tag{2.16}$$

Відповідно до функції $F_{14}(Q_{14},255)$ виконуються такі дії (формула 2.17).

$$F_{14}(Q_{14}, 255) = \begin{cases} R_0(S_0);\ if\ S_0 = 255\ then\ a_0 := a_0 - 2; \\ else\ a_0 := a_0 - 2, Q_{stop}; \\ end\ if; \\ W_1(0), a_1 := a_1 - 2; \\ R_2(S_2);\ if\ S_2 = 255\ then\ a_2 := a_2 - 2; \\ else\ a_2 := a_2 - 2, Q_{stop}; \\ end\ if; \\ W_3(0), a_3 := a_3 - 2. \end{cases} \tag{2.17}$$

Встановлюємо головки в положення, коли вони оглядають комірку з початковою адресою та змінюємо позиції трьох головок (формула 2.18).

$$Q_{15}, 0_r, \rightarrow \forall r, r = \overline{0,3}\ \ a_r := a_g, Q_{16}. \tag{2.18}$$

$$Q_{16}, 0_r, \rightarrow \{a_1 := a_1 + 1, a_2 := a_2 + 2, a_3 := a_3 + 3\}, Q_{17}.$$

Зчитуємо дані з осередків пам'яті зі збільшенням коду адреси та порівняємо лічені дані з еталоном (формула 2.19).

$$Q_{17}, 0_r \rightarrow \begin{cases} if\ a_3 \neq a_n\ then\ F_{17}(Q_{17}, 255), Q_{17}; \\ else\ F_{17}(Q_{17}, 255),\ Q_{19}; \\ end\ if. \end{cases} \tag{2.19}$$

Відповідно до функції $F_{17}(Q_{17},0)$ виконуються такі дії (формула 2.20).

$$F_{17}(Q_{17},0) = \forall r, r = \overline{0,3} \begin{cases} R_r(S_r); \\ if\ S_r = 0\ then\ a_r := a_r + 4; \\ else\ a_r := a_r + 4, Q_{stop}; \\ end\ if. \end{cases} \qquad (2.20)$$

Наприкінці програми містяться команди видачі результатів тестування у вигляді відповідних повідомлень: $Q_{stop}, X_r \rightarrow !$ або $Q_{19}, 0_r \rightarrow !$, ok.

Синтезований вище алгоритм можна реалізувати за допомогою автоматного програмування, а саме використовуючи концепцію рекурсивного автомату задля підвищеної ефективності виконання, аніж в інших існуючих реалізаціях даного алгоритму.

2.3 Граф переходів автомату

Частина станів з алгоритму наведеного вище можна об’єднати, зменшивши таким чином спільну кількість станів в алгоритмі, це буде потребувати додаткових регістрів, для запису туди даних необхідних для коректного переходу з одного стану в інший. Еквіваленти стану алгоритма March C- та системи, яка буде цей алгоритм виконувати, наведені у таблиці 2.1.

Рисунок 2.3 демонструє зв’язки між станами системи. Сигнали переходів наведені у схемі:

- Q0_c – сигнал визначаючий подальший шлях зі стану Q0;
- Q1_c – сигнал визначаючий подальший шлях зі стану Q1;
- Q3_c – сигнал визначаючий подальший шлях зі стану Q3;
- Q6_c – сигнал визначаючий подальший шлях зі стану Q6;
- Addr – адреса комірки RAM над якою виконується операція запису чи зчитування;
- Comp_res – результат порівняння зчитанного з RAM слова з еталонним;

Таблиця 2.1 – Відповідність між станами

Алгоритм	Система	Алгоритм	Система
Q0	Q0	Q11	Q7
Q1	Q1	Q12	Q6
Q2	Q2	Q13	Q6
Q3	Q0	Q14	Q8
Q4	Q3	Q15	Q0
Q5	Q4	Q16	Q1
Q6	Q0	Q17	Q9
Q7	Q3	Q19	Qend
Q8	Q5	Qstop	Qstop
Q9	Q6		
Q10	Q6		

