

Министерство образования Республики Беларусь

Учреждение образования
**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ**

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

К защите допустить:

Заведующая кафедрой ПОИТ

_____ Н. В. Лапицкая

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к дипломному проекту

на тему

**ПРОГРАММНОЕ СРЕДСТВО ВЕРИФИКАЦИИ АЛГОРИТМОВ
ТЕСТИРОВАНИЯ ОПЕРАТИВНЫХ ЗАПОМИНАЮЩИХ
УСТРОЙСТВ**

БГУИР ДП 1-40 01 01 03 027 ПЗ

Студент

А. А. Дементьева

Руководитель

А. А. Иванюк

Консультанты:

от кафедры ПОИТ

А. А. Иванюк

по экономической части

К. Р. Литвинович

Нормоконтролёр

С. В. Болтак

Рецензент

Минск 2016

РЕФЕРАТ

Пояснительная записка 104 с., 22 рис., 16 табл., 21 формул и
21 источников.

ОЗУ, ТЕСТИРУЮЩИЕ АЛГОРИТМЫ, ФУНКЦИОНАЛЬНЫЕ НЕИСПРАВНОСТИ, ВЕРИФИКАЦИЯ, АДАПТИВНЫЙ СИГНАТУРНЫЙ АНАЛИЗАТОР, ВСТРОЕННОЕ САМОТЕСТИРОВАНИЕ

Объектом исследований является покрывающая способность тестирующих алгоритмов ОЗУ, а также возможность верифицировать любой тест на любой модели неисправности. Цель работы – разработка программного средства верификации алгоритмов тестирования на различных функциональных моделях неисправностей ОЗУ. Создание данного программного средства обеспечит возможность улучшения качества разрабатываемых тестирующих алгоритмов, которые можно будет проверить на самых сложных моделях неисправностей.

Проведен анализ методов встроенного самотестирования ОЗУ, устройства работы динамических оперативных запоминающих устройств, современных тестирующих алгоритмов, а также видов функциональных моделей неисправностей ОЗУ.

Результатом разработки стала библиотека, написанная на языке программирования С++, которая содержит в себе классы симулятора динамической памяти. Данное программное средство легко внедрить в любую систему, отсутствие зависимостей от сторонних библиотек облегчает компиляцию под любой операционной системой.

Предполагается использование библиотеки программистами, занимающимися проблемами создания и верифицирования тестирующих алгоритмов оперативных запоминающих устройств.

Разработанное программное средство является экономически эффективным, т.к. полностью оправдывает вложенные в разработку средства.

СОДЕРЖАНИЕ

Введение	7
1 Обзор предметной области	8
1.1 Оперативные запоминающие устройства	8
1.2 Функциональные модели неисправностей ДОЗУ	15
1.3 Алгоритмы тестирования ДОЗУ	19
1.4 Обзор существующих аналогов	26
1.5 Постановка задачи	26
2 Используемые технологии	27
2.1 Язык программирования С++	28
2.2 Симуляторы ДОЗУ	29
3 Архитектура и модули системы	37
3.1 Основные компоненты симулятора	37
3.2 Симуляция процесса деградации памяти	39
3.3 Симуляция процесса регенерации памяти	41
3.4 Представление маршевых тестов	46
3.5 Моделирование функциональных неисправностей ОЗУ	49
3.6 Верификация неразрушающих маршевых тестов	52
4 Тестирование приложения	55
5 Методика использования разработанного приложения	60
6 Техничко-экономическое обоснование разработки ПС	65
6.1 Расчёт сметы затрат и цены программного продукта	65
6.2 Расчёт нормативной трудоемкости	67
6.3 Расчёт основной заработной платы исполнителей	69
6.4 Расчёт экономической эффективности у разработчика	72
6.5 Выводы по технико-экономическому обоснованию	73
Заключение	74
Список использованных источников	75
Приложение А Исходный код программного средства.	77

ОПРЕДЕЛЕНИЯ И СОКРАЩЕНИЯ

В пояснительной записке применяются следующие определения и сокращения.

ОЗУ - оперативное запоминающее устройство.

ЗЭ - запоминающий элемент.

ЗУ - запоминающее устройство.

ДОЗУ - динамическое запоминающее устройство.

АСА - адаптивный сигнатурный анализатор.

ПО - программное обеспечение.

ОС - операционная система.

ООП - объектно-ориентированное программирование.

SAF - stuck-at faults.

TF - transaction faults.

CF - coupling fault.

CFin - inverse coupling faults.

CFid - idempotent coupling faults.

CFst - state coupling faults.

DRAM - dynamic random access memory.

SAODC - self-adjusting output data compression.

ВВЕДЕНИЕ

Оперативные запоминающие устройства выполняют одну из важнейших функций в современных цифровых системах обработки и хранения информации. На протяжении последних лет наблюдается устойчивая тенденция по совершенствованию технологии производства цифровых устройств. Увеличение удельного веса запоминающих устройств по сравнению с операционными устройствами в вычислительных системах является следствием достижений современной микроэлектроники и вычислительной техники.

Постоянное увеличение емкости и уменьшение технологических норм производства оперативной памяти приводит к значительному увеличению количества сбоев и отказов запоминающих устройств в процессе эксплуатации цифровой техники. По результатам исследований отказы ОЗУ составляют до 70% от общего числа отказов вычислительных систем. Таким образом, обнаружение неисправных состояний ОЗУ является весьма важной и актуальной проблемой. Для повышения отказоустойчивости и надежности оперативных запоминающих устройств применяется ряд мер, позволяющих обнаружить, локализовать и исправить возникающие неисправности и ошибки. Традиционно эти проблемы решаются с применением маршевых тестов, эффективно обнаруживающих простейшие модели неисправностей ОЗУ. Но с неуклонным ростом емкости ОЗУ, которая превышает 10^9 бит, а также существенное отличие физической структуры ОЗУ от их логической организации, делает задачу обнаружения неисправностей всё более сложной и трудоемкой. С каждым годом разрабатываются всё новые и новые тесты, описываются достаточно сложные модели неисправностей, вследствие чего крайне необходим аппарат для проверки тестирующих алгоритмов на различных моделях неисправностей. Это улучшит качество и позволит достичь высоких показателей надежности и отказоустойчивости современных цифровых систем.

В данном дипломном проекте реализуются некоторые из известных алгоритмов тестирования оперативных запоминающих устройств. Моделируются наиболее распространенные функциональные неисправности ОЗУ, а также проверяется покрывающая способность неразрушающих маршевых тестов. В результате получилась библиотека симулятора динамического ОЗУ, написанная на языке программирования C++, пригодная для решения практических задач в реальных перспективах.

1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

В данном разделе будет произведён обзор предметной области задачи, решаемой в рамках дипломного проекта;

1.1 Оперативные запоминающие устройства

Компактная микроэлектронная память находит широкое применение в самых различных по назначению электронных устройствах. Понятие «память» связывается с ЭВМ и определяется как ее функциональная часть, предназначенная для записи, хранения и выдачи данных. Комплекс технических средств, реализующих функцию памяти, называется запоминающим устройством (ЗУ).

Микросхема памяти содержит выполненные в одном полупроводниковом кристалле матрицу-накопитель, представляющую собой совокупность элементов памяти (ЭП) и функциональные узлы, необходимые для управления матрицей-накопителем, усиления сигналов при записи и считывании, обеспечения режима синхронизации. Элемент памяти может хранить один разряд числа, т. е. один бит информации.

По назначению микросхемы памяти делят на две группы: для оперативных запоминающих устройств (ОЗУ) и для постоянных запоминающих устройств (ПЗУ). ОЗУ предназначено для использования в условиях, когда необходимо выбирать и обновлять хранимую информацию. Вследствие этого в ОЗУ предусматриваются три режима работы: режим хранения при отсутствии обращения к ЗУ, режим чтения информации и режим записи новой информации. При этом в режимах чтения и записи ОЗУ должно функционировать с высоким быстродействием (время чтения или записи составляет доли микросекунды). В цифровых вычислительных устройствах ОЗУ используются для хранения промежуточных и конечных результатов обработки данных. При отключении источника питания информация в ОЗУ теряется. В условном графическом обозначении функция ОЗУ задается комбинацией символов «RAM» – random access memory (память с произвольным доступом) [1].

Существует два типа ОЗУ: статическое и динамическое. Статическое ОЗУ (Static RAM, SRAM) конструируется с использованием D-триггеров. Информация в ОЗУ сохраняется на протяжении всего времени, пока к нему подаётся питание: секунды, минуты, часы и даже дни. Статическое ОЗУ работает очень быстро. Обычно время доступа составляет несколько на-

носекунд. По этой причине статическое ОЗУ часто используют в качестве кэш-памяти второго уровня [2].

В динамическом ОЗУ (Dynamic RAM, DRAM), напротив, триггеры не используются. Динамическое ОЗУ представляет собой массив ячеек, каждая из которых содержит транзистор и крошечный конденсатор. Конденсаторы могут быть заряженными и разряженными, что позволяет хранить нули и единицы. Поскольку электрический заряд имеет тенденцию исчезать, каждый бит в динамическом ОЗУ должен обновляться (перезаряжаться) каждые несколько миллисекунд, чтобы предотвратить утечку данных. Поскольку обновление должна заботиться внешняя логика, динамическое ОЗУ требует более сложного сопряжения, чем статическое, хотя этот недостаток компенсируется большим объёмом [3].

Поскольку динамическому ОЗУ нужен только один транзистор и один конденсатор на бит (статическому ОЗУ требуется в лучшем случае 6 транзисторов на бит), динамическое ОЗУ имеет очень высокую плотность записи (много битов на одну микросхему). По этой причине основная память почти всегда строится на основе динамических ОЗУ. Однако динамические ОЗУ работают очень медленно (время доступа занимает десятки наносекунд). Таким образом, сочетание кэш-памяти на основе статического ОЗУ и основной памяти на основе динамического ОЗУ соединяет в себе преимущества обоих устройств.

Существует несколько типов динамических ОЗУ. Самый древний тип, который все еще используется, - FPM (Fast sub Mode - быстрый постраничный режим). Это ОЗУ представляет собой матрицу битов. Аппаратное обеспечение представляет адрес строки, а затем - адреса столбцов. Благодаря явно передаваемым сигналам память работает асинхронно по отношению к главному тактовому генератору системы [4].

FPM постепенно замещается памятью EDO (Extended Data Output - память с расширенными возможностями вывода), которая позволяет обращаться к памяти еще до того, как закончилось предыдущее обращение. Такой конвейерный режим, хотя и не ускоряет доступ к памяти, повышает пропускную способность, позволяя получить больше слов в секунду.

Память типа FPM и EDO сохраняла актуальность в те времена, когда продолжительность цикла работы микросхем памяти не превышала 12 нс. Впоследствии, с увеличением быстродействия процессоров, сформировалась потребность в более быстрых микросхемах памяти, и тогда на смену асинхронным режимам FPM и EDO пришли синхронные динамические ОЗУ (Synchronous DRAM, SDRAM). Синхронное динамическое ОЗУ управ-

ляется от главного системного тактового генератора. Данное устройство представляет собой гибрид статического и динамического ОЗУ. Основное преимущество синхронного динамического ОЗУ состоит в том, что оно исключает зависимость микросхемы памяти от управляющих сигналов. ЦП сообщает памяти, сколько циклов следует выполнить, а затем запускает её. Каждый цикл на выходе дает 4, 8 или 16 бит в зависимости от количества выходных строк. Устранение зависимости от управляющих сигналов приводит к ускорению передачи данных между ЦП и памятью [5].

Следующим этапом в развитии памяти SDRAM стала память DDR (Double Data Rate - передача данных с двойной скоростью). Эта технология предусматривает вывод данных как на фронте, так и на спаде импульса, вследствие чего скорость передачи увеличивается вдвое. Например, 8-разрядная микросхема такого типа, работающая с частотой 200 МГц, дает на выходе два 8-разрядных значения 200 миллионов раз в секунду (разумеется, такая скорость удерживается в течение небольшого периода времени); таким образом, теоретически, кратковременная скорость может достигать 3,2 Гбайт/с. Интерфейсы памяти DDR2 и DDR3 обеспечивают дополнительный прирост производительности по сравнению с DDR за счет повышения скорости шины памяти до 533 МГц и 1067 МГц соответственно.

1.1.1 Динамические оперативные запоминающие устройства

Как уже отмечалось, информация в ячейке динамического ОЗУ представлена в виде наличия или отсутствия заряда на конденсаторе. Схема ячейки памяти ЯП динамического ЗУ на одном МОП-транзисторе с индуцируемым р-каналом представлена на рисунке 1.1 (выделена пунктирной линией). На схеме также показаны общие элементы для n-ячеек одного столбца. Главное достоинство этой схемы - малая занимаемая площадь. Накопительный конденсатор С1 имеет МДП-структуру и изготавливается в едином технологическом цикле. Величина его емкости составляет сотые доли пикоФарад. Конденсатор С1 хранит информационный заряд. Транзистор VT1 выполняет роль переключателя, передающего заряд конденсатора в разрядную шину данных ШД при считывании, либо заряжающего конденсатор при записи. В режиме хранения на адресной линии А1 должен присутствовать потенциал логической единицы, под действием которого транзистор VT1 будет закрыт и конденсатор С1 отключен от шины данных ШД. Включение конденсатора в шину данных осуществляется логическим нулем на линии. При этом на транзистор VT1 подается напряжение, что приводит к его открыванию [5].

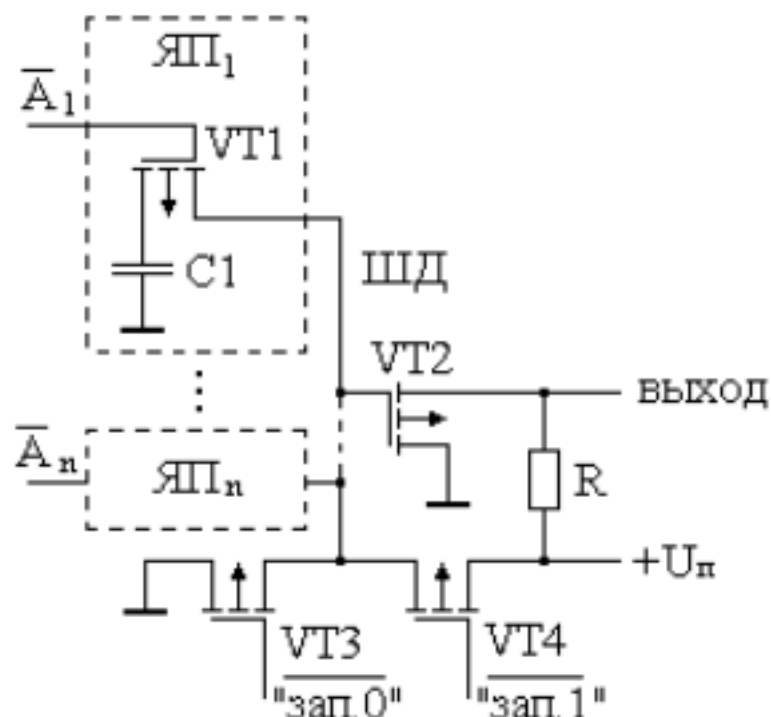


Рисунок 1.1 – Принципиальная схема ячейки динамического ОЗУ

Поскольку шина данных ШД объединяет все ячейки памяти данного столбца, то она характеризуется большой длиной и ее собственная емкость имеет существенное значение. Поэтому при открывании транзистора VT1 потенциал шины данных изменяется незначительно. Чтобы установившийся потенциал на ШД однозначно идентифицировать с уровнем напряжения логического нуля или логической единицы, используется усилитель на базе транзистора VT2 и резистора R. Непосредственно перед считыванием емкость шины данных подзаряжают подключением ее к источнику питания через транзистор VT4. Делается это для фиксации потенциала шины данных. При считывании информации происходит перераспределение заряда конденсатора и заряда шины данных, в результате чего информация, хранящаяся на конденсаторе C1, разрушается. Поэтому в цикле считывания необходимо произвести восстановление (регенерацию) заряда конденсатора. Для этих целей, а также для записи в ячейку памяти новых значений, используются транзисторы VT3 и VT4, которые подключают шину данных либо к источнику питания, либо к нулевому общему потенциалу. Для записи в ячейку памяти логической единицы необходимо открыть транзистор VT4 нулевым значением управляющего сигнала «зап.1» и подключить к шине данных источник питания. Для записи логического нуля необходимо

нулевым потенциалом на входе «/зап.0» открыть транзистор VT3. Одновременная подача логических нулей на входы «/зап.1» и «/зап.0» не допускается, так как это вызовет короткое замыкание источника питания на общий провод заземления.