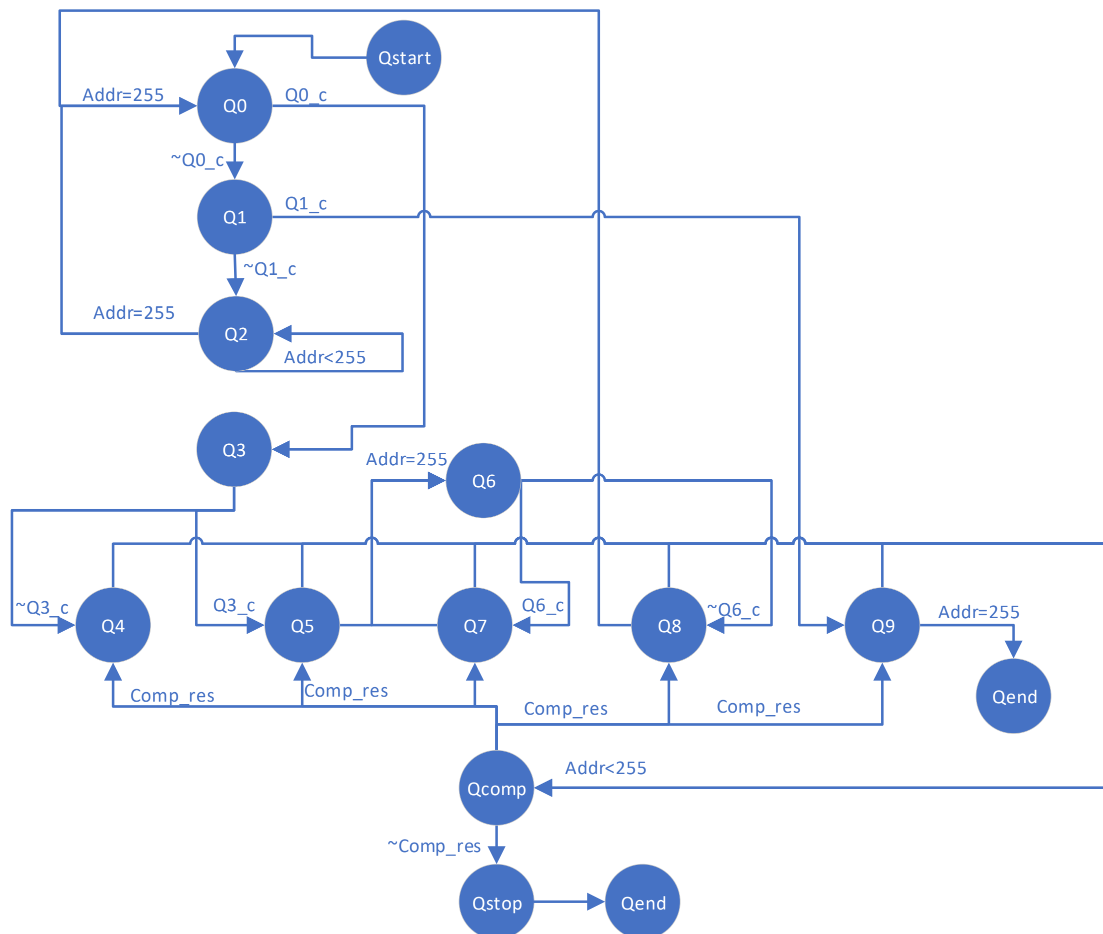


Рисунок 2.3 –Граф переходів керуючого пристрою

3 АВТОМАТНЕ ПРОГРАМУВАННЯ

3.1 Теорія автоматів

Цифрові схеми можна класифікувати як комбінаційні або послідовні. Комбінаційна схема - це така, вихідні значення якої залежать виключно від поточних вхідних значень, тоді як послідовна схема має виходи, які залежать від попередніх станів системи. Отже, перший не має пам'яті, тоді як другий потребує певної пам'яті.

Кінцевий автомат - це техніка моделювання та проектування для послідовних схем. У будь-який момент часу автомат перебуває в одному зі скінченної кількості можливих станів. Для кожного стану повністю визначені як вихідні значення, так і умови переходу в інші стани. Стан зберігається автоматом, і умови переходу зазвичай переоцінюються на кожному (позитивному) фронті синхронізації, тому процедура зміни стану завжди синхронна, оскільки автомат може переходити в інший стан лише тоді, коли годинник тікає.

З точки зору апаратного забезпечення кінцеві автомати можна класифікувати на два типи на основі їх вхідних з'єднань.

1. Автомат Мура. Вхід, якщо він існує, підключається лише до логічного блоку, який обчислює наступний стан.
2. Автомат Мілі. Вхід підключається до обох логічних блоків, тобто для наступного стану та для фактичного виходу.

З апаратної точки зору, на основі типів переходів і характеру виходів, наступним чином.

1. Звичайні (категорія 1) кінцеві автомати (рис.3.1a), складається з машин лише з нерозрахованими переходами та виходами, які не залежать від попередніх (минулих) вихідних значень.

2. Кінцеві автомати з часовим поясненням (категорія 2). Ця категорія, показана на рис.3.1b, вона подібна до категорії 1, за винятком того факту, що один або більше її переходів залежать від часу (тому ці автомати можуть мати усі чотири типи переходів: умовний, часовий, умовно-часовий і безумовний).

3. Рекурсивні (категорія 3) кінцеві автомати (рис.3.1). Можуть мати всі чотири типи переходів, але один або більше виходів залежать від попередніх (минулих) вихідних значень, тому такі виходи мають бути зареєстровані. В стандартній архітектурі виходи виробляються комбінаційним логічним блоком FSM, тому поточні вихідні значення «забуваються» після того, як автомат виходить із цього стану; отже, щоб реалізувати рекурсивний автомат, необхідна якась додаткова пам'ять.

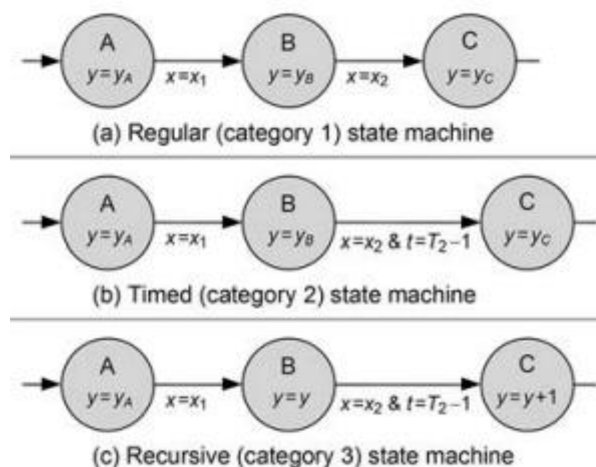


Рисунок 3.1 – Категорії кінцевих автоматів

3.2 Рекурсивні автомати

Рекурсивні кінцеві автомати (RSM), у яких вершини можуть бути або звичайними станами, або можуть відповідати викликам інших кінцевих автоматів потенційно рекурсивним способом, RSM можуть моделювати потік керування в типових послідовних імперативних мовах програмування з рекурсивними викликами процедур.

Рекурсивний кінцевий автомат складається з набору складових машин. Кожен компонент має набір вузлів і блоків (кожен з яких відображається на компонент), чітко визначений інтерфейс, що складається з вузлів входу та виходу, а також ребер, що з’єднують вузли/блоки. Ребро, що входить у блок, моделює виклик компонента, пов’язаного з блоком, а ребро, що залишає блок, відповідає поверненню з цього компонента. Завдяки рекурсії базовий глобальний простір станів є нескінченним і поводиться як система, що проштовхує вниз.

Загальна архітектура автоматів категорії 3 узагальнено на рис. 3.2a. Таймер необов'язковий, але допоміжний регістр є обов'язковим. На цій ілюстрації допоміжний регістр потрібен тільки для сигналу, який виробляє output2, тому для цього виводу додатковий вихідний регістр (рис. 3.2b) ніколи не потрібен (пунктирні лінії вказують на те, що output2 може бути як незареєстрованою, так і зареєстрованою версією outp). Тобто додатковий вихідний регістр потрібен лише для виходів, не оброблених допоміжним регістром. З іншого боку, output1 не зареєстрований, тому залежно від програми для нього може знадобитися додатковий вихідний регістр. Отримані реалізації описані нижче.

Рекурсивний автомат Мура: використовується схема, зображена на рисунку 3.2a, з входом (якщо він існує), підключеним лише до логічного блоку для наступного стану, і з незареєстрованим виходом.

Рекурсивний автомат Мілі: знову використовується схема на рисунку 3.2a, але цього разу з входом, підключеним до обох логічних блоків (для виведення та для наступного стану).

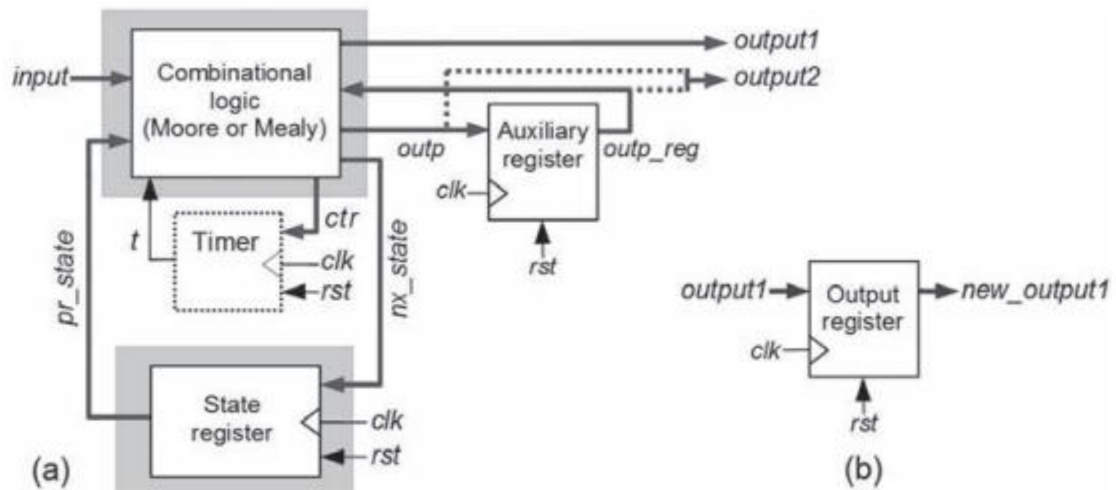


Рисунок 3.2 – Загальна архітектура для автоматів категорії 3

Зразок RSM має три компоненти, як показано на рисунку 3.3. Компонент A1 має 4 вузли, з яких ul і u2 є вузлами входу, а u4 є вузлом виходу, і два блоки, з яких bl відображається на компонент A2, а b2 відображається на A3. Вузли входу та виходу є інтерфейсом керування компонентом, за допомогою якого він може спілкуватися з іншими компонентами. Кінцеві автомати компонентів можуть вважатись процедурами, а ребро, що входить у поле на заданому вході, викликом процедури, пов’язаної з полем із заданими значеннями аргументів. Вузли входу аналогічні аргументам, тоді як вузли виходу моделюють значення, що повертають.

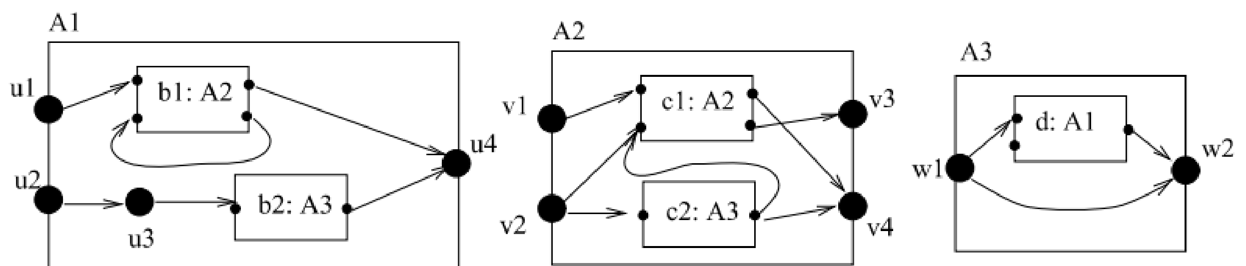


Рисунок 3.3 – Зразок рекурсивного кінцевого автомата

Реалізація алгоритму тестування оперативної пам’яті за допомогою рекурсивного автомату може бути вигідною саме через відсутність необхідності обробляти кожен стан як окремий.

3.3 Порівняння рекурсивного та ітеративного підходу

Загалом можна розглянути два типи рекурсії. Перший (тривіальний) тип дозволяє замінити ітераційну послідовність операцій (тобто цикл) рекурсивним викликом процедури, яка виконує кожну ітерацію циклу та перевіряє умову для виходу з циклу. Другий тип рекурсії є двійковий (N-ary, N>2) пошук, такий як той, що виконується над бінарним (N-ary) деревом. У цьому конкретному випадку послідовність рекурсивних викликів також дає змогу відслідковувати піддерева, які досліджуються під час кроків прямого поширення.

Експерименти з функціями C/C++ показують кращу продуктивність для ітераційного алгоритму приблизно в 1,5 рази. Експерименти з проектами Handel-C/VHDL також демонструють переваги ітераційної реалізації з точки зору як ресурсів FPGA, так і часу виконання (коефіцієнт 1,03/(1,01-1,15) і 0,99/(1,16-1,2, відповідно). Але рекурсивні специфікації можуть мати інші переваги, такі як ясність опису, легкість тестування (налагодження), зміни та обслуговування. Крім того, якщо використовується ієрархічна модульна специфікація, рекурсія не споживає додаткові ресурси.

Попередні дослідження показують, що використання вбудованих блоків пам’яті дозволяє значно зменшити зайняті ресурси FPGA, куди розміщуються стеки викликів. Таким чином, для рекурсивної та ітераційної реалізації кількість зрізів FPGA стає практично однаковою. Для дуже складних алгоритмів модульні рекурсивні реалізації будуть значно кращими.

У деяких потенційні випадках завдяки модульному рекурсивному підходу ми можемо скористатися стратегією «розділяй і володарюй», тобто додаткова складність схеми не так важлива, як ясність алгоритму.

Також важливо додати, що ієрархічні модульні специфікації забезпечують пряму підтримку повторного використання. Це може призвести до скорочення часу розробки та навіть до зменшення апаратних ресурсів у деяких випадках.

3.4 Реалізація рекурсивного автомату на мові VHDL

Це буде розширення до шаблонів автомата Мура 1 і 2 категорії, які позначені на рисунку 3.1. Єдиними відмінностями є ті, які необхідні для включення допоміжного реєстру, обов'язкового для машин категорії 3. Тому фактично такий автомат можна назвати Timed recursive state machine.

Архітектура складається з двох частин: декларативної частини (до початку) та частини операторів (від початку); обидва мають нові елементи для розміщення допоміжного регістру.

Для роботи з допоміжним регістром створюються два сигнали, це показано у лістингу 3.1.

Лістинг 3.1 – Два сигнали для допоміжного регістра

```
21      signal outp, outp_reg: std_logic_vector(...);
```

Передбачається, що є лише один вихід і його потрібно зберегти, але пам'ятайте, що схема може мати кілька виходів, не всі зареєстровані. Фактична кількість допоміжних регістрів визначається кількістю виходів, які залежать від минулих значень.

Включення процесу виведення допоміжного регістру та заміна outp на outp_reg у правій частині рекурсивних рівнянь. Останній усуває рекурсивність, таким чином дозволяючи вихід обчислювати за допомогою комбінаційної схеми. Процес, який реалізує допоміжний регістр у лістингу 3.2.