На рисунке 1.2 показан пример структуры микросхемы динамического ОЗУ емкостью 64кбит. Данные в этой микросхеме памяти представлены как 64к отдельных бит, т.е. формат памяти 64к x 1. Ввод и вывод осуществляется отдельно, для чего предусмотрена пара выводов DI (вход) и DO (выход). Для ввода адреса имеется восемь контактов $A_0 - A_7$. Адресация к 64к ячейкам памяти осуществляется шестнадцатиразрядными адресами $A_0 - A_{15}$. Причем сначала на входы $A_0 - A_7$ подаются восемь младших разрядов $A_0 - A_7$ адреса, а затем – восемь старших разрядов $A_8 - A_{15}$. Восемь младших разрядов адреса фиксируются в регистре адреса строки подачей сигнала /RAS (сигнал выборки строки). Восемь старших разрядов адреса фиксируются в регистре адреса столбца подачей сигнала /CAS (сигнал выборки столбца). Такой режим передачи кода адреса называется мультиплексированным по времени. Мультиплексирование позволяет сократить количество выводов микросхемы. Ячейки памяти расположены в виде матрицы из 128 строк и 512 столбцов. Дешифратором строк вырабатывается адресный сигнал выборки A_i ячеек памяти i -ой строки, т.е. выбирается одна из 128 строк. Обращение к строке вызывает подключение 512 ячеек памяти через соответствующие разрядные шины данных ШД этой строки к усилителям считывания (по одному на столбец). При этом автоматически происходит подзаряд запоминающих конденсаторов всех ячеек памяти выбранной строки до исходного уровня за счет передачи усиленного сигнала по цепи обратной связи. Этот процесс называется регенерацией памяти. Дешифратор столбцов выбирает один из 512 усилителей считывания. Бит, выбранный в режиме считывания, выдается на линию DO. Если одновременно с сигналом /CAS при предварительно установленном сигнале /RAS действует сигнал записи /WR, то бит с входа DI будет записан в выбранную ячейку памяти, при этом выход DO микросхемы остается в отключенном состоянии в течение всего цикла записи [1].

На рисунке 1.3 представлены временные диаграммы, поясняющие работу динамического ОЗУ. В режиме считывания (рисунок 1.3,а) на адресные входы микросхемы подаются восемь младших разрядов $A_0 - A_7$ адреса, после чего вырабатывается сигнал /RAS, при этом производится выбор строки матрицы в соответствии с поступившим адресом. У всех ячеек памяти выбранной строки регенерируется заряд конденсаторов. Далее производится

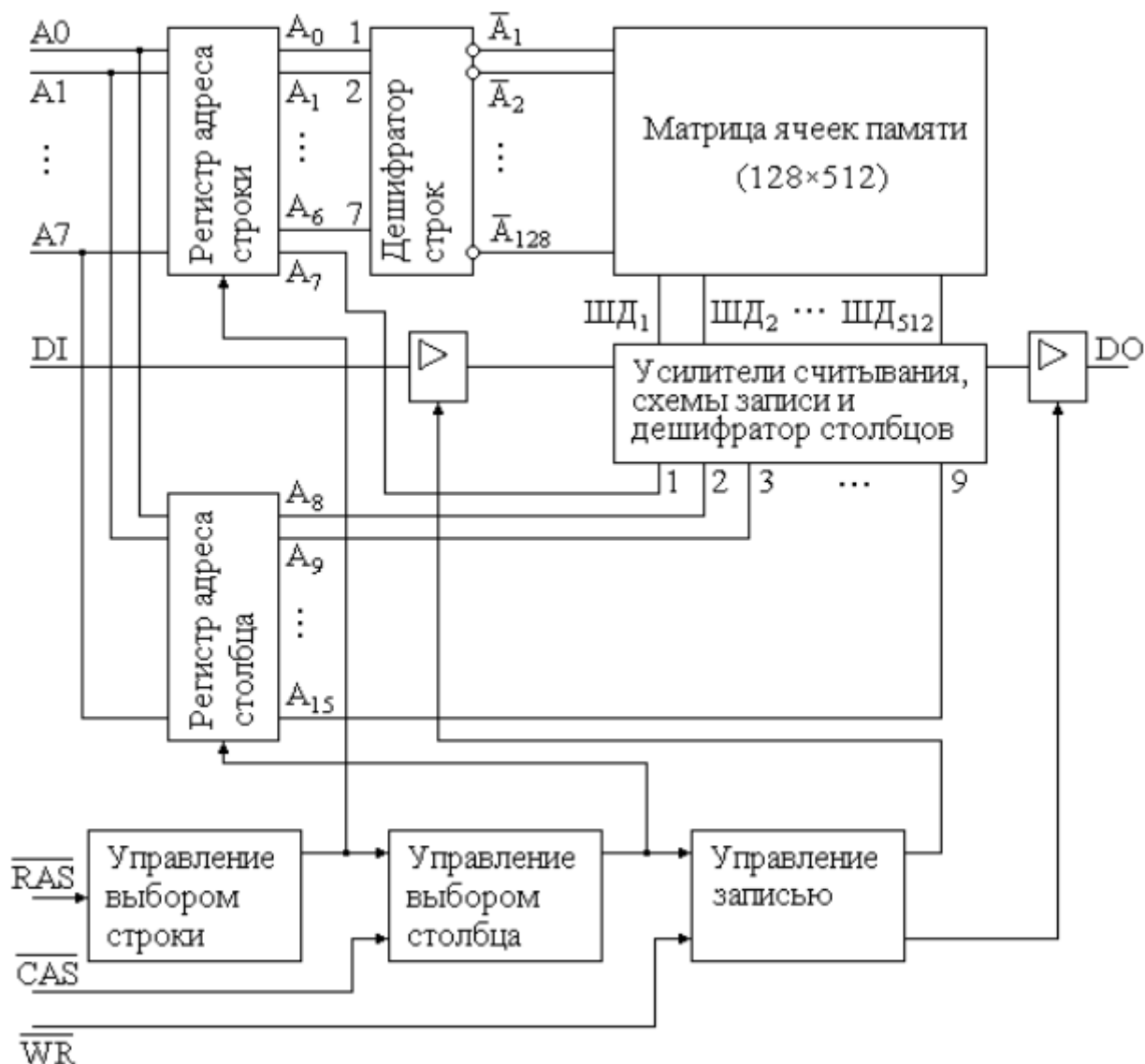


Рисунок 1.2 – Принципиальная схема ячейки динамического ОЗУ

подача на адресные входы микросхемы восьми старших разрядов адреса, после чего вырабатывается сигнал \overline{CAS} . Этим сигналом выбирается нужная ячейка памяти из выбранной строки и считанный бит информации поступает на выход микросхемы DO. В режиме считывания промежуток времени между подачей сигнала \overline{RAS} и появлением данных на выходе DO называется временем выборки t_v .

В режиме записи (рисунок 1.3,б) за время цикла записи $t_{цз}$ принимается интервал времени между появлением сигнала \overline{RAS} и окончанием сигнала \overline{WR} . В момент появления сигнала \overline{CAS} записываемые данные уже должны поступать на вход DI. Сигнал \overline{WR} обычно вырабатывается раньше сигнала \overline{CAS} .

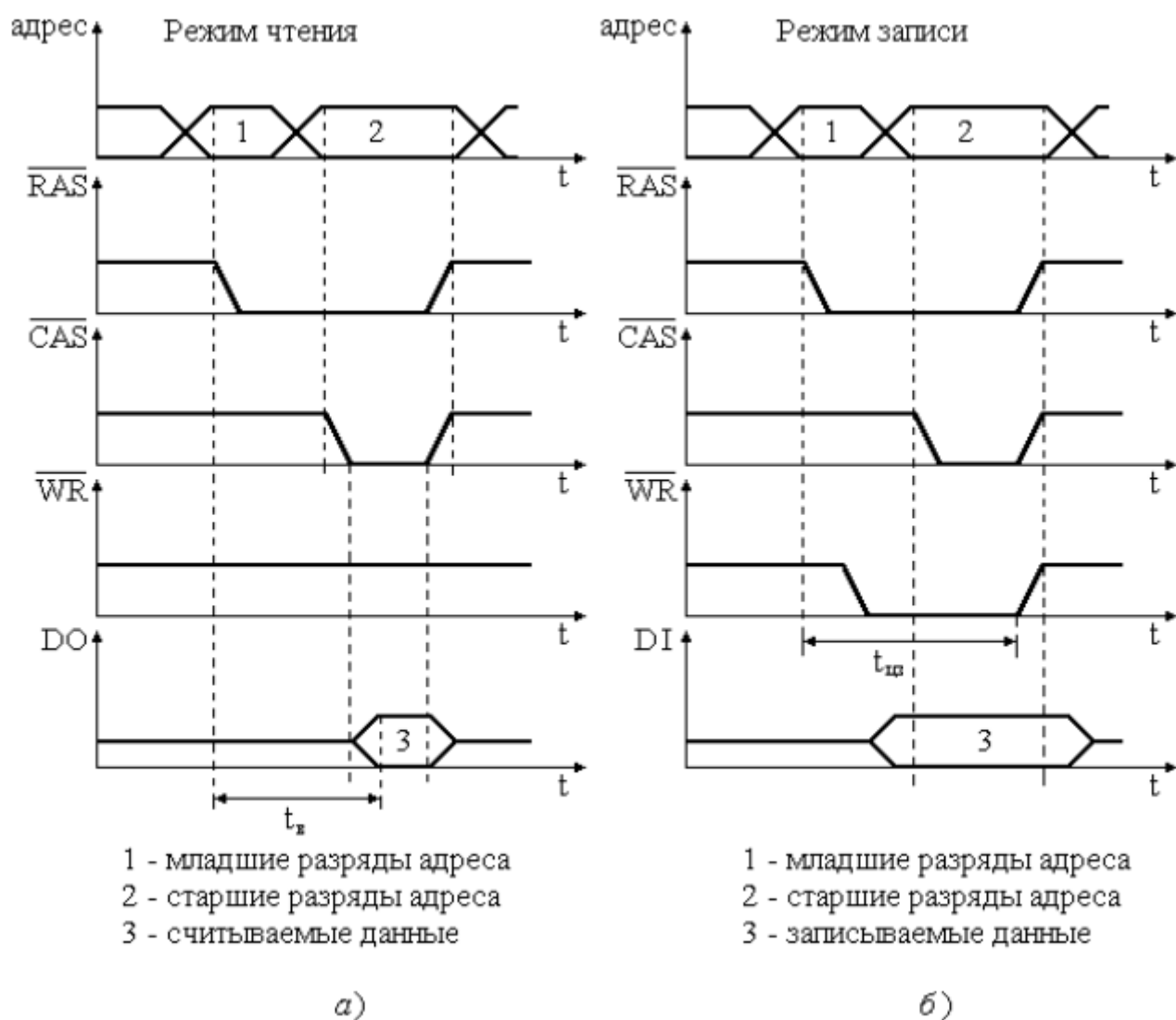


Рисунок 1.3 – Временная диаграмма работы ОЗУ динамического типа

Для каждого типа микросхем динамических ОЗУ в справочниках приводятся временные параметры, регламентирующие длительность управляющих сигналов, подаваемых на микросхему, а также порядок их взаимного следования. Заряд конденсатора динамического ОЗУ со временем уменьшается вследствие утечки, поэтому для сохранения содержимого памяти процесс регенерации каждой ячейки памяти должен производиться через определенное время. Следовательно, для предотвращения разряда запоминающих конденсаторов необходимо обращаться к каждой строке матрицы через определенное время. При обычном режиме работы ОЗУ это условие не соблюдается, так как обращение к одним ячейкам происходит часто, а к другим очень редко. Поэтому необходим специальный блок, ответственный за регенерацию памяти. Этот блок должен при отсутствии обращений к

ОЗУ со стороны внешних устройств циклически формировать на адресных входах $A_0 - A_6$ значения всех возможных адресов, сопровождая каждый из них управляющим сигналом /RAS, т.е. производить циклическое обращение ко всем 128 строкам матрицы ячеек памяти. Регенерацию необходимо проводить и в те моменты времени, когда ОЗУ используется устройствами, приостанавливая на время регенерации взаимодействие ОЗУ с этими устройствами, т.е. путем перевода этих устройств в режим ожидания.

Из изложенного выше следует, что использование динамического ОЗУ требует довольно сложной схемы управления. Если учесть, что обращение к ОЗУ со стороны устройств, с которыми оно работает, и обращение со стороны схемы регенерации не зависят друг от друга, следовательно, могут возникать одновременно, то необходима схема, обеспечивающая упорядоченность этих обращений. Для этих целей существуют схемы, управляющие работой динамических ОЗУ. Это так называемые контроллеры динамического ОЗУ, реализованные на одном кристалле. Их использование позволяет значительно упростить построение памяти на динамических ОЗУ.

Лидером в производстве микросхем динамического ОЗУ на сегодняшний день является фирма Samsung. Емкость одной микросхемы DRAM достигает значения 128 Мбайт и более. Кроме того, этой фирмой предлагается ряд передовых идей по обеспечению наибольшего быстродействия. Например, операции чтения и записи выполняются дважды за один такт – по переднему и заднему фронтам тактового импульса. Фирмой Mitsubishi предложена концепция встраивания в микросхемы динамической памяти статической кэш-памяти небольшого объема (Cashed DRAM), в которой хранятся наиболее часто запрашиваемые данные.

1.2 Функциональные модели неисправностей ДОЗУ

Причиной неисправного состояния ОЗУ является наличие физического или механического дефекта либо множества подобных дефектов, количество и многообразие которых практически неограниченно. В зависимости от технологических особенностей при производстве ОЗУ и внешних факторов при его эксплуатации могут появляться новые типы и разновидности дефектов. Определение факта возникновения дефекта и его классификация представляется весьма трудоёмкой и зачастую неразрешимой задачей.

Функциональные неисправности ОЗУ подразделяются на два подмножества: неисправности матрицы запоминающих элементов и неисправности электронного обрамления. Второе подмножество включает неисправности

дешифратора адреса и неисправности логики чтения/записи. Доминирующее значение имеют неисправности матрицы запоминающих элементов.

К неисправностям матрицы запоминающих элементов ОЗУ относят неисправности, в которых участвуют: одна ячейка ОЗУ; две ячейки ОЗУ; несколько ячеек ОЗУ, в общем случае более чем две.

К неисправностям, затрагивающим одну ячейку ОЗУ, относят неисправности типа SAF и TF [6].

Константные неисправности (Stuck-At Faults - SAF). Неисправный ЗЭ ОЗУ постоянно находится в состоянии логического нуля (SAF0) или логической единицы (SAF1), независимо от операций, выполняемых с неисправным ЗЭ и другими ЗЭ ОЗУ. Как и для случая произвольной логики, различают однократные и многократные константные неисправности.

Переходные неисправности (Transition Faults - TF). Подобные неисправности характеризуются невозможностью перехода состояния неисправного ЗЭ из 0 в 1 (TF1 \uparrow) или из 1 в 0 (TF \downarrow) при выполнении соответствующих операций записи. Данный тип неисправности достаточно близок по своей сути к константным неисправностям. Действительно, если ячейка, имеющая переходную неисправность, оказывается в состоянии, из которого она не может перейти в другое, ее поведение повторяет поведение ячейки, содержащей константную неисправность.

Среди неисправностей, в которых участвуют две ячейки ЗУ, выделяют неисправности типа CF и PSF.

Неисправности взаимного влияния (Coupling Fault - CF). При описании данной неисправности выделяют влияющий ЗЭ (i - Aggressor Cell), изменение логического состояния которого воздействует на состояние зависимого ЗЭ (j - Victim Cell). Неисправности взаимного влияния CF подразделяются на три типа:

а) *Инверсные неисправности взаимного влияния* (Inverse Coupling Faults - CF_{in}). При наличии данной неисправности изменение значения b_i влияющего ЗЭ вызывает инвертирование значения b_j зависимого ЗЭ. Возможны следующие виды неисправностей CF_{in}: $\wedge(\uparrow, b_j^*)$, $\wedge(\downarrow, b_j^*)$, $\vee(\uparrow, b_j^*)$, $\vee(\downarrow, b_j^*)$. Символ \wedge и символ \vee задают взаимное расположение влияющего и зависимого запоминающих элементов ОЗУ. Первый символ \wedge означает, что ЗЭ с меньшим адресом влияет на ЗЭ с большим адресом ($i < j$), а символ \vee используется в случае, когда адрес влияющего ЗЭ больше адреса зависимого ЗЭ ($i > j$).