Лістинг 3.2 – Реалізація допоміжного регістра

```
28      --Auxiliary register:
29      process (clk, rst)
30      begin
31         if (rst='1') then
32            outp_reg <= <initial value>;
33         elsif rising_edge(clk) then
34            outp_reg <= outp;
35         end if;
36      end process;
```

Якщо хтось віддає перевагу, цей процес можна поєднати з процесом для державного реєстру FSM (результат коротший код, але менш дидактичний, без впливу на результат).

Процес, який реалізує секцію комбінаційної логіки автомату на мові VHDL у лістингу 3.3.

Лістинг 3.3 – Комбінаційна логіки автомата

```
42      process (all) --list proc. inputs if "all" not supported
43      begin
44         case pr_state is
45            when A =>
46               outp <= outp_reg;
47               tmax <= T1-1;
48               if <condition> then
49                  nx_state <= B;
50               elsif <condition> then
51                  nx_state <= ...;
52               else
53                  nx_state <= A;
54               end if;
55            when B =>
56               outp <= outp_reg + 1;
57               tmax <= T2-1;
58               if <condition> then
59                  nx_state <= C;
60               elsif <condition> then
61                  nx_state <= ...;
62               else
63                  nx_state <= B;
64               end if;
65            when C =>
66               ...
67         end case;
68      end process;
```

Відмінність тут полягає в тому, що outp_reg, замість самого outp, з'являється в правій частині (спочатку) рекурсивних рівнянь (рядки 46 і 56).

Цікавим аспектом FSM категорії 3 є те, що допоміжний регістр також може відігравати роль вихідного регістра (для безглюкової та/або конвеєрної конструкції). Для цього ми просто надсилаємо outp_reg замість outp.

Модель автомата Мура більш інтуїтивне, завдяки тому що всі розрахункові операції залежать тільки від стану, у якому перебуває система, тому програмування було виконане на основі цієї моделі.

Зациклені рекурсивні автомати. Так як генерація адреси та даних виконується в одному стані системи, а порівняння зчитаних даних із еталонними у іншому потрібні кілька покажчиків (лічильників) необхідних для реалізації такої FSM. Загальна проблема представлена на рисунку 3.4a. Автомат повинен залишатися лише один період синхронізації в кожному стані, але цикл повинен повторюватися N_{ab} разів. Рішення показано на рисунку 11.4b. Лічильник (k) збільшується лише в стані B, зберігаючи своє значення в стані A.

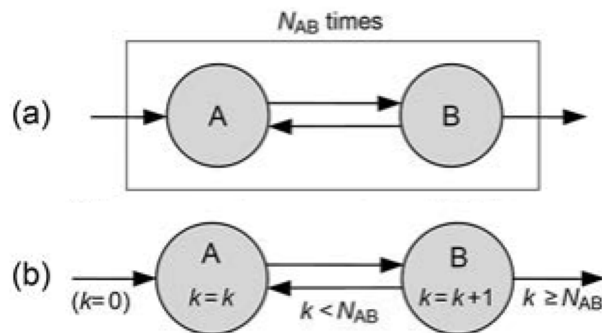


Рисунок 3.4 – Зациклені рекурсивні автомати

4 ПРОГРАМУВАННЯ ТА ІМПЛЕМЕНТАЦІЯ

4.1 Платформа для імплементації

Платформою для реалізації системи тестування пам'яті пропонується обрати FPGA. Програмована вентильна матриця (рисунок 4.1), або FPGA, є типом інтегральної схеми (IC), яка дозволяє розробляти спеціальну логіку для швидкого створення прототипів і остаточного проектування системи. FPGA відрізняються від інших IC завдяки своїй властивій гнучкості, яка дозволяє програмувати та перепрограмувати їх за допомогою завантаження програмного забезпечення, щоб адаптуватись до мінливих потреб більшої системи, для якої вони призначені, ця особливість дозволяє реконфігурувати тести під будь-які існуючі несправності у пам'яті. Ще однією перевагою FPGA є їхня можливість паралельної обробки, що забезпечує мінімально можливу затримку в процесі виконання. Завдяки архітектурі «моря воріт» ПЛІС здатні обробляти дані паралельно, завдяки чому операції виконуються одночасно, а не послідовно. ПЛІС вимагають конфігурації, щоб логічні схеми пристрою та з'єднання знали, яку роль вони повинні відігравати в реалізації конкретної програми. Використовуючи спеціалізоване програмне забезпечення (зазвичай надається постачальником FPGA), розробники проектують логіку, яка буде реалізована в FPGA, використовуючи або графічний запис (зазвичай використовується для менших FPGA), або мову опису обладнання (VHDL).

У роботі застосована FPGA Xilinx Spartan 6. Сімейство Spartan®-6, пропонує нову, більш ефективну логіку подвійної регістрової 6-вхідної таблиці пошуку (LUT) і багатий вибір вбудованих блоків системного рівня та забезпечує передові можливості системної інтеграції з найнижчою загальною вартістю. Програмування на мові VHDL та імплементація на ПЛІС здійснені

за допомогою середи Xilinx ISE 14.7. Симуляція виконана у програмі Active-HDL 9.1.

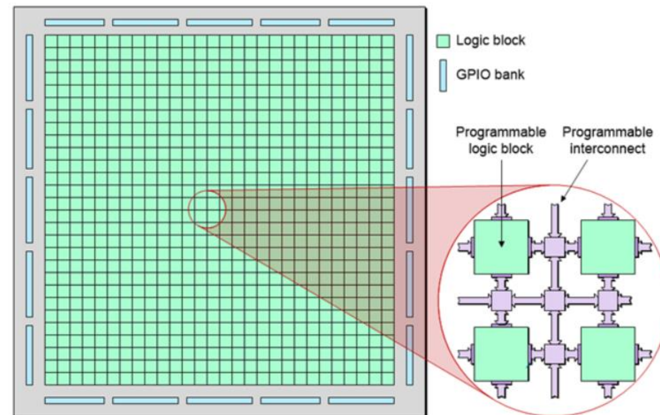


Рисунок 4.1 – Вигляд FPGA зсередини

4.2 VHDL опис системи

В якості SRAM буде використина модель яка симулює його поведінку. Основні функції: запис та зчитування по вказаній адресі (лістинг 4.1). Ці процеси мають синхронізацію по тактовому сигналу clk.

Лістинг 4.1 – Симуляція SRAM

```
subtype Trow is std_logic_vector(7 downto 0);
type    TmemArray is ARRAY (255 downto 0) of Trow;
Signal MemArray : TmemArray;
begin
     process(r,w,address,cs,din,clk) is
     begin
          if cs = ‘0’ and w = ‘0’ and rising_edge(clk) then
               MemArray(conv_integer(address))<=din;
          elsif cs = ‘0’ and r=’0’ and rising_edge(clk) then
               dout<=MemArray(conv_integer(address));
          end if;
end process;
end ram;
```

Голівки у системі представлені масивом з чотирма ячейками, у кожній з яких зберігається адреса на яку вказує відповідна голівка в даний момент часу (лістинг 4.2). Ітеративний регістр у свою чергу вказує на активну

голівку в даний конкретний такт та його максимальне значення дорівнює кількості голівок. Ітеративний регістр функціонує як лічильник та сумується з одиницею кожен такт у станах, які відповідають за генерацію тестів. Для його роботи потрібен ще один сигнал (i), з яким будуть проводитись операції у станах системи, після чого i_reg буде синхронізуватись з ним. Синхронізація виникає кожен такт clk.

Лістинг 4.2 – Голівки та ітеративний регістр

```
signal i, i_reg : std_logic_vector(1 downto 0);
type THead is ARRAY (3 downto 0) of STD_LOGIC_VECTOR(7 downto
0);
signal h : THead:= (others => "00000000");
process (clk, rst) is
begin
     if rst='1' then
          i_reg  <= "00";
     elsif rising_edge(clk) then
          i_reg  <= i;
     end if;
end process;
```

В якості прикладу реалізації генерації тестів наведемо опис стану Q4, який позначен як стан Q5 у формулах 2.7 та 2.8. За лістингом 4.3 у регістр зчитування r надходить перший біт ітеративного регістра i_reg(0), бо голівки з номерами 0 та 2 відповідають за зчитування, в бітовій формі обидва числа мають ‘0’ як перший біт. Так як система виконує одну операцію (зчитування або запису) за один такт clk, то для регістра запису w значення першого біту i_reg інвертується. Регістру адреси присвоюється адреса з масиву голівок. В регістр вхідних даних Din надходить максимальне число, а саме 255, це зроблено для тестування усіх бітів одного слова пам’яті за один такт. Далі виконується перевірка чи не вийшов алгоритм за межи матриці SRAM. Якщо так, то сигнал переходу Q3_c встановлюється в ‘1’ та виконується перехід до стану Q0, де поточні адреси всіх голівок будуть скинуті, також виконується скидання ітеративного регістру i_reg. Якщо ні, то наступним станом встановлюється компаратор, де буде виконано порівняння зчитаних даних з

еталонними, для цього у всі біти регістру еталонних даних comp_data записується ‘0’. Буфер стану buf_state встановлюється в Q4 для того щоб компаратор повернув систему у стан, який його викликав.

Лістинг 4.3 – Опис типового стану системи

```
when Q4 =>
address <= h(conv_integer(i_reg));
r <= i_reg(0);
w <= not i_reg(0);
din <= conv_std_logic_vector(255,8);
if (i_reg = "11" and h(conv_integer(i_reg)) = nmax) then
     nx_state <= Q0;
     i <= "00";
     Q3_c <= '1';
else
     nx_state <= Qcomp;
     i <= i_reg;
     buf_state <= Q4;
     comp_data <= (others => '0');
sig_step <= 2;
end if;
```

Між станами генерації тестів голівки потрібно скидати в початкове положення або встановлювати в якесь конкретне положення. Це виконується так як показано у лістингу 4.4. Наступний стан визначається в залежності від сигналів переходу, які повинні встановлюватись в якесь конкретне положення під час виходу зі стану генерації тестів.

Лістинг 4.4 – Операція скидання адрес голівок

```
when Q0 =>
nx_state <= Q0;
i <= "00";
for ii in 0 to 3 loop
     h(ii) <= (others => '0');
end loop;
if (Q0_c = '0') then
     nx_state <= Q1;
elsif (Q0_c = '1') then
     nx_state <= Q3;
end if;
```

Компаратор виконує перевірку регістру r, якщо r не дорівнює ‘0’ та операція зчитування та компарації не повинні виконуватись, та порівняння зчитаних даних (Dout) з еталонними (comp_data). Якщо перевірка не пройдена система тестування зупиняється (Qstop). В іншому випадку регістр стан повертається в той, який викликав компаратор (buf_state). Виконується зміна адреси головіки з кроком (step) який повинен був бути встановлений у стані, що викликав компаратор. Відпрацьовує лічильник на ітеративному регістрі (i). Усе описане вище продемонстроване у лістингу 4.5.

Лістинг 4.5 – Опис компаратору

```
when Qcomp =>
if (r = '0' and dout /= comp_data) then
     nx_state <= Qstop;
else
     i <= i_reg + 1;
     nx_state <= buf_state;
     h(conv_integer(i_reg)) <= h(conv_integer(i_reg)) + step;
end if;
```

4.3 Імплементація засобами Xilinx ISE

Як видно на рисунку 4.2 імплементована система займає 7% від усіх доступних слайсів, та 5% від усіх доступних комірок LUT, що доволі непоганий результат.

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	57	4,800	1%	
Number used as Flip Flops	7			
Number used as Latches	50			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	135	2,400	5%	
Number used as logic	131	2,400	5%	
Number using O6 output only	110			
Number using O5 output only	0			
Number using O5 and O6	21			
Number used as ROM	0			
Number used as Memory	4	1,200	1%	
Number used as Dual Port RAM	0			
Number used as Single Port RAM	4			
Number using O6 output only	4			
Number using O5 output only	0			
Number using O5 and O6	0			
Number used as Shift Register	0			
Number of occupied Slices	47	600	7%	
Number of MUXCYs used	8	1,200	1%	
Number of LUT Flip Flop pairs used	139			

Рисунок 4.2 – Витрачені ресурси FPGA

5 ТЕСТУВАННЯ СИСТЕМИ

5.1 Перевірка працездатності окремих станів

Відлагодження та симуляція потребують заздалегідь налаштованих сигналів скидання та тактових сигналів, для цього був описаний testbench (тестовий стенд)(лістинг 5.1), який підключає систему тестування та монтує сигнали. Період обох тактових сигналів встановлено у 10 ns, у системі один синхронізується по зростанню, інший – по падінню.

Лістинг 5.1 – Testbench

```
constant T : time := 10 ns;
signal  clk,rst,clk2: std_logic;
signal error : std_logic;
signal finish : std_logic;
begin
    unit : march_test port map (clk=>clk, rst=>rst,
     error=>error, finish=>finish, clk2=>clk2);
    process is
    begin
        clk <= '0';
          clk2 <='0';
        wait for T/2;
        clk <= '1';
          clk2<='1';
        wait for T/2;
    end process;
    rst <= '1', '0' after 5 ns;
end tb_tb;
```

Рисунок 5.1 демонструє початок виконання алгоритму, який закладено у систему. Як видно процес Q0 встановив голівки в нульове положення, Q1 вистроїв їх за порядковим номером, Q2 виконує послідовний запис даних з регістра Din у масив пам’яті MemArray. Орієнтуючись на зростаючий такт clk виконується один запис, після чого адреса голівки збільшується і алгоритм переходить до наступної голівки. Регістр протягом виконання Q2 установлений в ‘0’.