б) *Неисправности взаимного влияния прямого действия* (Idempotent Coupling Faults - CF_{id}). При изменении значения b_j влияющего ЗЭ происхо-

дит принудительная установка определённого логического значения 0 или 1 в зависимом ЗЭ. Различают восемь неисправностей прямого действия: $\wedge(\uparrow,0)$, $\wedge(\uparrow,1)$, $\wedge(\downarrow,0)$, $\wedge(\downarrow,1)$, $\vee(\uparrow,0)$, $\vee(\uparrow,1)$, $\vee(\downarrow,0)$ и $\vee(\downarrow,1)$. При исследовании эффективности тестов ОЗУ анализируется их покрывающая способность для всех 12 неисправностей взаимного влияния, в которых участвуют две ячейки (2-Coupling Faults) [6].

в) *Статические неисправности взаимного влияния* (State Coupling Faults - CFst). Переход зависимого ЗЭ в какое либо состояние b_j возможен при определённом значении b_i влияющего ЗЭ. Возможно восемь неисправностей CFst: $\wedge(0,0)$, $\wedge(0,1)$, $\wedge(1,0)$, $\wedge(1,1)$, $\vee(0,0)$, $\vee(0,1)$, $\vee(1,0)$ и $\vee(1,1)$.

Кодочувствительные неисправности (Pattern Sensitive Faults - PSF) рассматриваются как обобщение моделей неисправностей взаимного влияния. Для подобных неисправностей логическое состояние или изменение логического состояния одного ЗЭ ОЗУ может зависеть от содержимого (0 или 1) или от логических переходов из 1 в 0 или из 0 в 1 влияющих ЗЭ ОЗУ. В случае кодочувствительной неисправности PSF_k , в которой участвуют k запоминающих элементов ОЗУ, подразумевается, что влияющими ЗЭ в предельном случае могут быть любые $k-1$ из N ЗЭ ОЗУ, а зависимым один из оставшихся $N-k+1$ ЗЭ. Такие неисправности называются неограниченными (Unrestricted) кодочувствительными неисправностями. Очевидно, что практическая значимость подобной модели невысока, так как сложность теста памяти и время его реализации при такой интерпретации кодочувствительной неисправности превышает всякие возможные пределы современных значений емкостей ЗУ. Поэтому при рассмотрении кодочувствительных неисправностей вводятся ограничения как на количество ЗЭ k , так и на их местоположение [6].

При тестировании ОЗУ, как правило, придерживаются модели *границных кодочувствительных неисправностей* (Neighborhood Pattern Sensitive Faults - NPSF) как более реальной модели кодочувствительных неисправностей. Для таких моделей первым и необходимым ограничением является количество ЗЭ, участвующих в неисправности. Как правило, k не превышает 10, это следует из того факта, что для тестирования подобных неисправностей необходимо время пропорциональное величине 2^k . Вторым ограничением является физическое соседство ЗЭ. При этом зависимый ЗЭ называется *базовым* ЗЭ (Base Cell), или базовой ячейкой, которая в данном случае является аналогом *жертвы*, а остальные $k-1$ ЗЭ соседними ЗЭ (Neighborhood Cells), или *соседними* ячейками, выступающими в роли агрессоров, количество и местоположение которых может быть произвольным. Поэтому на

практике обычно используют модели кодочувствительных неисправностей NPSF k с числом k , равным 3, 5 и 9, конфигурации и обозначение которых приведены на рисунке 1.4.

							C_n				C_n	C_n	C_n	
	C_n	C_b	C_n			C_n	C_b	C_n			C_n	C_b	C_n	
							C_n				C_n	C_n	C_n	
NPSF3					NPSF5					NPSF9				

Рисунок 1.4 – Модели кодочувствительных неисправностей

Как видно из рисунка, в каждой из приведённых неисправностей в явном виде выделяется базовый ЗЭ - C_b и соседние ЗЭ C_n , которые являются физическими соседями по отношению к базовому элементу.

В зависимости от эффекта влияния на базовый ЗЭ различают несколько классических типов кодочувствительных неисправностей NPSF k . *Пассивными кодочувствительными неисправностями* (Passive NPSF - PNPSF) являются неисправности, при которых состояние базового ЗЭ не может быть изменено для определённого кода в $k-1$ соседних ЗЭ ОЗУ. *Под активными кодочувствительными неисправностями* (Active NPSF - ANPSF) понимают неисправности, в которых базовый ЗЭ изменяет свое состояние из-за изменения кода в соседних ЗЭ. Изменение кода для подобных неисправностей происходит в результате изменения состояния на противоположное только в одном соседнем элементе, в то время как остальные ячейки сохраняют предыдущее состояние. *Статические кодочувствительные неисправности* (Static NPSF - SNPSF) характеризуются тем, что для определённой комбинации значений в соседних ЗЭ состояние базового ЗЭ принудительно устанавливается в состояние 0 или состояние 1. Главным отличием статических неисправностей от активных кодочувствительных неисправностей является длительность процесса установления неверного значения в базовой ячейке [6].

Рассмотренные неисправности являются классическими и наиболее часто используются при тестировании ОЗУ. Однако существует и ряд других моделей неисправностей, также обсуждаемых и применяемых при тестировании памяти. В первую очередь здесь необходимо отметить *неисправности дешифратора адреса* (Address Decoder Faults - AF), или *адресные*

неисправности. Однако обнаружение подобных неисправностей обеспечивается тестами, обнаруживающими неисправности матрицы ЗЭ.

Также к электронному обрамлению в большей мере относятся *неисправности типа обрыв* (Stuck Open Fault - SOF), которые также могут быть обнаружены при тестировании матрицы ОЗУ. Логика чтение/запись обеспечивает интерфейс между массивом ячеек и внешней шиной данных. Данный функциональный блок может содержать те же типы неисправностей, что и массив ячеек памяти (SAF, TF, CFin, CFid). Модель типа *разрушающая операция чтения* (Read Disturb Fault Model - RDF) практически аналогична неисправности *слабый запоминающий элемент* (Weak Cell Fault Model - WCF) и обнаруживается путем внесения в классический маршевый тест временной задержки перед выполнением следующего маршевого элемента. Для ДОЗУ также характерна *неисправность сохранения данных* (Data Retention Fault - DRF), которая, так же, как и неисправность RDF и WCF, обнаруживается путем внесения временных задержек в тест памяти.

1.3 Алгоритмы тестирования ДОЗУ

В условиях постоянного усложнения разрабатываемых средств электронной техники, как функционального, так и структурного (уже сегодня на подложке микропроцессора может быть расположено несколько миллионов транзисторов), в условиях необходимости быстрого продвижения уже разработанных устройств на рынок (быстрое моральное устаревание средств вычислительной техники) проблема надежного тестирования современных микропроцессоров и микропроцессорных систем становится все более актуальной. При этом очевидно, что качество и оперативность проектирования, эффективность и надежность функционирования микропроцессоров и микропроцессорных систем существенно зависят от качества и достоверности результатов решения задач их тестового диагностирования.

Рассмотрим классические методы тестирования ОЗУ.

1.3.1 Тестирующий алгоритм Zero-One

Тест Zero-One также известен как MSCAN. Он является достаточно простым и состоит из следующих шагов:

- а) запись 0 во все ячейки памяти;
- б) чтение ячеек памяти с ожидаемым значением 0;
- в) запись 1 во все ячейки памяти;
- г) чтение ячеек памяти с ожидаемым значением 1.

Сложность данного теста составляет $4n$. Не все неисправности типа Afs обнаруживаются. Неисправности типа SAFs обнаруживаются при условии исправного дешифратора адресов. Не все TFs и CFs обнаруживаются.

1.3.2 Тестирующий алгоритм Checkboard

Данный алгоритм предполагает создание сетки из нулей и единиц в шахматном порядке, как представлено на рисунке 1.5.

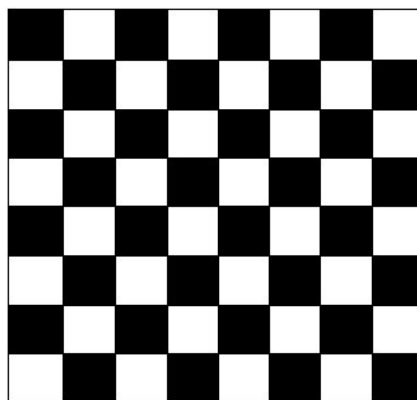


Рисунок 1.5 – Состояние ячеек памяти при тестировании алгоритмом Checkboard

Алгоритм теста состоит из следующих шагов:

- а) запись 0 во все черные ячейки памяти и 1 во все белые ячейки;
- б) чтение всех ячеек памяти;
- в) запись 1 во все черные ячейки памяти и 0 во все белые ячейки;
- г) чтение всех ячеек памяти.

Сложность теста составляет $4n$. Тест обнаруживает не все неисправности Afs. SAFs обнаруживаются при условии исправного дешифратора адресов. Не все TFs и CFs обнаруживаются. В данном алгоритме важно построить шахматную доску на физической сетке элементов, а не на логическом уровне их взаимного расположения.

1.3.3 Тестирующий алгоритм GALPAT

Алгоритм GALPAT (Galloping Pattern) состоит в заполнении памяти 0 (1), кроме базовой ячейки (base-cell), которая содержит 1 (0). На протяжении теста базовая ячейка перемещается по всей матрице. Тест проиллюстрирован на рисунке 1.6.

Сложность алгоритма составляет $4n^2$. Покрывающая способность теста достаточно высокая, все неисправности типа Afs, TFs, CFs и SAFs обнаруживаются и локализуются. Псевдокод теста приведён в листинге 1.1.

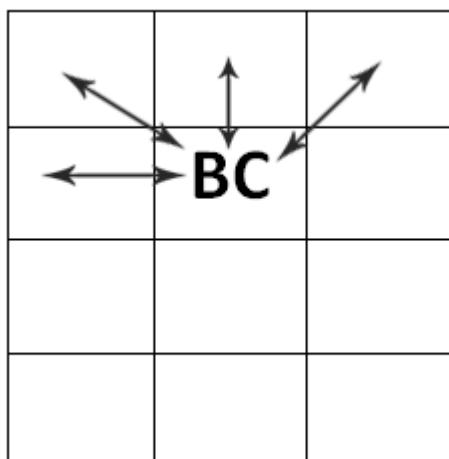


Рисунок 1.6 – Тестирующий алгоритм GALPAT

Листинг 1.1 – Псевдокод реализации алгоритма GALPAT

```

function GALPAT =
  Write background 0;
  for i in 0 .. 1 do
    for BC in 0 .. N-1 do
      Complement BC;
      for OC in 0 .. N-1 and OC != BC do
        Read BC;
        Read OC
      end for
      Complement BC;
      Write background 1;
    end for
  end

```

1.3.4 Тестирующий алгоритм WALPAT

Алгоритм WALPAT (Walking Pattern) очень похож на тест GALPAT. Различие лишь в том, что базовая ячейка считывается после того, как все остальные ячейки были просканированы. Сложность составляет $2n^2$. Тест проиллюстрирован на рисунке 1.7.

Покрывающая способность данного теста, как и алгоритма Galloping Pattern, достаточно высокая, все неисправности типа Afs, TFs, CFs и SAFs обнаруживаются и локализуются. Псевдокод теста приведён в листинге 1.2.

Листинг 1.2 – Псевдокод реализации алгоритма WALPAT

```

function WALPAT =
  Write background 0;
  for i in 0 .. 1 do
    for BC in 0 .. N-1 do
      Complement BC;

```


1.3.6 Тестирующий алгоритм Butterfly

На протяжении теста базовая ячейка сдвигается по матрице памяти. При каждом сдвиге в цикле проверяются её ближайшие соседи в радиусе не более половины от длины строки и столбца матрицы ячеек. Тест представлен на рисунке 1.9.

		6			
		1			
9	4	5,10	2	7	
		3			
		8			

Рисунок 1.9 – Тестирующий алгоритм Butterfly

Сложность алгоритма составляет $5n \log n$. Тест обнаруживает все неисправности типа SAFs и некоторые из неисправностей типа AFs. Псевдокод теста приведён в листинге 1.3.

Листинг 1.3 – Псевдокод реализации алгоритма Butterfly

```
function Butterfly =  
  Write background 0;  
  For i in 0 .. 1 do  
    For BC in 0 .. N-1 do  
      Complement BC;  
      dist = 1;  
      While dist <= mdist do // mdist < 0.5 col/row length  
        Read cell dist north from BC;  
        Read cell dist east from BC;  
        Read cell dist south from BC;  
        Read cell dist west from BC;  
        Read BC;  
        dist *=2;  
      end while  
      Complement BC  
    end for  
    Write background 1  
  end for  
end
```

1.3.7 Тестирующий алгоритм Surround Disturb

Алгоритм SD (Surround Disturb) Позволяет проверить поведение ячеек в строке, когда комплиментарные значения записываются в ячейки, расположенные на соседних рядах матрицы. Этот алгоритм разработан на пред-

положении, что ячейки памяти наиболее чувствительны к помехам от своих ближайших соседей. Тест представлен на рисунке 1.10.

		1		
	0	0	0	
		1		

Рисунок 1.10 – Тестирующий алгоритм Surround Disturb

Псевдокод алгоритма представлен в листинге 1.4.

Листинг 1.4 – Псевдокод реализации алгоритма Surround Disturb

```

For each cell [p,q] // row = p, col = q
{
  write 0 in cell[p,q-1];
  write 0 in cell[p,q];
  write 0 in cell [p, q+1];
  write 1 in cell[p-1,q];
  read 0 from cell[p,q+1];
  write 1 in cell[p+1,q];
  read 0 from cell[p, q-1];
  read 0 from cell[p,q];
}

```

1.3.8 Маршевые тесты

Маршевые тесты являются самыми простыми и оптимальными для тестирования памяти. Они имеют линейную сложность и симметрию. Маршевый тест состоит из конечной последовательности маршевых элементов. В свою очередь маршевый элемент состоит из последовательности операций чтения и записи, которые применяются ко всем ячейкам памяти в возрастающем или убывающем порядке адресов. Все операции маршевого элемента выполняются до перехода к следующей ячейке памяти. Обозначается маршевый элемент следующим образом: rx - чтение из ячейки памяти, в которой должно храниться значение x ; wx - запись в ячейку памяти значения x ; \uparrow - возрастающий порядок адресов (от 0 до $n-1$); \downarrow - убывающий порядок адресов (от $n-1$ до 0); \updownarrow - возрастающий или убывающий порядок адресов (направление не важно).

Пример маршевого теста: $\{\uparrow(w1); \uparrow(r1, w0)\}$. Такой маршевый тест предполагает следующий алгоритм:

- а) проходя от 0 до n-1 ячейки памяти записать 1 в i-ую ячейку;
- б) проходя от 0 до n-1 ячейки памяти прочитать значение из i-ой ячейки, в которой должна быть 1, и записать в эту ячейку 0.

В таблице 1.1 представлены некоторые маршевые тесты.

Таблица 1.1 – Маршевые тесты

Наименование	Алгоритм
MATS	$\{\Downarrow(w0); \Downarrow(r0, w1); \Downarrow(r1)\}$
MATS+	$\{\Downarrow(w0); \uparrow(r0, w1); \Downarrow(r1, w0)\}$
MATS++	$\{\Downarrow(w0); \uparrow(r0, w1); \Downarrow(r1, w0, r0)\}$
March X	$\{\Downarrow(w0); \uparrow(r0, w1); \Downarrow(r1, w0); \Downarrow(r0)\}$
March C-	$\{\Downarrow(w0); \uparrow(r0, w1); \uparrow(r1, w0); \Downarrow(r0, w1); \Downarrow(r1, w0); \Downarrow(r0)\}$
March A	$\{\Downarrow(w0); \uparrow(r0, w1, w0, w1); \uparrow(r1, w0, w1); \Downarrow(r1, w0, w1, w0); \Downarrow(r0, w1, w0)\}$
March B	$\{\Downarrow(w0); \uparrow(r0, w1, r1, w0, r0, w1); \uparrow(r1, w0, w1); \Downarrow(r1, w0, w1, w0); \Downarrow(r0, w1, w0)\}$

Важным этапом в эволюции функциональных тестов ОЗУ стало применение неразрушающих методов тестирования. Необходимость в их использовании обусловлена критичностью современных приложений (сетевые серверы, системы управления) к показателям надежности [7]. В таких случаях необходимо использовать тестовые алгоритмы, которые не разрушали бы содержимое ОЗУ. Преобразовать разрушающий маршевый тест в его неразрушающий аналог достаточно просто. Неразрушающие тесты имеют ряд обязательных требований: любой маршевый элемент начинается операцией чтения, отсутствует начальная инициализация памяти ($\Downarrow(wx)$). После выполнения теста необходимо вернуть память в её первоначальное состояние. Идея неразрушающих тестов состоит в замене операций чтения и записи конкретных значений на операции чтения и записи прямых и обратных (комплиментарных) значений, хранящихся в ячейках памяти. Теперь неразрушающие операции маршевых тестов примут следующий вид: wdc, wd - запись dc и d значений в ячейку памяти; rdc, rd - чтение из ячейки памяти ожидаемых значений dc и d. Значение dc есть комплиментарное значение к d. На примере теста MATS+ его неразрушающий аналог будет выглядеть следующим образом: $\{\uparrow(rd, wdc); \Downarrow(rdc, wd)\}$.

1.4 Обзор существующих аналогов

В открытых источниках информация о существующих аналогах не найдена. Даже если таковые имеются, то они скорее всего являются закрытыми инструментами компаний, занимающихся проблемой тестирования памяти. Вследствие этого факта сравнение с ними невозможно. Исходя из отсутствия свободно распространяемых утилит по верификации тестирующих алгоритмов оперативных запоминающих устройств, разработка данного программного средства имеет смысл.

1.5 Постановка задачи

В результате выполнения дипломного проекта должна быть разработана библиотека, написанная на языке C++, для верификации тестирующих алгоритмов оперативных запоминающих устройств со следующими спецификациями:

- разрабатываемое ПО должно иметь возможность запускаться под платформами Windows(7,8,10);
- разрабатываемое ПО должно позволять производить верификацию тестирующих алгоритмов ОЗУ, в зависимости от требований конечного пользователя;
- подвергаться тестированию должна динамическая память, симулируемая инструментом DRAMSim2;
- время работы ПО должно быть приемлемым.

2 ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ

Выбор технологий является важным предварительным этапом разработки сложных информационных систем. Платформа и язык программирования, на котором будет реализована система, заслуживает большого внимания, так как исследования показали, что выбор языка программирования влияет на производительность труда программистов и качество создаваемого ими программного кода.

Ниже перечислены некоторые факторы, повлиявшие на выбор используемой технологий:

- разрабатываемое ПО должно иметь возможность запускаться под платформами Windows(7,8,10);
- дальнейшей поддержкой проекта, возможно, будут заниматься разработчики, не принимавшие участие в выпуске первой версии;
- имеющийся разработчик имеет опыт работы с объекто-ориентированными языками программирования.

Основываясь на опыте работы имеющихся программистов разрабатывать ПО целесообразно с помощью языка C++. Приняв во внимание необходимость обеспечения доступности дальнейшей поддержки ПО, возможно, другой командой программистов, необходимость работы с различными ОС, целесообразно не использовать малоизвестные и сложные языки программирования. Так как при разработке программного средства будет использоваться симулятор DRAM, написанный на C++, целесообразно выбрать именно этот язык для создания внешней среды для этого симулятора. Таким образом, с учетом вышеперечисленных факторов, целесообразно остановить выбор на следующих технологиях:

- операционные системы: семейство Windows(7,8,10);
- язык программирования C++.

Для реализации поставленной задачи предпочтительно использовать стандартную библиотеку STL без использования сторонних библиотек для обеспечения простой компиляции проекта под любой операционной системой. Высокий уровень абстракции языка, полноценные механизмы ООП, большое количество контейнеров и алгоритмов библиотеки STL позволяют наиболее просто и «красиво» решить поставленную задачу. Разрабатываемое программное обеспечение в некоторой степени использует данное преимущество языка программирования.

Далее проводится характеристика используемых технологий и языка программирования более подробно.

2.1 Язык программирования C++

C++ - компилируемый, статически типизированный язык программирования общего назначения. Синтаксис C++ унаследован от языка C. Одним из принципов разработки было сохранение совместимости с C. Тем не менее, C++ не является в строгом смысле надмножеством C; множество программ, которые могут одинаково успешно транслироваться как компиляторами C, так и компиляторами C++, довольно велико, но не включает все возможные программы на C [8].

Язык поддерживает такие парадигмы программирования, как процедурное программирование, объектно-ориентированное программирование, обобщённое программирование, обеспечивает модульность, отдельную компиляцию, обработку исключений, абстракцию данных, объявление типов (классов) объектов, виртуальные функции. Стандартная библиотека включает, в том числе, общеупотребительные контейнеры и алгоритмы. C++ сочетает свойства как высокоуровневых, так и низкоуровневых языков. В сравнении с его предшественником – языком C, – наибольшее внимание уделено поддержке объектно-ориентированного, а также обобщённого программирования [9].

C++ широко используется для разработки программного обеспечения, являясь одним из самых популярных языков программирования. Область его применения включает создание операционных систем, разнообразных прикладных программ, драйверов устройств, приложений для встраиваемых систем, высокопроизводительных серверов, а также развлекательных приложений (игр). Существует множество реализаций языка C++, как бесплатных, так и коммерческих и для различных платформ. Например, на платформе x86 это GCC, Visual C++, Intel C++ Compiler, Embarcadero (Borland) C++ Builder и другие. C++ оказал огромное влияние на другие языки программирования, в первую очередь на Java и C# [10].

Главное нововведение C++ - механизм классов, дающий возможность определять и использовать новые типы данных. Программист описывает внутреннее представление объекта класса и набор функций-методов для доступа к этому представлению. Одной из заветных целей при создании C++ было стремление увеличить процент повторного использования уже написанного кода. Концепция классов предлагала для этого механизм наследования. Наследование позволяет создавать новые (производные) классы с расширенным представлением и модифицированными методами, не затрагивая при этом скомпилированный код исходных (базовых) классов.

Вместе с тем наследование обеспечивает один из механизмов реализации полиморфизма - базовой концепции объектно-ориентированного программирования, согласно которой, для выполнения однотипной обработки разных типов данных может использоваться один и тот же код. Собственно, полиморфизм - тоже один из методов обеспечения повторного использования программного кода [11].

Введение классов не исчерпывает всех новаций языка C++. В нем реализованы полноценный механизм структурной обработки исключений, отсутствие которого в C значительно затрудняло написание надежных программ, механизм шаблонов - изощренный механизм макрогенерации, глубоко встроенный в язык, открывающий еще один путь к повторной используемости кода, и многое другое.

Таким образом, генеральная линия развития языка была направлена на расширение его возможностей путем введения новых высокоуровневых конструкций при сохранении сколь возможно полной совместимости с ANSI C. Конечно, борьба за повышение уровня языка шла и на втором фронте - те же классы позволяют при грамотном подходе упрятывать низкоуровневые операции, так что программист фактически перестает непосредственно работать с памятью и системно-зависимыми сущностями.

2.2 Симуляторы ДОЗУ

Для моделирования процессов оперативной памяти используется множество симуляторов, от упрощенных моделей, которые лишь регулируют временные задержки и пропускную способность доступа к памяти, и до детализированных симуляторов, которые могут точно отразить сложное поведение современных систем памяти. К сожалению, в открытом доступе находится малая их часть. В этом разделе будут рассмотрены и сравнены некоторые симуляторы, информация о которых открыта.

2.2.1 DRAMSim2

Одним из популярных симуляторов памяти является DRAMSim2, разработанный в университете Мэриленд, США. DRAMSim2 – это потактовый симулятор оперативной памяти, реализованный на языке C++ как объектно-ориентированная модель DDR2/3 памяти. Помимо прочего данный симулятор обеспечивает надежный инструмент для визуализации и сравнения влияния системных параметров памяти на ключевые метрики производительности, такие как пропускная способность, временные задержки и потребляемую мощность [12].

Ядро симулятора заключено в единый объект, который называется MemorySystem. Он состоит из двух компонент: контроллера памяти и ОЗУ. Диаграмма главных компонентов симулятора DRAMSim2 и их связей изображена на рисунке 2.1.

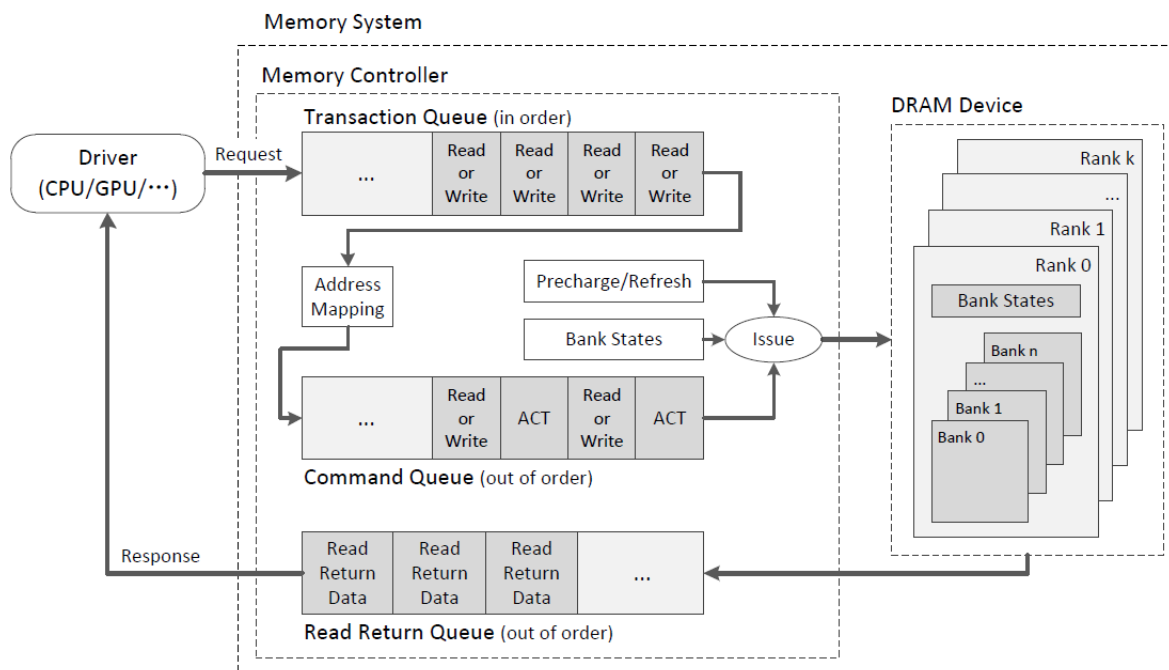


Рисунок 2.1 – Диаграмма компонентов симулятора DRAMSim2

Для объекта MemorySystem необходимо наличие двух ini-файлов: для инициализации системы в целом и для описания ОЗУ. Файл инициализации ОЗУ содержит описание параметров конкретного устройства, таких как временные ограничения и параметры энергопотребления устройства. Значения этих параметров можно найти в справочных данных, предоставляемых производителями ОЗУ.

Пакет DRAMSim2 содержит несколько ini-файлов для устройств Micron DDR2/3 различного объёма и скорости. Ini-файл системы содержит параметры, независимые от конкретного типа ОЗУ. Они включают в себя такие параметры, как схемы отображения адресов, опции отладки, структура очереди контроллера памяти и другие детали симуляции.

После создания объекта MemorySystem код драйвера памяти должен зарегистрировать функции обратного вызова, чтобы получать уведомления о выполнении запросов. На этом этапе инициализация симулятора завершена. Код драйвера должен вызывать специальную функцию для каждого системного такта, а также другую функцию, для добавления запроса к па-

мости. По окончании обработки запроса симулятор информирует драйвер о выполненном действии, вызывая функцию обратного вызова [13].

Поскольку вся логика взаимодействия с симулятором заключена в простом интерфейсе с минимальным количеством функций, симулятор легко внедрить в любую систему, будь то простой драйвер или потактовый симулятор ЦП, такой как MARSSx86. DRAMSim2 может быть скомпилирован как автономный исполняемый модуль, либо же в качестве библиотеки. Соответственно симулятор работает в двух режимах : автономном и встроенном в другую систему. В автономном режиме симулятор считывает список команд из trace-файла, хранящегося на диске. Во встроенном режиме симулятор предоставляет базовую функциональность для создания объекта системы памяти и добавления запросов к нему.

DRAMSim2 не использует никаких сторонних библиотек, кроме стандартной библиотеки C++ STL. Благодаря этому симулятор легко скомпилировать под любой операционной системой, на которой установлен компилятор GNU C++. Для запуска симулятора под ОС Windows потребуются Cygwin или MinGW [14].

Поскольку производители отказываются публиковать детали работы их контроллеров памяти, DRAMSim2 моделирует современные DDR2/3 контроллеры в общем виде. Запросы от драйвера (любой модуль, который выдаёт запросы, например центральный процессор) накапливаются в очереди транзакций в порядке их выполнения. Эти транзакции преобразуются в команды ОЗУ и направляются в очередь команд, а затем выдаются запоминающему устройству. Контроллер памяти отслеживает состояние каждого банка памяти, на основе чего определяет, какой запрос должен быть передан ОЗУ следующим. Взаимодействие драйвера, контроллера памяти и DRAM отображено на рисунке 2.1. Выдача запросов в случайном порядке помогает оптимизировать использование банка и тем самым повысить пропускную способность памяти и уменьшить задержки.

После получения устройством DRAM команды и данных от контроллера памяти, список состояний банков используется для проверки ошибок, чтобы убедиться, что время полученной команды верное.

В дополнение к моделированию изменения состояния системы во время операций чтения и записи, контроллер памяти DRAMSim2 так же моделирует эффект восстановления памяти ОЗУ. Моделирование регенерации памяти имеет важное значение, так как операции восстановления являются основным источником задержек при выполнении запросов к памяти. Запросы чтения, поступившие во время восстановления памяти, должны ожидать

обработки намного дольше, чем другие запросы.

Достоинством симулятора DRAMSim2 является простой интерфейс, общедоступный исходный код, который можно скачать с сайта [GitHub.com](https://github.com/DramSim2), подробная документация, наличие готовых конфигурационных файлов для различных типов памяти. Симулятор легко скомпилировать под любой операционной системой, при этом не требуется установка дополнительных библиотек. В качестве серьёзного недостатка является отсутствие в бесплатной версии симулятора возможности записи и считывания данных, но относительная простота кода и грамотная архитектура системы позволяет программисту доработать этот момент.

2.2.2 Ramulator

Ramulator – это быстрый, потактовый DRAM симулятор, целью создания которого была потребность в расширяемом симуляторе, который может быть легко модифицирован под любой стандарт памяти, чтобы иметь представление о достоинствах современных DRAM [15].

В основе Ramulator лежит обобщённый шаблон для моделирования систем DRAM, который позже конкретизируется под определённый стандарт памяти. Благодаря такому гибкому дизайну симулятор может обеспечить поддержку широкого спектра стандартов DRAM: DDR3/4, LPDDR3/4, GDDR5, WIO1/2, HBM, а также академических стандартов (SALP, AL-DRAM, TLDRAM, RowClone, and SARP). Важно заметить, что данный симулятор не жертвует скоростью памяти для обеспечения такой масштабируемости.

Ramulator основывается на важном наблюдении: любая DRAM может быть представлена в виде иерархии конечных автоматов, где поведение каждого автомата, как и иерархии в целом, диктуется стандартами DRAM. Из любого предоставленного стандарта DRAM симулятор извлекает полную спецификацию для иерархии конечных автоматов и их поведения, которую затем объединяет в единый класс (например, `DDR3.h/cpp`). Вследствие этого симулятор легко переконфигурировать под другой стандарт памяти, что не требует изменений в коде.

В листинге 2.1 представлен класс DRAM, который является обобщённым шаблоном для построения иерархии (дерева) конечных автоматов (узлов). Экземпляр класса DRAM – это один из узлов в дереве, который будет заключать в себе конкретную реализацию стандарта DRAM (DRAM<DDR3>).

Листинг 2.1 – Метод записи конкретных бит в слове

```
// DRAM.h
```

```

template <typename T>
class DRAM {
    DRAM<T>* parent;
    vector<DRAM<T>*> children;
    T::Level level;
    int index;
    //more code...
};

// DDR3.h/cpp
class DDR3 {
    enum class Level {
        Channel,
        Rank,
        Bank,
        Row,
        Column,
        MAX
    };

    //more code...
};

```

На рисунке 2.2 представлено полностью реализованное дерево, состоящее из узлов различного уровня: от канала до банка. Вместо того, чтобы создавать отдельный класс для каждого уровня (DDR3 Channel, DDR3 Rank, DDR3 Bank), Ramulator просто представляет каждый уровень как очередное свойство узла дерева. Это свойство может быть легко переназначено для адаптации иерархий под различные уровни. Ramulator так же предоставляет контроллер памяти, который взаимодействует с деревом через его корень. Каждый раз, когда контроллер памяти инициирует запрос или операцию, производится обход дерева, затрагивающий только необходимые узлы во время выполнения процесса.