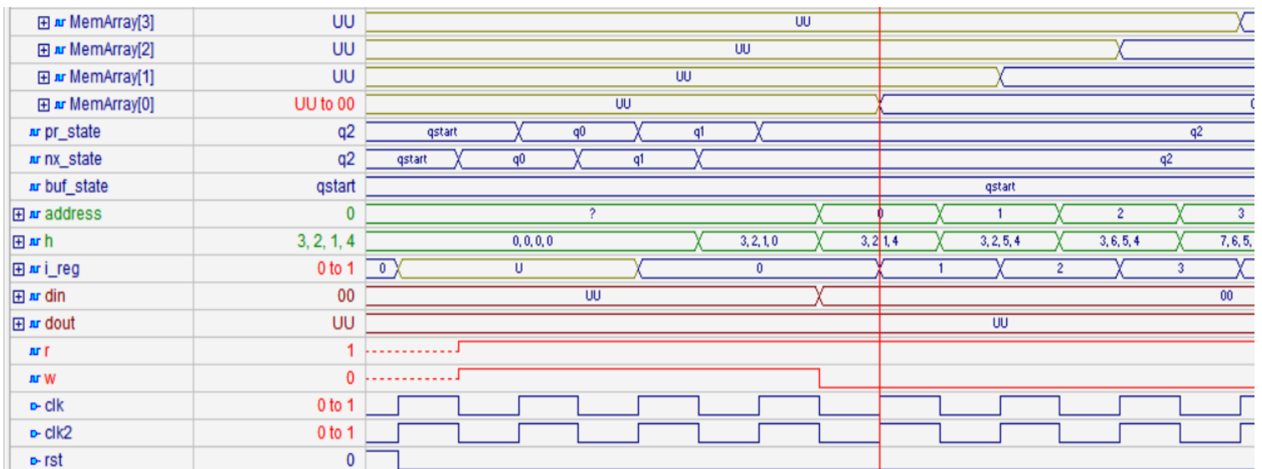


Рисунок 5.1 – Переходи Qstart->Q0->Q1->Q2

Як можна побачити на рисунку 5.2 коли остання голівка h(3) виконала операцію над останньою адресою система змінює стан з Q2 на Q0 та слідом на Q3, де виконується встановлення голівок у початкову позицію. Далі йде стан Q4. Бачимо що були зчитані дані у Dout по адресі з першої голівки, потім порівняні з Comp_data, error містить ‘0’, тому помилки не виявлено, тому відбувається запис числа 255 у комірку MemArray[0] на яку вказує друга голівка h(1). R та W синхронізовані з падінням clk2 та поперемінно інвертують своє значення для виконання зчитування або запису

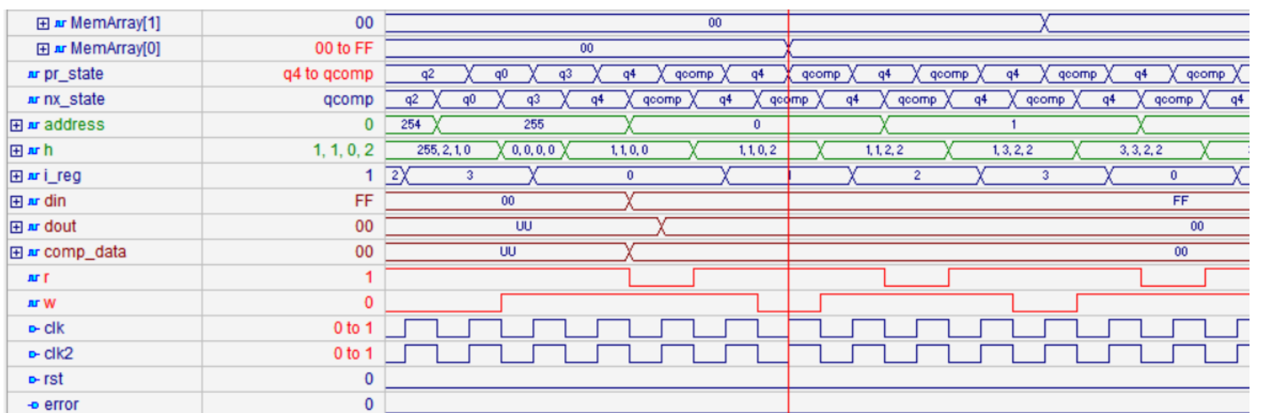


Рисунок 5.2 – Переходи Q2->Q0->Q3->Q4

Дивлячись на рисунок 5.3 можна побачити як алгоритм у Q7 добігає кінця, регістр адреси шляхом негативної ітерації дійшов до 0, тому алгоритм переходить у стан Q6, де голівки встановлюються на кінець масиву, щоб

виконувати зворотню ітерацію. На першій ітерації Q8 R встановлюється у ‘0’ ,таким чином Q8 виконує зчитування у Dout зверху вниз та порівняння з 255, у разі успіху W встановлюється у ‘0’ та виконується перемикання на наступну голівку, яка вже виконує запис нулів у масив пам’яті, що ми і бачимо на рисунку.