Система конечных автоматов подразумевает набор состояний, переход между которыми осуществляется под внешним воздействием. Конечный автомат рассматриваемого симулятора хранит в себе текущее состояние, которое может принимать одно из значений, определяемых в классе конкретного вида памяти (например, DDR3). Узел может переходить из одного состояния в другое при получении одной из команд, которые так же определяются классом симулируемой памяти. Так же узел хранит в себе таблицу временных параметров, по которой определяется ближайшее время, через которое узел сможет принять ту или иную команду. Цель этой таблицы – предотвратить преждевременный переход узла из одного состояния в другое. У каждого узла в дереве есть 3 функции: decode, check и update. Команда, поступившая к конкретному узлу, должна декодироваться. Функция decode

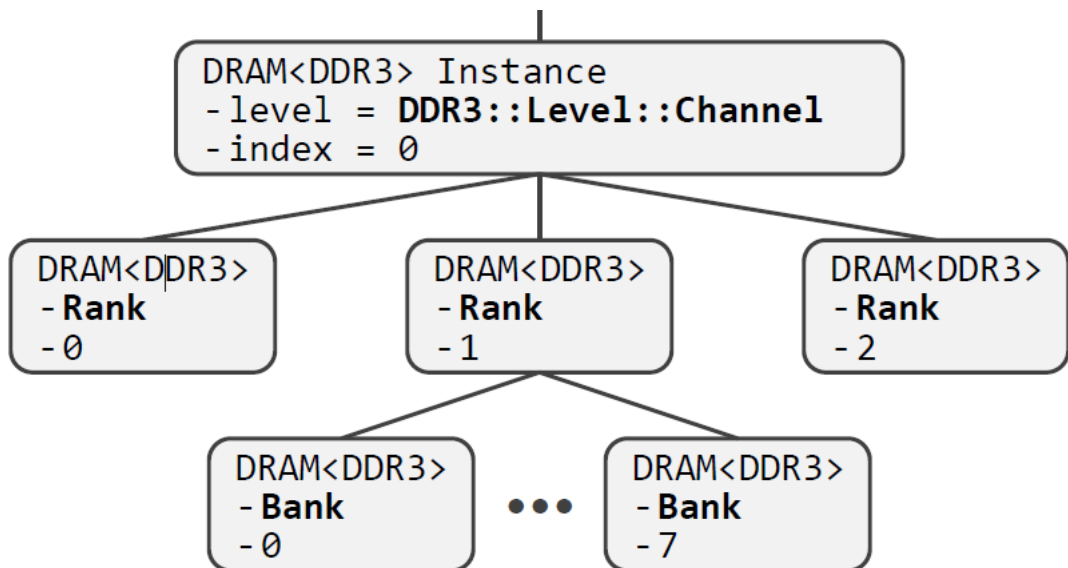


Рисунок 2.2 – Дерево конечных автоматов памяти DDR3

распознаёт команду и проверяет возможность её выполнения, основываясь на статусе узла. Например, команда чтения не может быть выполнена, если rank отключен или bank закрыт. Для данной команды и адреса функция decode вернёт команду, которая должна быть выполнена перед командой чтения. Даже если нет команд, которые должны быть предварительно выполнены перед операцией, это не означает, что поступившая команда может выполняться тут же. Например, bank может быть не готов к чтению, если он был активирован недавно. Для этих целей функция check сообщает, можно ли выполнить команду прямо сейчас (на текущем цикле). Если функция check разрешила выполнение команды, то контроллер памяти выполняет ее. Функция update в зависимости от команды модифицирует статус узла и его временные параметры в таблице.

Кроме того Ramulator поддерживает унифицированный контроллер памяти, который совместим со всеми стандартами DRAM(поддерживаемых в симуляторе). Контроллер памяти содержит три очереди запросов к памяти: чтение, запись и техническое обслуживание. В то время, как очереди чтения/записи наполняются запросами, поступившими от внешнего источника команд (trace-файл с командами чтения и записи), очередь технического обслуживания узлов наполняется другим видом команд (обновление, отключение питания, автообновление), генерируемых внутри контроллера памяти, когда они необходимы. Для обработки запросов трёх очередей контроллер памяти взаимодействует с иерархическим деревом DRAM используя три функции узла, описанные выше.

Таким образом Ramulator представляет собой расширяемый DRAM симулятор, обеспечивающий потактовую модель широкого разнообразия стандартов: DDR3/4, LPDDR3/4, GDDR5, WIO1/2, HBM, SALP, ALDRAM, TL-DRAM, RowClone, and SARP. Модульная архитектура системы позволяет легко дополнять симулятор новыми стандартами. Для некоторых стандартов Ramulator способен предоставлять отчеты по энергопотреблению. Ramulator оснащен простым контроллером памяти, представляющим из себя внешнее API для отправки и получения запросов памяти. Симулятор доступен в двух различных форматах : для автономного использования и для встроенного в систему режима. Для компиляции симулятора требуется компилятор GNU C++ и библиотека clang++. Из-за использования дополнительной библиотеки могут возникнуть трудности при компиляции проекта в операционной системе Windows. В бесплатной версии симулятора отсутствует возможность реального хранения данных.

2.2.3 Обоснование выбора симулятора DRAM

Среди прочих симуляторов, информация о которых есть в открытом доступе, можно отметить еще несколько симуляторов.

DrSim – потактовый симулятор DRAM памяти. Поддерживает стандарты памяти DDR2, DDR3 и LPDDR2. Написан на языке C++. Для компиляции потребуется GNU C++ и GNU Make. В целом этот симулятор очень напоминает DRAMSim2. Он так же работает в двух режимах: автономном и встроенном. В конфигурационных файлах задаются параметры для конкретного типа DRAM, а также основные параметры для системы в целом. В режиме автономной работы команды считываются из trace-файла, результаты выполнения выводятся на консоль. Во встроенном режиме внешней системе нужно выполнять роль генератора тактов для памяти, а также отправлять запросы. По завершении выполнения запроса вызывается функция обратного вызова, чтобы позволить пользователю увидеть результаты выполнения команды [16].

USIMM – симулятор DRAM, поддерживающий стандарт DDR3. Симулятор написан на языке C, без использования дополнительных библиотек. Главный модуль симулятора запускает цикл, в котором он получает команду из очереди запросов и запускает функцию `updateMemory`. Эту функцию реализует контроллер памяти, который проверяет временные параметры DRAM, чтобы определить, какие команды могут быть выполнены на данном цикле. Пользователь должен предоставить функцию `scheduler`, которая будет выдавать команды для каналов в каждый цикл работы памяти. В

конфигурационных файлах описываются параметры для памяти [17].

Среди рассмотренных симуляторов Ramulator является наиболее гибкой и хорошо спроектированной моделью динамической памяти. Однако при попытке компиляции проекта под ОС Windows возникли проблемы с библиотекой clang++. Данный симулятор поддерживает широкий круг стандартов памяти, однако в рамках задачи дипломного проекта такой ширины не требуется. Симулятор DRAMSim2 является относительно простым и небольшим проектом, который легко скомпилировать под любой операционной системой, при этом не требуется инсталляция дополнительных библиотек. Простой интерфейс позволяет легко встраивать симулятор в другие системы. Симуляторы DrSim и USIMM во многом уступают симулятору DRAMSim2. Как по наличию скромной документации, так и по грамотности построения структуры исходного кода и удобству использования. В открытом доступе отсутствуют симуляторы с возможностью хранения реальных данных, вследствие чего симулятор DRAMSim2 вполне подойдёт для дополнительных модификаций и адаптаций под текущую задачу.

3 АРХИТЕКТУРА И МОДУЛИ СИСТЕМЫ

3.1 Основные компоненты симулятора

Для симуляции динамической оперативной памяти был выбран симулятор DRAMSim2. Данный симулятор обладает рядом преимуществ: простота, независимость от сторонних библиотек, хорошо организованная внутренняя архитектура программы.

Симулятор является потактовым, вследствие чего все основные классы унаследованы от единого базового класса SimulatorObject. В листинге 3.1 представлена его структура. Объект этого класса имеет единственное поле - счетчик currentClockCycle, который инкрементируется методом step. Таким образом все компоненты симулятора работают в единой временной плоскости. Метод update каждый производный класс имплементирует по-своему, вкладывая нужную ему функциональность.

Листинг 3.1 – Класс SimulatorObject

```
class SimulatorObject
{
public:
    uint64_t currentClockCycle;

    void step();
    virtual void update()=0;
};
```

Самыми важными компонентами симулятора являются три класса: MemorySystem, MemoryController и DRAMDevice. Класс MemorySystem является ядром симулятора, включающим в себя всю его функциональность. Класс имеет простой интерфейс, через который внешний драйвер взаимодействует с памятью. С помощью метода AddTransaction драйвер добавляет запрос в память. Метод Update нужно вызывать в конце каждого цикла. Отсчет циклов производит драйвер памяти.

Класс MemoryController накапливает запросы от драйвера в очереди транзакций, транслируя их затем в команды для устройства памяти. Сама память представлена классом DRAMDevice, включающим в себя классы Rank и Bank. В зависимости от конфигурации каждый Rank содержит в себе определенное количество объектов Bank. В классе Bank матрица ячеек памяти представлена картой, где номер ряда является ключом, а значение представляет собой вектор слов, чьи порядковые номера соответствуют колонкам ячеек. Класс Bank представлен в листинге 3.2.

Листинг 3.2 – Класс Bank

```
class Bank
{
public:
    Bank();
    ~Bank();

    void read(BusPacket *busPacket);
    void write(const BusPacket *busPacket);

public:
    BankState currentState;

private:
    typedef map<uint64_t, std::auto_ptr<std::vector<uint16_t>> > row_map;
    row_map rowEntries;
};
```

Для данного симулятора в открытом доступе имеется версия с возможностью хранения данных, что значительно ускорило разработку программного средства. Симулируемая память является слово-ориентированной. По этой причине появилась необходимость добавления функциональности для записи конкретных бит, а не только слов, в ячейки памяти. В листинге 3.3. представлен метод writeBit, который записывает 0 или 1 (в зависимости от параметра set) в позицию bit в ячейке.

Листинг 3.3 – Метод записи конкретных бит в слове

```
void Bank::writeBit(int row, int col, int bit, bool set)
{
    auto row_ = rowEntries[row].get();

    if (set)
        (*row_)[col] |= 1 << bit;
    else
    {
        if ((*row_)[col] & (1 << bit))
            (*row_)[col] ^= 1 << bit;
    }
}
```

В целом работа всей системы представляет собой следующее: контроллер памяти накапливает в очереди транзакций запросы от драйвера; на каждом новом цикле работы системы контроллер памяти извлекает одну транзакцию из очереди, транслирует её адрес в значения номера ранка, банка, ряда и колонки ячейки памяти, преобразует пакет транзакции в команду для устройства памяти и помещает этот пакет в очередь команд; из очереди команд по одному пакету в цикл извлекается команда и контроллер памяти определяет, какое действие нужно совершить устройству памяти. Решение

зависит от множества факторов: состояния банков (например, команда чтения не может быть выполнена, если адресуемая ячейка не активна, в этом случае нужна предварительная команда Activate); контроллер регенерации, который может приостановить все транзакции в системе и запустить процесс обновления памяти. Каждую команду записи в память отслеживает адаптивный сигнатурный анализатор, о котором будет рассказано в следующих подразделах. Контроллер регенерации с помощью сигнатурного анализатора выясняет, были ли неполадки в памяти за время её работы, и при необходимости запускает тесты. Внутри самого ОЗУ находится контроллер неисправностей, который имитирует поведение ячеек с физическими неисправностями, а также контролирует процесс разрядки ячеек при длительном отсутствии команды Refresh. ОЗУ в свою очередь в ответ на команды чтения возвращает контроллеру памяти считанные данные, которые сохраняются в очереди пакетов чтения и затем отправляются драйверу. Основные модули системы представлены на рисунке 3.1.

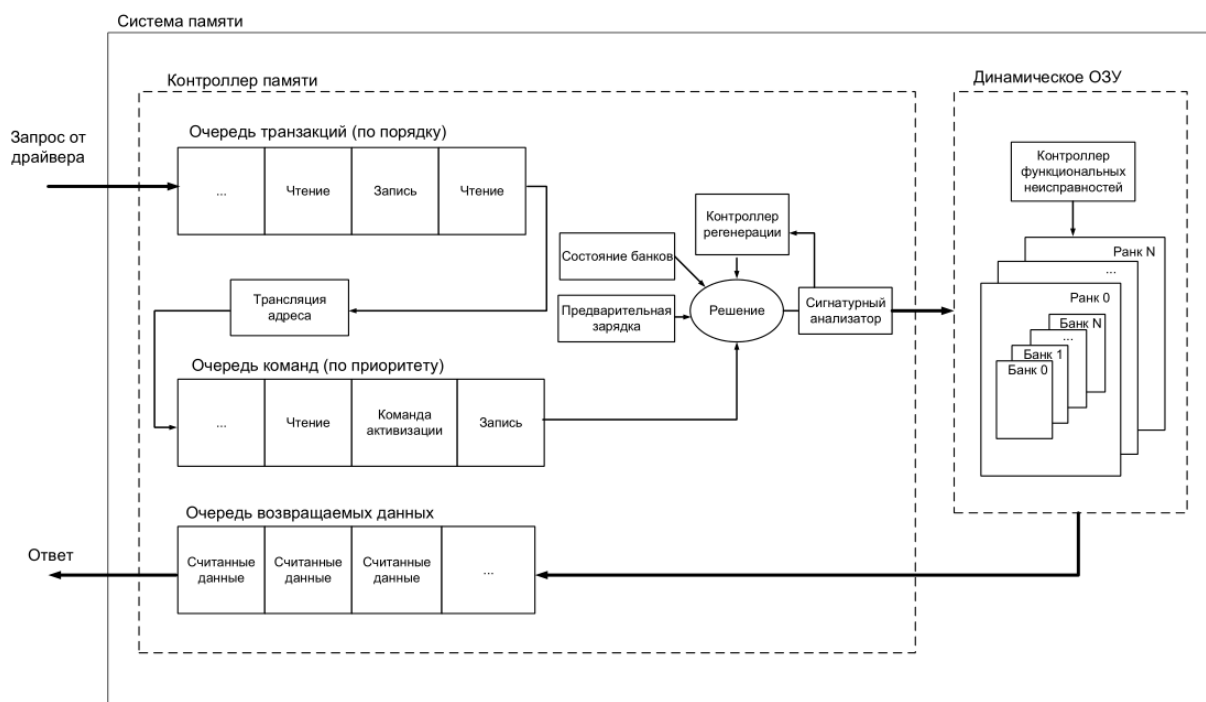


Рисунок 3.1 – Основные компоненты симулятора ОЗУ

3.2 Симуляция процесса деградации памяти

Источником ошибок может является свойство конденсаторов памяти разряжаться. Это происходит в случаях, когда период обновления памяти сдвинут и ячейки успевают терять свой заряд.

Этот процесс симулирует класс `Discharger`, который следит за периодами обновления памяти, и если такой период не наступил вовремя, то начинается постепенная деградация отдельных бит памяти в случайных местах (биты теряют свои данные и хранят 0).

Для генерации случайных адресов ячеек и позиций бит в них используется класс `uniform_int_distribution` стандартной библиотеки шаблонов языка C++. Он генерирует случайные числа в диапазоне $[a, b]$, подчинённые дискретному равномерному распределению, которое представлено на формуле 3.1 следующей функцией распределения:

$$P(i|a,b) = \frac{1}{b - a + 1}, a \leq i \leq b. \quad (3.1)$$

Плотность распределения показана на рисунке 3.2.

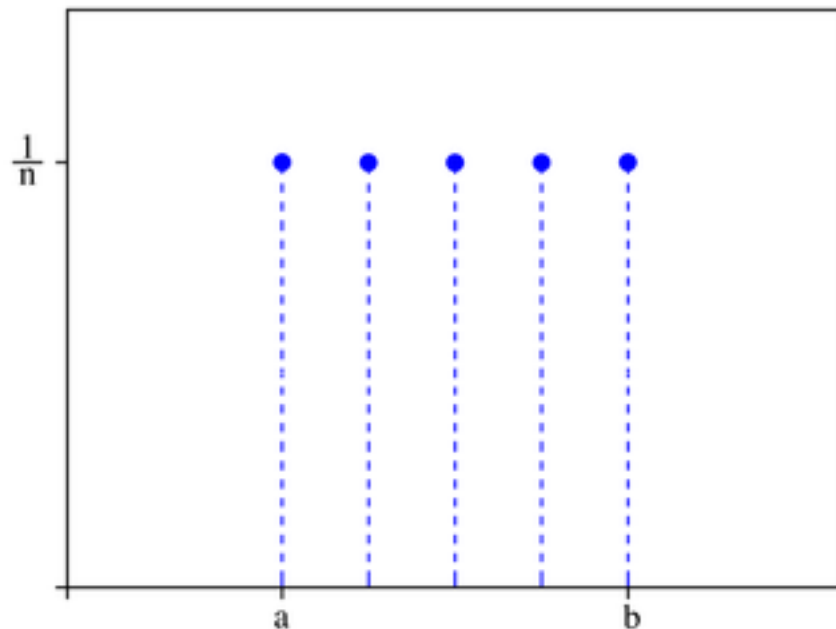


Рисунок 3.2 – Плотность распределения дискретного равномерного распределения

В зависимости от того, сколько циклов прошло с момента, когда должен был начаться процесс обновления памяти, количество генерируемых адресов, а следовательно количество бит, значения которых стекают в 0, возрастает по формуле:

$$f(t) = e^{\frac{\sqrt{t}}{5.2}}. \quad (3.2)$$

По умолчанию период обновления памяти $T_{refresh} = 2600$ циклов. 1 цикл равен 3 нсек. Таким образом каждые 7800 нсек память должна обнов-

ляться. В итоге в соответствии с формулой примерно через 2000 циклов(с учетом погрешности генератора адресов, т.к. один и тот же адрес может сгенерироваться несколько раз) все ячейки обернутся в 0, если соответствующий процесс регенерации памяти так и не был запущен. График функции представлен на рисунке 3.3.

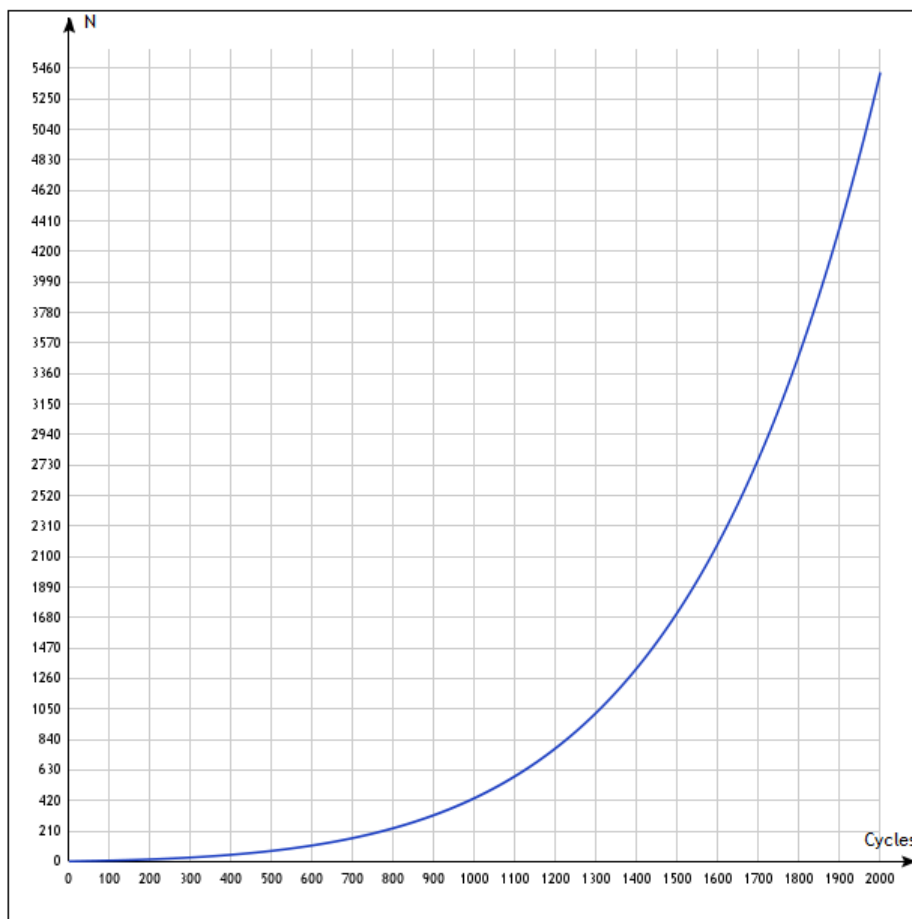


Рисунок 3.3 – График экспоненциальной функции 3.2

3.3 Симуляция процесса регенерации памяти

Заряд конденсатора динамического ОЗУ со временем уменьшается вследствие утечки, поэтому для сохранения содержимого памяти процесс регенерации каждой ячейки памяти должен производиться через определенное время. На время регенерации взаимодействие ОЗУ с внешними устройствами(драйвером) должно быть приостановлено, т.е. путем перевода этих устройств в режим ожидания.

В первую очередь необходимо выяснить, произошли ли ошибки во время работы памяти. Наиболее экономичным по временным затратам и эф-

фективным методом является алгоритм адаптивного сжатия выходных данных (АСВД) (Self-Adjusting Output Data Compression - SAODC). Подробно данная технология рассмотрена в работах [18] и [19]. Данный подход позволяет полностью избежать временных издержек для вычисления эталонной сигнатуры. Согласно этой концепции эталонная характеристика (сигнатура) S_{ref} начального содержимого бит-ориентированного ОЗУ вычисляется как сумма по модулю два всех адресов ячеек, содержащих значение 1.

АСА обладает следующими свойствами [7]:

- сигнатуры для произвольного состояния памяти и обратного ему равны между собой;
- конечное значение сигнатуры не зависит от способа и направления движения по памяти при вычислении сигнатуры;
- при использовании АСА в качестве сигнатурного анализатора все однократные неисправности памяти будут обнаруживаться и диагностироваться; при этом адрес неисправности вычисляется путем побитового сложения по модулю 2 двух сигнатур, полученных до и после проявления неисправности;
- все двукратные неисправности обнаруживаются.

Пример вычисления эталонной сигнатуры для ОЗУ с 8-ю ячейками представлен на рисунке 3.4.

При таком методе изменения в памяти не потребуют заново вычислять эталонную сигнатуру, достаточно просто сложить по модулю два адрес изменившейся ячейки со значением эталонной сигнатуры. Обновление сигнатуры выглядит следующим образом:

$$S_{ref}^{new} = S_{ref}^{old} \oplus a \cdot (M[a]^{new} \oplus M[a]^{old}), \quad (3.3)$$

где a —адрес изменившейся ячейки;

$M[a]^{new}$ —новое значение ячейки;

$M[a]^{old}$ —старое значение ячейки.

Для слово-ориентированного ОЗУ применяется та же схема сжатия данных, но потребуются некоторые дополнительные действия. Пусть имеется память с матрицей ячеек 16 на 16. Т.е. 16 рядов и 16 колонок. Каждая ячейка имеет объём 8 бит. Применяя метод АСА (адаптивный сигнатурный анализатор), теперь нужно оперировать адресами не ячеек, а бит памяти. Таким образом для 5-го бита в ячейке, расположенной на 4-ом ряду и 2-ой колонке адрес будет представлен числом 0x215 в шестнадцатиричной системе счисления, или числом 0100 0010 101 в двоичной. Построение адреса

ОЗУ

Адреса	Ячейки
000	0
001	0
010	1
011	0
100	0
101	1
110	1
111	0

$$S_{ref} = 010 \oplus 101 \oplus 110 = 001$$

Рисунок 3.4 – Вычисление эталонной сигнатуры для бит-ориентированного ОЗУ

бита пояснено на рисунке 3.5.

Адрес

ряд			колонка				бит			
0	1	0	0	0	0	1	0	1	0	1

Рисунок 3.5 – Представление адреса бита в слово-ориентированном ОЗУ

У метода АСА есть недостаток, который заключается в том, что 0-вой адрес ячейки никак не повлияет на общую сумму адресов, т.к. $0 \oplus S = S$. Исходя из этого вместе с данным методом используется также бит четности [7]. Бит четности - это дополнительный бит в младшем разряде адреса ячейки, который всегда содержит 1-цу. Таким образом даже нулевой адрес ячейки повлияет на общую сумму адресов. Но главная особенность бита четности заключается в том, что с его помощью можно определить, какой кратности произошла ошибка. Если тестовая и эталонная сигнатуры не совпали и бит четности равен нулю, тогда произошло четное количество

ошибок, в противном случае нечетное.

После каждой операции записи в память, эталонная сигнатура обновляется. При наступлении периода регенерации, высчитывается тестовая сигнатура S_{test} и затем сравнивается с эталонной S_{ref} . Если сигнатуры совпали - значит с памятью всё в порядке, в противном случае произошли ошибки. При однократной ошибке достаточно просто найти ядрес ячейки и позицию бита, в котором произошла ошибка. Сложение двух сигнатур $S_{test} \oplus S_{ref}$ дает адрес бита с изменившимися данными. Таким образом процедура регенерации памяти состоит из следующих шагов:

а) при наступлении периода обновления памяти вычисляется тестовая сигнатура и сравнивается с эталонной;

б) если сигнатуры не совпали и бит четности равен 1-це, то предполагаем, что произошла однократная ошибка. Инвертируется бит по адресу $S_{test} \oplus S_{ref}$;

в) если сигнатуры не совпали и бит четности равен 0-лю, то произошло четное количество ошибок, переход к шагу д;

г) если сигнатуры совпали, то была однократная ошибка, которую удалось устранить. В противном случае переход к шагу д;

д) запуск неразрушающего маршевого теста для выявления функциональных неисправностей памяти;

е) если тест ничего не обнаружил, значит была многократная ошибка, в противном случае неисправность, которую нужно диагностировать;

ж) повторяется шаг а.

Блок-схема алгоритма регенерации представлена на рисунке 3.6.

За регенерацию памяти отвечает класс `RegenerationController`. Класс `SAODCCController` обеспечивает вычисление рабочей и эталонной сигнатур. Метод обновления эталонной сигнатуры при каждой операции чтения представлен в листинге 3.4

Листинг 3.4 – Обновление эталонной сигнатуры ОЗУ

```
void SAODCCController::UpdateRef(BusPacket* packet)
{
    if (packet->busPacketType == DATA)
    {
        uint16_t oldData = dramDevice->read(packet->address);
        uint16_t newData = packet->data->getData();
        if (oldData != newData)
        {
            int address = GetAddress(packet->address, data);
            RefSignature ^= address;
        }
    }
}
```

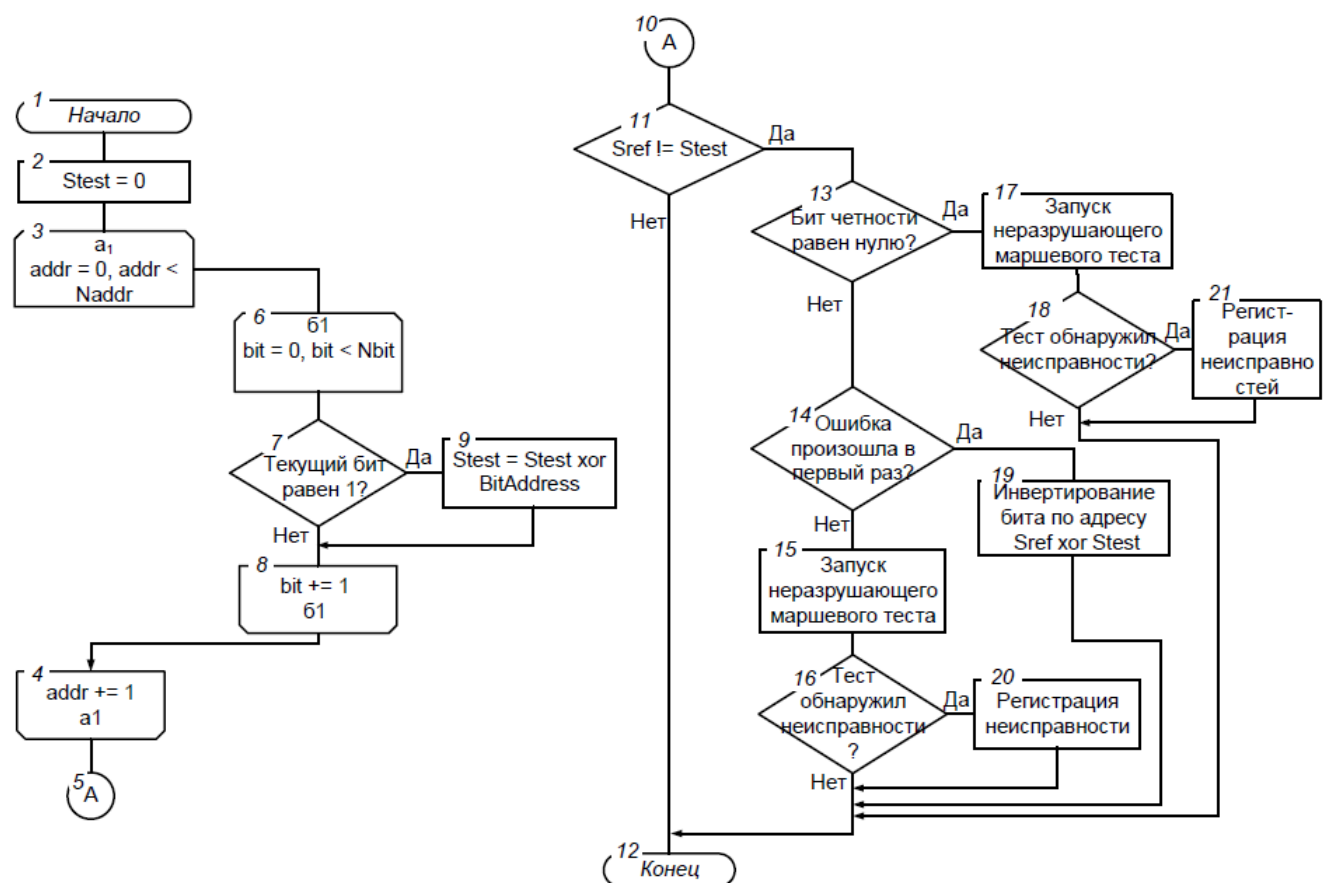


Рисунок 3.6 – Схема алгоритма регенерации памяти ОЗУ

```

}

int SAODCController::GetAddress(int rank, int bank, int row, int col, uint16_t
    data)
{
    //get 'bit address sum'
    if (!data) return 0;

    int bitSum = 0;
    int bitCount = 0; // '1' bit count
    for (int i = 0; (unsigned)i < DEVICE_WIDTH; i++)
    {
        if (data & (1 << i))
        {
            bitSum ^= i;
            bitCount++;
        }
    }
    if (bitCount % 2 == 0)
        addr.Clear();

    addr.bit = bitSum;
    int address = addr.GetPhysical();
}

```

```

    if (bitCount % 2 != 0)
        AddParityBit(address);

    return address;
}

```

3.4 Представление маршевых тестов

Для выявления неисправностей ОЗУ решено использовать маршевые тесты из-за их простоты и эффективности. Все маршевые тесты построены по единому принципу и различаются лишь набором элементов. Поэтому целесообразно представить такие тесты с помощью обобщенной структуры, которая может подстроиться под любой тест. Направление перебора адресов отражается элементами MD_UP, MD_DOWN и MD_BOTH перечисления MarchDirection. Операции чтения и записи также представлены элементами перечисления MarchOperation. Элементы MO_RD и MO_RDC обозначают операцию чтения прямого и обратного значения в буфер из ячейки памяти, элементы MO_WD и MO_WDC описывают операции записи прямого и обратного значения из буфера в ячейку памяти. Фаза маршевого теста отражена в структуре MarchPhase, представленной в листинге 3.5.

Листинг 3.5 – Структура фазы маршевого теста

```

struct MarchPhase
{
    MarchDirection direction;
    std::vector<MarchOperation> elements;
};

```

Таким образом маршевый тест представляется вектором, содержащим определённое количество структур MarchPhase. Инициализация вектора элементов маршевого теста March C- представлена в листинге 3.6.