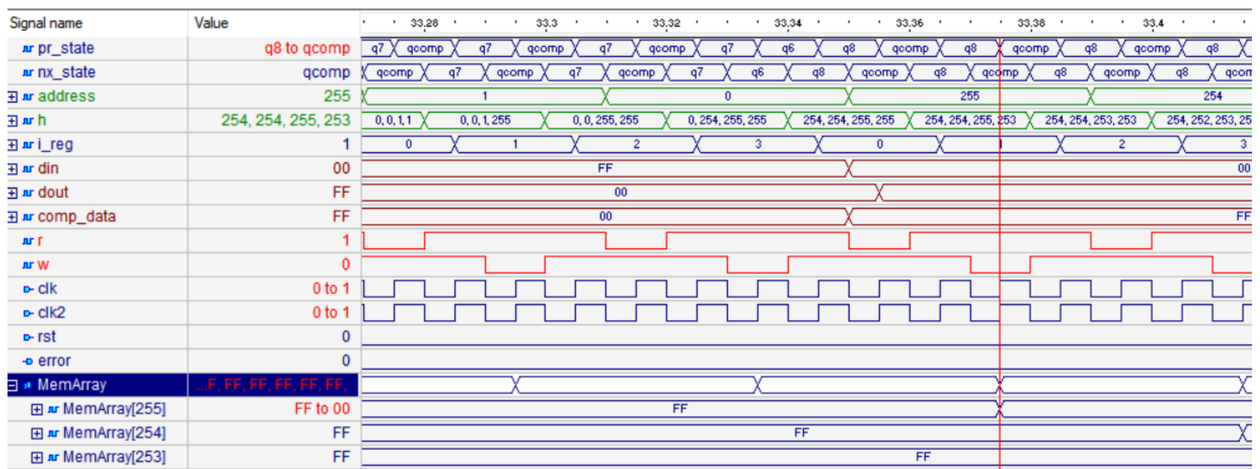


Рисунок 5.3 – Переходи Q7->Q6->Q8

Q8 завершує свій цикл як тільки адреса h(3) виконала дію над коміркою з адресою 0. Далі відбувається перехід у Q0 та Q1, які згодом встановлюють голівки на початок масиву пам’яті. Відбувається перехід у Q9, у R записується ‘0’, відбувається послідовне зчитування усіх комірок для порівняння їх із “00” яке зберігається у Comp_data. Ці процеси ілюстровані на рисунку 5.4.

Рисунок 5.5 відображає останній етап відпрацьовування алгоритму системи тестування. Після перевірки усіх комірок, а саме як тільки остання голівка h(3) виконала зчитування останньої комірки під номером 255 та успішно порівняла з еталоном, система переходить у стан Qend. Сигнал Finish сигналізує про те що тестування пам’яті завершене.

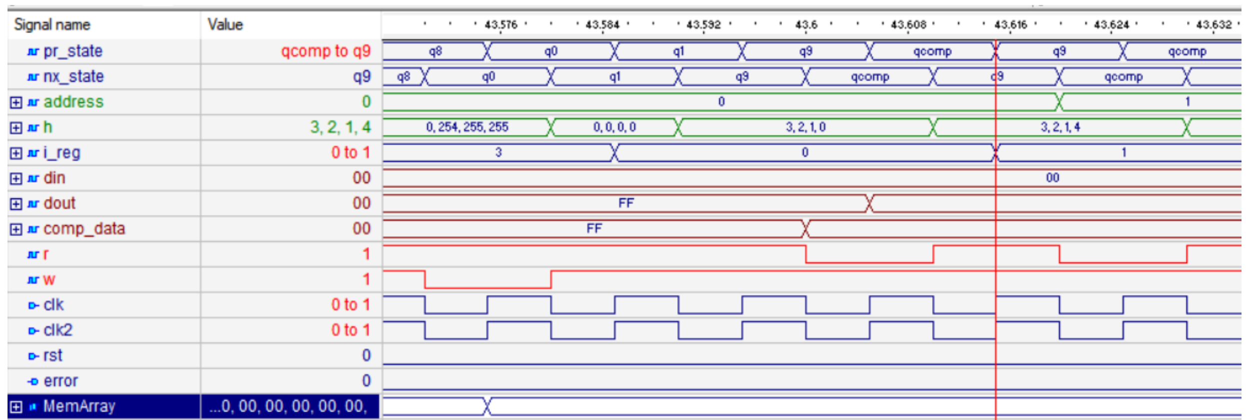


Рисунок 5.4 – Переходи Q8->Q1->Q9

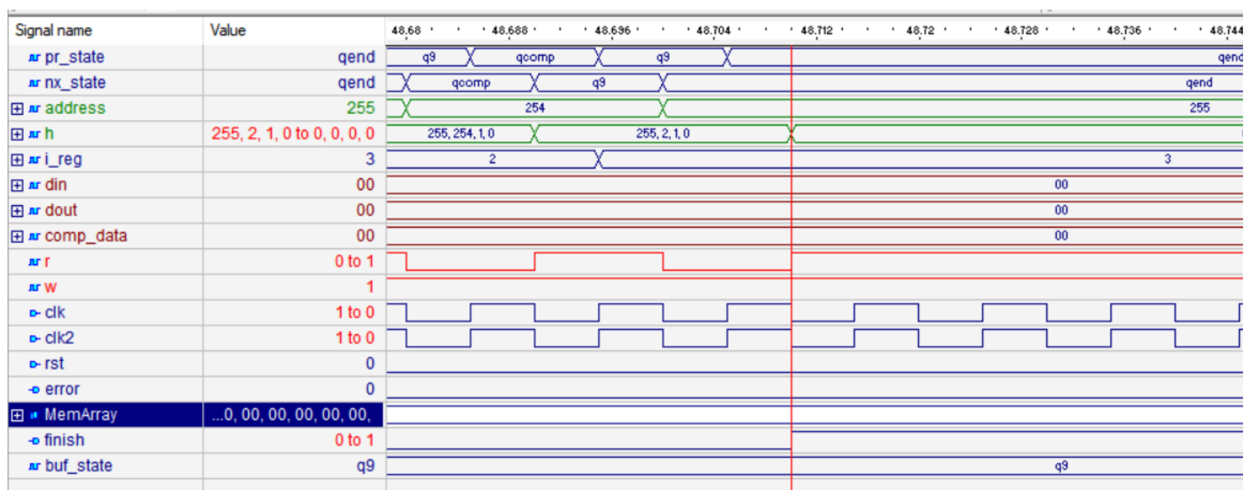


Рисунок 5.5 – Перехід Q9->Qend

Штучне введення несправності у секцію пам’яті повинно бути виконано для підтвердження того що система може розпізнавати несправності (лістинг 5.2). Результат наведений у рисунку 5.7.

Лістинг 5.2 – Несправність

```
MemArray(255) <= "10101001";
```

5.2 Час виконання

Як відомо кількість ітерацій для виконання алгоритму March C- дорівнює 10N, в нашому випадку N = 256, отже 2560 ітерацій. Період clk 10

ns, тому це займе 25600ns. Виконавши симуляцію за допомогою testbench (лістинг 5.1) отримуємо час виконання 48720ns, як наведено на рисунку 5.6. Поділимо 48720ns на 25600ns. Отже наш алгоритм виконується за 19N. Це викликано тим що стани Q4, Q7, Q8, Q9 кожного такту звертаються до компаратора, в результаті виходить вдвічі більше ітерацій. Технічно 19N це кількість ітерацій, яка необхідна системі для виконання всіх закладених тестів, в той час як сам алгоритм March C- залишається довжиною у 10N ітерацій.