Листинг 3.6 – Инициализация вектора элементов маршевого теста March C-

```

void InitializeTest()
{
    std::vector<MarchPhase> phases;
    phases.push_back({ MD_UP, { MO_RD, MO_WDC } });
    phases.push_back({ MD_UP, { MO_RDC, MO_WD } });
    phases.push_back({ MD_DOWN, { MO_RD, MO_WDC } });
    phases.push_back({ MD_DOWN, { MO_RDC, MO_WD } });
    phases.push_back({ MD_DOWN, { MO_RD } });
};

```

Другие виды маршевых тестов инициализируются подобным образом. Программа поддерживает следующие маршевые тесты: March C-, March

A, March B, March X, March Y, MATS, MATS+, MATS++.

Логика создания и выполнения маршевых тестов находится в классе `MarchTestController`. Запуск теста производит класс `RegenerationController`. Для сжатия данных используется адаптивный сигнатурный анализатор, рассмотренный в предыдущем пункте. На этапе инициализации теста в качестве эталонной сигнатуры берётся уже вычисленная ранее рабочая сигнатура класса `RegenerationController`. Далее на каждой фазе маршевого теста вычисляется тестовая сигнатура и в конце фазы сравнивается с эталонной. При правильном функционировании памяти сигнатуры должны всегда совпадать вне зависимости от операций, производимых элементами маршевого теста. Это происходит потому, что маршевый тест является неразрушающим, а также благодаря свойству АСА, которое заключается в равенстве сумм адресов ячеек, содержащих 1, и адресов ячеек, содержащих 0. Выполнение маршевого теста занимает определённое время, а потому выполнение отдельных его фаз разбито на несколько циклов работы симулятора. Выполнение теста приводится в листинге 3.7.

Листинг 3.7 – Функция выполнения маршевого теста

```
void MarchTestController::Update()
{
    if (!state.testStarted) return;

    saodc.ClearTestSig();
    MarchPhase& phase = phases[state.phase];
    int addrStart = 0;
    int counter = 0;
    int bottom = 0;
    int top = NUM_BANKS * NUM_COLS * NUM_ROWS - 1;

    if (phase.direction == MD_UP || phase.direction == MD_BOTH)
    {
        addrStart = bottom;
        counter = 1;
    }
    else if (phase.direction == MD_DOWN)
    {
        addrStart = top;
        counter = -1;
    }
    for (int addr = addrStart; addr >= bottom && addr <= top; addr += counter)
    {
        uint16_t buffer = 0;
        uint16_t old = 0;
        for (auto& el : phase.elements)
        {
            RunElement(el, addr, buffer, old);
        }
    }
}
```

```

    }
    int err = saodc.Compare();
    if (err)
    {
        state.testPassed = false;
        cout << "[MarchTest] Errors are detected while running phase! Signature
            sum is " << err << "(" << addrTranslator.GetDescription(err, true)
            << ")" << endl;
    }
    state.phase++;
    if (state.phase == phases.size())
    {
        state.testCompleted = true;
        state.testStarted = false;
    }
}

```

Выполнение каждой отдельной операции маршевого элемента приведено в листинге 3.8.

Листинг 3.8 – Выполнение элемента маршевого теста

```

void MarchTestController::RunElement(int element, int address, uint16_t &
    buffer, uint16_t& oldValue)
{
    int r = 0, b = 0, row = 0, col = 0;
    addrTranslator.Translate(address, r, b, row, col);
    uint16_t data = 0;
    switch (element)
    {
        case MO_RD:
            data = dramDevice->read(r, b, row, col);
            buffer = data;
            break;
        case MO_RDC:
            data = dramDevice->read(r, b, row, col);
            buffer = ~data;
            break;
        case MO_WD:
            dramDevice->write(r, b, row, col, buffer);
            break;
        case MO_WDC:
            dramDevice->write(r, b, row, col, ~buffer);
            break;
    }
    //convertData
    if (element == MO_RD || element == MO_RDC)
    {
        saodc.UpdateTestSig(addr, data ^ oldValue);
        oldValue = data;
    }
}

```

3.5 Моделирование функциональных неисправностей ОЗУ

Для моделирования функциональных неисправностей ОЗУ используется обобщенная структура `Fault`, представленная в листинге 3.9.

Листинг 3.9 – Структура описания функциональной неисправности ОЗУ

```
struct Fault
{
    FaultType type;
    uint16_t victimVal;
    uint16_t agressorVal;
    int agressorAddr;
};
```

Поле `type` может принимать одно из следующих значений: SAF, TF, AF, CFfin, CFid, CFst. Они объединены в перечислении `FaultType`. Поле `victimVal` для разных моделей неисправностей имеет различное значение, но обобщенно оно хранит значение ячейки-жертвы. Так для неисправностей типа SAF `victimVal` будет содержать значение ячейки памяти, которое эта ячейка хранит постоянно и изменить его не может. Для неисправности TF это поле хранит значение, перейдя в которое ячейка превращается в константную, т.е. не может поменять своё состояние. Для неисправностей CFid и CFst данное поле хранит значение, в которое зависимая ячейка принудительно переходит (для CFid) или в которое она может перейти только при определённом значении в ячейке-агрессоре (для CFst). Для неисправностей типа CFfin и AF данное поле не используется.

Поле `agressorAddr` хранит адрес ячейки-агрессора. Это поле используется в неисправностях типа CF и AF. Поле `agressorVal` используется только для CF. Поле `agressorVal` хранит значение ячейки-агрессора, при котором зависимая ячейка может перейти в состояние `victimVal` (для CFst) или значение, при переходе в которое, ячейка-жертва насильно меняет своё значение на `victimVal` (для CFfin и CFid). Для неисправности типа AF поле `agressorAddr` хранит адрес ячейки, при записи значения в которую записываемое значение дублируется по адресу ячейки-жертвы.

Класс, контролирующий поведение ячеек в соответствии с различными моделями неисправностей, называется `FaultController` и находится в классе `DRAMDevice`. При каждой операции записи в память `FaultController` вмешивается в работу системы и имитирует поведение неисправностей. Все основные виды неисправностей выявляются на протяжении работы симулятора, однако при неисправностях типа SAF необходимо инициализировать значения ячеек памяти на старте симулятора.

Метод, анализирующий операцию записи в ячейку памяти и принимающий решение о выполнении тех или иных действий, соответствующих моделям неисправностей, представлен в листинге 3.10.

Листинг 3.10 – Реализация поведения ячеек с функциональными неисправностями

```
void FaultController::DoFaults(Address addr, uint16_t data)
{
    uint16_t oldData = dramDevice->read(addr);
    uint16_t bitMask = oldData ^ data;

    for (unsigned i = 0; i < DEVICE_WIDTH; i++)
    {
        if (bitMask & (1 << i))
        {
            addr.bit = i;
            DoOperationOnBit(addr, data & (1 << i) ? 1 : 0, oldData & (1 << i) ?
                1 : 0);
        }
    }
}

void FaultController::DoOperationOnBit(Address address, uint16_t newVal,
    uint16_t oldVal)
{
    int addr = address.GetPhysical();
    if (IsFaulty(addr))
    {
        Fault fault = GetCellAttributes(addr, false);
        switch (fault.type)
        {
            case SAF: break;
            case TF:
            {
                if (oldVal != fault.victimValue)
                {
                    WriteBit(address, newVal);
                }
                break;
            }
            case CFin:
            {
                WriteBit(address, newVal);
                break;
            }
            case CFid:
            {
                WriteBit(address, newVal);
                break;
            }
            case CFst:
            {
```



```

        if (newVal != fault.victimValue)
            WriteBit(address, newVal);
        else
        {
            Address agrAdr(fault.agressorAddress, true);
            if (dramDevice->readBit(agrAdr) == fault.agressorValue)
                WriteBit(address, newVal);
        }
        break;
    }
    default: break;
}
}
else
{
    if (IsAgressor(addr))
    {
        Fault fault = GetCellAttributes(addr, true);
        switch (fault.type)
        {
            case CFin:
            {
                if (newVal == fault.agressorValue && newVal != oldVal)
                {
                    Address vicAdr(fault.victimAddress, true);
                    (*dramDevice->ranks)[vicAdr.rank].banks[vicAdr.bank].
                        invertBit(vicAdr);
                }
                break;
            }
            case CFid:
            {
                if (newVal == fault.agressorValue && newVal != oldVal)
                {
                    Address vicAdr(fault.victimAddress, true);
                    WriteBit(vicAdr, fault.victimValue);
                }
                break;
            }
            case AF:
            {
                Address vicAdr(fault.victimAddress, true);
                WriteBit(vicAdr, newVal);
                break;
            }
            default :
                break;
        }
        WriteBit(address, newVal);
    }
    else
    {
        WriteBit(address, newVal); } } }

```

3.6 Верификация неразрушающих маршевых тестов

Для верификации были взяты следующие неразрушающие маршевые тесты: March C-, March A, March B, March X, March Y, MATS, MATS+, MATS++. Описание тестов находится в таблице 3.1.

Таблица 3.1 – Неразрушающие маршевые тесты

Название	Сложность	Описание теста
March C-	9N	$\{\uparrow (rd, wdc); \uparrow (rdc, wd); \downarrow (rd, wdc); \downarrow (rdc, wd); \updownarrow (rd)\}$
March A	14N	$\{\uparrow (rd, wdc, wd, wdc); \uparrow (rdc, wd, wdc); \downarrow (rdc, wd, wdc, wd); \downarrow (rd, wdc, wd)\}$
March B	16N	$\{\uparrow (rd, wdc, rdc, wd, rd, wdc); \uparrow (rdc, wd, wdc); \downarrow (rdc, wd, wdc, wd); \downarrow (rd, wdc, wd)\}$
March X	5N	$\{\uparrow (rd, wdc); \downarrow (rdc, wd); \updownarrow (rd)\}$
March Y	5N	$\{\uparrow (rd, wdc, rdc); \downarrow (rdc, wd, rd); \updownarrow (rd)\}$
MATS	3N	$\{\updownarrow (rd, wdc); \updownarrow (rdc)\}$
MATS+	4N	$\{\uparrow (rd, wdc); \downarrow (rdc, wd)\}$
MATS++	5N	$\{\uparrow (rd, wdc); \downarrow (rdc, wd, rd)\}$

Для верификации потребовалось запустить каждый тест для каждой модели неисправности, находящейся в памяти. Всего были проверены 26 неисправностей: 2 вида SAF, 2 вида TF, 2 вида AF, 4 вида CFin, 8 видов CFid и 8 видов CFst. В качестве неисправностей AF были взяты только два случая: когда адрес ячейки, чье значение при записи дублируется в ячейку-жертву, меньше, чем адрес ячейки-жертвы, и наоборот.

Для неисправностей SAF, TF, AF и неисправностей типа CFin результаты тестов приведены в таблице 3.2.

Для неисправностей CFid результаты тестов приведены в таблице 3.3.

Для неисправностей CFst результаты тестов приведены в таблице 3.4.

Обобщенные результаты по всем тестам представлены в таблице 3.5.

Из результатов тестов видно, что наибольшей покрывающей способностью обладает маршевый тест March C-. По сравнению с тестами March A и March B, которые также имеют хорошие показатели обнаружения неисправностей, тест March C- не только охватывает больше неисправностей, но и является оптимальным по скорости по сравнению с предыдущими тестами.

Таблица 3.2 – Покрывающая способность неразрушающих маршевых тестов неисправностей SAF, TF, AF и CFin

Тест	Модели неисправностей									
	SAF		TF		AF		CFin			
	0	1	↓	↑	$\wedge(b)$	$\vee(b)$	$\wedge(\uparrow, b)$	$\wedge(\downarrow, b)$	$\vee(\uparrow, b)$	$\vee(\downarrow, b)$
March C-	+	+	+	+	+	+	+	+	+	+
March A	+	+	+	+	+	+	+	+	+	+
March B	+	+	+	+	+	+	+	+	+	+
March X	+	+	+	+	+	+	+	+	+	+
March Y	+	+	+	+	+	+	+	+	+	+
MATS	+	+	+	-	+	-	+	-	+	-
MATS+	+	+	+	-	+	+	+	-	+	+
MATS++	-	-	-	+	+	+	+	-	+	+

Таблица 3.3 – Покрывающая способность неразрушающих маршевых тестов неисправностей CFid

Тест	Модели неисправностей							
	CFid							
	$\wedge(\uparrow, 0)$	$\wedge(\uparrow, 1)$	$\wedge(\downarrow, 0)$	$\wedge(\downarrow, 1)$	$\vee(\uparrow, 0)$	$\vee(\uparrow, 1)$	$\vee(\downarrow, 0)$	$\vee(\downarrow, 1)$
March C-	+	+	+	+	+	+	+	+
March A	+	+	+	+	+	+	+	+
March B	+	+	+	+	+	+	+	+
March X	-	+	-	+	+	-	+	-
March Y	-	+	-	+	+	-	+	-
MATS	-	+	-	-	+	-	-	-
MATS+	-	+	-	-	+	-	+	-
MATS++	-	+	-	-	+	-	+	-

Таблица 3.4 – Покрывающая способность неразрушающих маршевых тестов неисправностей CFst

Тест	Модели неисправностей							
	CFst							
	\wedge (0,0)	\wedge (0,1)	\wedge (1,0)	\wedge (1,1)	\vee (0,0)	\vee (0,1)	\vee (1,0)	\vee (1,1)
March C-	+	+	+	+	+	+	+	+
March A	+	+	-	-	-	+	+	+
March B	+	+	-	-	-	+	+	+
March X	+	+	-	-	-	-	+	+
March Y	+	+	-	-	-	-	+	+
MATS	-	+	-	-	-	-	-	+
MATS+	-	+	-	-	-	-	-	+
MATS++	+	-	-	-	-	-	+	-

Таблица 3.5 – Покрывающая способность неразрушающих маршевых тестов

Тест	Модели неисправностей					
	SAF	TF	AF	CFin	CFid	CFst
March C-	2/2	2/2	2/2	4/4	8/8	8/8
March A	2/2	2/2	2/2	4/4	8/8	5/8
March B	2/2	2/2	2/2	4/4	8/8	5/8
March X	2/2	2/2	2/2	4/4	4/8	4/8
March Y	2/2	2/2	2/2	4/4	4/8	4/8
MATS	2/2	1/2	1/2	2/4	2/8	2/8
MATS+	2/2	1/2	2/2	3/4	3/8	2/8
MATS++	0/2	1/2	2/2	3/4	3/8	2/8

4 ТЕСТИРОВАНИЕ ПРИЛОЖЕНИЯ

Программное средство разрабатывалось, как библиотека. Для использования симулятора необходима внешняя среда в качестве драйвера, который создаст объект класса `MemorySystem` и будет добавлять транзакции в очередь, а также отсчитывать циклы работы симулятора. Тестирование предполагает проверку работоспособности самой библиотеки в рамках драйвера. Поэтому перед запусками тестов-кейсов необходимо заранее проделать следующие шаги:

- а) создать консольное приложение на языке C++;
- б) подключить библиотеку в проект;
- в) создать объект класса `MemorySystem`;
- г) создать очередь для транзакций;
- д) организовать цикл для отсчета тактов для симулятора.

Тестирование предполагает наличие файлов конфигурации и файла инициализации для симулятора. Все значения, записываемые в память, не должны превышать максимальный объём хранимых данных в ячейке.

Для оценки правильности работы программного средства было проведено тестирование. Тест-кейсы для проверки базовой функциональности программного средства представлены в таблицах 4.1 и 4.2.

Таблица 4.1 – Тестирование корректной конфигурации памяти

Название тест-кейса и его описание	Ожидаемый результат	Фактический результат
Верная конфигурация памяти 1) Инициализировать объект <code>MemorySystem</code> входными параметрами: путь к <code>ini</code> -файлу, путь к файлу конфигурации памяти, объем памяти; 2) запустить цикл работы симулятора.	1) В консоли отображается информация о запуске системы, количестве ранков и объеме симулируемой памяти; 2) объем памяти и количество ранков совпадает со значениями в файле конфигурации памяти.	Тест пройден

Таблица 4.2 – Тестирование хранения данных в памяти

Название тест-кейса и его описание	Ожидаемый результат	Фактический результат
<p>Хранение данных в памяти</p> <p>1) Инициализировать объект MemorySystem входными параметрами: путь к ini-файлу, путь к файлу конфигурации памяти, объем памяти;</p> <p>2) запустить цикл работы симулятора;</p> <p>3) добавить в очередь транзакций операцию записи значения по адресу;</p> <p>4) добавить в очередь транзакций операцию чтения по этому же адресу.</p>	<p>1) В консоли отображается информация о запуске системы и объеме симулируемой памяти;</p> <p>2) отображается информация о получении пакета с операцией записи;</p> <p>3) отображается информация о возвращаемом пакете с результатом чтения;</p> <p>4) результат чтения равен значению, записанному ранее операцией записи в память.</p>	Тест пройден

Тестирование процесса разрядки конденсаторов памяти в случае сдвига периода обновления памяти представлено в таблице 4.3.