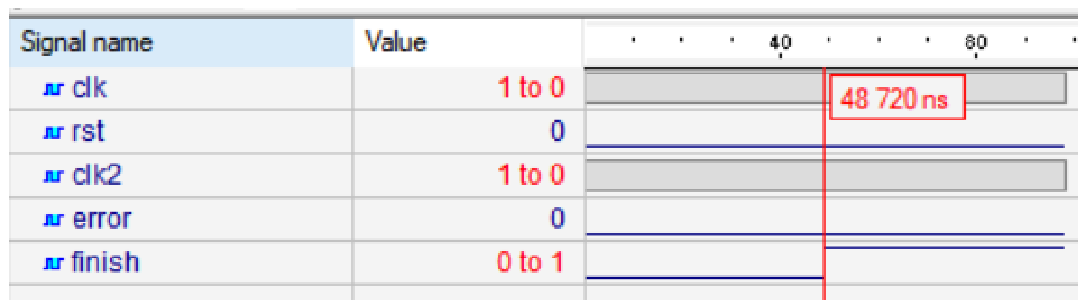


Рисунок 5.6 – Симуляція testbench

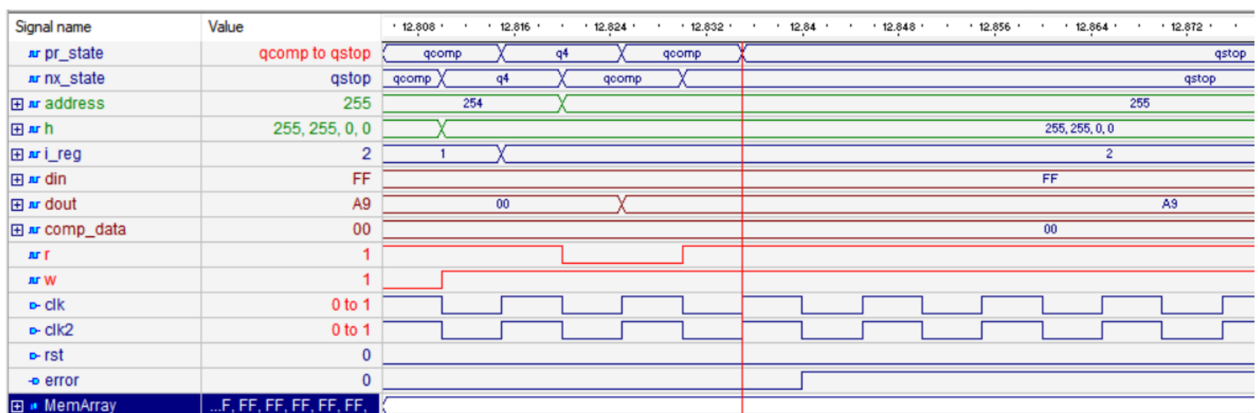


Рисунок 5.7 – Помилку знайдено

ВИСНОВКИ

У роботі були розглянуті помилки, які можуть виникать при взаємодії з комірками пам’яті RAM. Оглянуті різні підходи для знаходження несправних комірок пам’яті. Використана концепція BIST для побудування системи тестування комірок. Керуючий пристрій спроектовано за принципами автоматного програмування, а саме на основі часового рекурсивного автомату. Запрограмовану систему імплементовано на ПЛІС Xilinx Spartan 6. Синтезований алгоритм March C- покриває більшу частину існуючих несправностей SRAM. Для тестування DRAM система потребує модифікацій, а саме покриття несправностей характерних для DRAM. Спроектована система сприяє створенню модифікацій. Подальші дослідження потребують розробки системи відновлювання пам’яті BISR (Built-in Self Repair) для об’єднання її зі спроектованою системою BIST. Об’єднана система буде самодостатньою системою обслуговування несправностей в оперативній пам’яті.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Аль Мади М. К., Моамар Д. Н, Рябцев В. Г. Алгоритмы тестового диагностирования полупроводниковых запоминающих устройств. – К. : «Корнійчук», 2008. – 220 с.

2. Фалевич Б. Я. Теория алгоритмов: Учеб. пособие. – М. : Машиностроение, 2004. – 160 с.

3. Рябцев В. Г. Проектування мобільних програм діагностування сучасних мікросхем пам'яті. // Вісник ЧДТУ. – 2002. – № 2. – С. 25 – 29.

4. Michael L.Bushnell. Essentials of Electronic Testing for Digital Memory and Mixed-Signal VLSI Circuits. / Michael L.Bushnell. Vishwani D.Agarwal. – К. : Frontiers in Electronic Testing (FRET, volume 17), 2002.

5. Cheng-Wen Wu. VLSI test principles and architectures: design for testability. / Laung-Terng Wang, Cheng-Wen Wu. – К. : Elsevier, 2006.

6. Van De Goor A.J. Using MARCH tests to test SRAM / A.J Van De Goor. // Design & Test of Computers, IEEE. – 1993. – Volume 10. Issue 1. – С. 8–14.

7. M.A. Breuer. Diagnosis and Reliable Design of Digital Systems / M.A. Breuer, A.D. Friedman. – К. : Computer Science Press, Woodland Hills, Calif. – 1976.

8. A.J. van de Goor. Testing Semiconductor Memories, Theory and Practice. / A.J. van de Goor. – М. : John Wiley&Sons, Chichester, UK. – 1991.

9. .Simple and Efficient Algorithms for Functional RAM Testing : IEEE lntel Test Conference, 1982 / M. Marinescu // IEEE Computer society Press. – 1982. – С. 23–239.

10. Kamran Zarrineh. Automatic Generation and compaction of MARCH tests for memory arrays. / Kamran Zarrineh, Shambhu J. Upadhyaya, Sreejith Chakravarty. // IEEE Transactions on Very Large Scale Integration (VLSI) Systems. – Dec. 2001. – Vol. 9, №6. – С. 845–857.

11. Pedroni, Volnei A. Finite state machines in hardware : theory and design (with VHDL and SystemVerilog) / Volnei A. Pedroni. – Cambridge, Massachusetts: The MIT Press. – 2013. – 350c.

12. R. Dekker. A Realistic Fault Model and Test Algorithms for Static Random Access Memories / R. Dekker // IEEE Trans. Computers. – 1990. – Vol. C-9, No.6. – C. 567-572.

13. FPGA-based implementation and comparison of recursive and iterative algorithms : Proceedings of the 2005 International Conference on Field Programmable Logic and Applications (FPL), August 24-26 2005, Tampere, Finland / Valery Sklyarov. – Tampere, Finland, 2005.

14. M.Mamatha. Memory Testing using March C-Algorithm / M.Mamatha, M.Muralidha // International Journal of VLSI System Design and Communication Systems. – October,2014. – Vol.02, Issue.07. – C.512-517.