Таблица 4.3 – Тестирование процесса деградации памяти при сдвинутом периоде обновления

Название тест-кейса и его описание	Ожидаемый результат	Фактический результат
<p>Деградация памяти</p> <p>1) Инициализировать объект MemorySystem входными параметрами: путь к ini-файлу, путь к файлу конфигурации памяти, объем памяти, количество циклов, на которое сдвинут период обновления памяти;</p> <p>2) добавить в очередь транзакций операцию записи значения, все биты которого находятся в состоянии логической единицы(максимально возможное значение, хранимое ячейкой);</p> <p>3) запустить цикл работы симулятора;</p> <p>4) через равные промежутки времени в цикле добавлять операции чтения по адресу, в который ранее было записано значение.</p>	<p>1) В консоли отображается информация о запуске системы и объеме симулируемой памяти;</p> <p>2) спустя период времени, после которого должен начаться период обновления памяти(в соответствии с файлом конфигурации) каждая новая операция чтения возвращает постепенно убывающее значение.</p>	Тест пройден

Следует протестировать поведение системы при наличии в памяти многократных ошибок. Для этого запускаются тесты из таблицы 4.4.

Таблица 4.4 – Тестирование процесса регенерации памяти при сдвинутом периоде обновления

Название тест-кейса и его описание	Ожидаемый результат	Фактический результат
Регенерация памяти 1) Инициализировать объект MemorySystem входными параметрами: путь к ini-файлу, путь к файлу конфигурации памяти, объем памяти, количество циклов, на которое сдвинут период обновления памяти; 2) запустить цикл работы симулятора.	1) В консоли отображается информация о запуске системы и объеме симулируемой памяти; 2) после наступления периода обновления появляется сообщение о несовпадении эталонной и тестовой сигнатур; 3) появляется уведомление о запуске маршевого теста; 4) маршевый тест пройден, неисправности не обнаружены.	Тест пройден

В открытых источниках находится информация о покрывающей способности некоторых маршевых тестов, например MATS, MATS++ и других. На основе этой информации строятся тесты, для проверки правильности работы тестирующих алгоритмов ОЗУ. Тесты предполагают наличие файла неисправностей в формате csv. Общий алгоритм тестирования проверки работоспособности маршевых тестов един и сводится к таблице 4.5.

Таблица 4.5 – Тестирование поведения системы при наличии в памяти неисправностей

Название тест-кейса и его описание	Ожидаемый результат	Фактический результат
<p>Работоспособность маршевых тестов</p> <p>1) Инициализировать объект MemorySystem входными параметрами: путь к ini-файлу, путь к файлу конфигурации памяти, объем памяти, путь к файлу неисправностей;</p> <p>2) запустить цикл работы симулятора.</p>	<p>1) В консоли отображается информация о запуске системы и объеме симулируемой памяти;</p> <p>2) после наступления периода обновления появляется сообщение о несовпадении эталонной и тестовой сигнатур;</p> <p>3) появляется уведомление о запуске маршевого теста;</p> <p>4) маршевый тест не пройден, обнаружены неисправности.</p>	Тест пройден

Таким образом, результат тестирования подтверждает, что программное средство верификации алгоритмов тестирования оперативных запоминающих устройств функционирует в полном соответствии со спецификацией требований.

5 МЕТОДИКА ИСПОЛЬЗОВАНИЯ РАЗРАБОТАННОГО ПРИЛОЖЕНИЯ

Программное средство представляет из себя библиотеку, написанную на языке C++. Использование симулятора возможно в рамках драйвера. Драйвер должен создать объект класса `MemorySystem`, инициализировать его и зарегистрировать функции обратной связи. Организовав цикл, драйвер должен отсчитывать такты для симулятора, вызывая на каждой итерации метод `update` класса `MemorySystem`. Команды чтения и записи отправляются системе памяти через метод `addTransaction`.

Объект класса `MemorySystem` инициализируется значениями:

- а) `deviceIniFilename` – путь к файлу конфигурации памяти;
- б) `systemIniFilename` – путь к файлу инициализации системы;
- в) `traceFilename` – путь к файлу регистрации, если он не задан, то все сообщения выводятся на консоль;
- г) `megsOfMemory` – объём симулируемой памяти в мегабайтах;
- д) `faultFilePath` – путь к файлу неисправностей;
- е) `marchTest` – название маршевого теста, который будет запускаться для выявления неисправностей;
- ж) `refreshPeriodShift` – количество тактов, на которое будет сдвинут период обновления памяти.

Функции обратной связи получают уведомления о выполнении транзакций. Необходимо зарегистрировать две функции, одна из которых будет получать уведомления о выполнении операции записи в память, а вторая функция получает результат операции чтения. Сигнатуры функций обратной связи представлены в листинге 5.1.

Листинг 5.1 – Сигнатуры функций обратного вызова

```
void read_complete(uint64_t address, uint16_t data, size_t);  
void write_complete(unsigned id, uint64_t address, uint64_t cycle);
```

Создание объекта `MemorySystem` и регистрация функций обратного вызова представлена в листинге 5.2.

Листинг 5.2 – Инициализация системы и регистрация функций обратного вызова

```
string systemIniFilename = "system.ini";  
string deviceIniFilename = "ini/DDR2_micron_1Mb.ini";  
unsigned megsOfMemory = 1;  
  
memory = new MemorySystem(0, deviceIniFilename, systemIniFilename, "", "  
SimulationResults.txt", megsOfMemory, 0, "MATS++", "faults.csv");
```

```
ReadDataCB *read_cb = new Callback<TestingSystem, void, uint64_t, uint16_t,
    size_t>(this, &TestingSystem::read_complete);
TransactionCompleteCB *write_cb = new Callback<TestingSystem, void, unsigned,
    uint64_t, uint64_t>(this, &TestingSystem::write_complete);
memory->RegisterCallbacks(read_cb, write_cb, 0);
```

Файл конфигурации памяти имеет расширение ini и состоит из следующих параметров:

- а) NUM_BANKS – количество банков в одном ранке;
- б) NUM_ROWS – количество рядов в матрице запоминающих элементов банка;
- в) NUM_COLS – количество колонок в матрице запоминающих элементов банка;
- г) DEVICE_WIDTH – объем одной ячейки матрицы запоминающих элементов банка;
- д) REFRESH_PERIOD – время в наносекундах, через которое память переходит в режим обновления, причем один цикл работы системы длится условно три наносекунды.

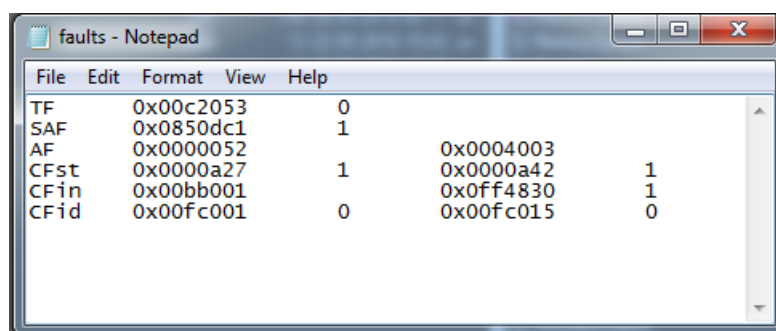
Остальные параметры являются специфическими настройками временных характеристик конкретного вида памяти.

Файл инициализации системы имеет расширение ini и состоит из следующих параметров:

- а) TRANS_QUEUE_DEPTH – длина очереди транзакций;
- б) CMD_QUEUE_DEPTH – длина очереди команд;
- в) DEBUG_TRANS_Q – флаг, регулирующий трансляцию пакетов, поступающих в очередь транзакций;
- г) DEBUG_CMD_Q – флаг, регулирующий трансляцию пакетов, поступающих в очередь команд;
- д) DEBUG_ADDR_MAP – флаг, регулирующий трансляцию сообщений от дешифратора адресов;
- е) DEBUG_BUS – флаг, регулирующий трансляцию пакетов, проходящих по шине данных;
- ж) DEBUG_BANKSTATE – флаг, регулирующий трансляцию состояния банков памяти.

Файл неисправностей имеет расширение csv. Файл представляет собой текстовый документ, в котором сохранена табличная информация. Все колонки таблицы разделены символом табуляции. Всего различается пять колонок. Первая колонка содержит название модели неисправности. Это поле может принимать одно из следующих значений: SAF, TF, AF, CFin,

CFid, CFst. Вторая и четвертая колонка содержит адреса ячейки-жертвы и ячейки-агрессора в шестнадцатиричном формате. Третья и пятая колонка содержит значения ячейки-жертвы и ячейки-агрессора в соответствии с моделью неисправности. Для неисправностей типа SAF и TF четвертое поле (адрес ячейки-агрессора) не используется, так же, как и пятое поле (значение ячейки-агрессора). Пример файла, описывающего неисправности, приведён на рисунке 5.1.



File	Edit	Format	View	Help
TF	0x00c2053	0		
SAF	0x0850dc1	1		
AF	0x0000052		0x0004003	
CFst	0x0000a27	1	0x0000a42	1
CFin	0x00bb001		0x0ff4830	1
CFid	0x00fc001	0	0x00fc015	0

Рисунок 5.1 – Файл описания неисправностей памяти

На рисунке 5.2 приведён скриншот работы программы при исправной памяти. Операции четня и записи транслируются с двух сторон: от драйвера и от контроллера памяти.

```

===== MemorySystem 0 =====
== Loading device model file 'ini/DDR2_micron_1Mb.ini' ==
== Loading system model file 'system.ini' ==
TOTAL_STORAGE : 0.25MB | 1 Ranks |
[Testing System] Write data at address 0x12458 on cycle 10
~ current clcclCycle: 10 ~
[DPKT] Rank receiving <-- : DATA: '65535'
~ current clcclCycle: 14 ~
[DPKT] Rank returning --> : DATA: '65535'
[Testing System] Read data [65535] from address 0x12458
[Testing System] Write data at address 0x458 on cycle 29
~ current clcclCycle: 29 ~
[DPKT] Rank receiving <-- : DATA: '115'
~ current clcclCycle: 33 ~
[DPKT] Rank returning --> : DATA: '115'
[Testing System] Read data [115] from address 0x458
[Testing System] Write data at address 0x3758 on cycle 48
~ current clcclCycle: 48 ~
[DPKT] Rank receiving <-- : DATA: '21'
~ current clcclCycle: 52 ~
[DPKT] Rank returning --> : DATA: '21'
[Testing System] Read data [21] from address 0x3758
I received a REFRESH command on cycle 2601
=====

```

Рисунок 5.2 – Работа симулятора при исправном устройстве памяти

При сдвинутом периоде обновления памяти произвольные ячейки ОЗУ

теряют свой заряд. Чем больше времени прошло с момента не наступившего сигнала Refresh, тем больше ячеек повредится. Эти изменения в памяти засекает адаптивный сигнатурный анализатор, который при наступлении периода обновления памяти вычисляет рабочую сигнатуру. Пример работы симулятора со сдвинутым периодом обновления приведен на рисунке 5.3.

```

~ current clocCycle: 10 ~
[DPKT] Rank receiving <-- : DATA: '65535'
~ current clocCycle: 14 ~
[DPKT] Rank returning --> : DATA: '65535'
[Testing System] Read data [65535] from address 0x12458
[Testing System] Write data at address 0x458 on cycle 29
~ current clocCycle: 29 ~
[DPKT] Rank receiving <-- : DATA: '115'
~ current clocCycle: 33 ~
[DPKT] Rank returning --> : DATA: '115'
[Testing System] Read data [115] from address 0x458
[Testing System] Write data at address 0x3758 on cycle 48
~ current clocCycle: 48 ~
[DPKT] Rank receiving <-- : DATA: '21'
~ current clocCycle: 52 ~
[DPKT] Rank returning --> : DATA: '21'
[Testing System] Read data [21] from address 0x3758
I received a REFRESH command on cycle 5201
[Regeneration Controller] Detected error(s) in memory.
    Signature Sum = 126080(r:0, b:0, row:246, col:4, bit:0)
[Regeneration Controller] March Test Started.
[Regeneration Controller] March Test Passed.
=====
===== Printing Statistics [id:0]=====
Total Return Transactions : 6 (12 bytes) aggregate average bandwidth 0.000GB/s
-Rank 0 :
  -Reads : 3 (6 bytes)
  -Writes : 3 (6 bytes)
  -Bandwidth / Latency (Bank 0): 0.000 GB/s          106.000 ns
  -Bandwidth / Latency (Bank 1): 0.000 GB/s          0.000 ns
  -Bandwidth / Latency (Bank 2): 0.000 GB/s          0.000 ns
  -Bandwidth / Latency (Bank 3): 0.000 GB/s          0.000 ns
  -Bandwidth / Latency (Bank 4): 0.000 GB/s          0.000 ns
  -Bandwidth / Latency (Bank 5): 0.000 GB/s          0.000 ns
  -Bandwidth / Latency (Bank 6): 0.000 GB/s          0.000 ns
  -Bandwidth / Latency (Bank 7): 0.000 GB/s          0.000 ns
== Power Data for Rank 0

```

Рисунок 5.3 – Работа симулятора при сдвинутом периоде обновления памяти

На рисунке 5.3 видно, что АСА засек ошибки в памяти, т.к. эталонная и рабочая сигнатуры не совпали, и запустил маршевый тест для выявления неисправностей. Разрядка конденсаторов динамического ОЗУ является ошибкой, а не функциональной неисправностью, а потому маршевый тест ничего не обнаружил.

На рисунке 5.4 показана работа симулятора памяти при наличии неисправностей. В файле описания неисправностей записано две модели неисправностей: SAF и CFst. Ячейка-жертва находится по адресу 0x27, т.е. 0-ой

ранк, 0-ой банк, 0-ая строка, 2-ая колонка и 7-ой бит. Эта ячейка не может поменять значение логической единицы на логический ноль, если ячейка-агрессор, которая находится по адресу 0x42, содержит логический ноль.

```
==== MemorySystem 0 ====
== Loading device model file 'ini/DDR2_micron_1Mb.ini' ==
== Loading system model file 'system.ini' ==
TOTAL_STORAGE : 0.25MB | 1 Ranks |
[Testing System] Write data at address 0x2 on cycle 10
~ current clcclCycle: 10 ~
[DPKT] Rank receiving <-- : DATA: '128'
~ current clcclCycle: 14 ~
[DPKT] Rank returning --> : DATA: '128'
[Testing System] Read data [128] from adress 0x2
[Testing System] Write data at address 0x2 on cycle 22
~ current clcclCycle: 22 ~
[DPKT] Rank receiving <-- : DATA: '127'
~ current clcclCycle: 26 ~
[DPKT] Rank returning --> : DATA: '255'
[Testing System] Read data [255] from adress 0x2
I received a REFRESH command on cycle 2601
[Regeneration Controller] Detected error(s) in memory.
  Signature Sum = 78(r:0, b:0, row:0, col:2, bit:7)
[Regeneration Controller] March Test Started.
[MarchTest] Errors are detected while running phase!
  Signature sum is 78(r:0, b:0, row:0, col:2, bit:7)
[MarchTest] Errors are detected while running phase!
  Signature sum is 79(r:0, b:0, row:0, col:2, bit:7)
[MarchTest] Errors are detected while running phase!
  Signature sum is 78(r:0, b:0, row:0, col:2, bit:7)
[Regeneration Controller] March Test Failed.
=====
===== Printing Statistics [id:0]=====
Total Return Transactions : 4 (8 bytes) aggregate average bandwidth 0.001GB/s
```

Рисунок 5.4 – Работа симулятора при наличии функциональных неисправностей

На рисунке 5.4 отображено, что в начале в ячейку по адресу 0x2 (0-ой ранк, 0-ой банк, 0-ая строка, 2-ая колонка) было записано значение 128, что означает запись логической единицы в 7-ой бит слова. Операция чтения по этому же адресу подтверждает, что ячейка хранит это значение. Но при попытке записать значение 127 в ячейку памяти, что означает запись в биты с 0-го по 6-ой значения логической единицы, а в 7-ой бит логического нуля, в реальности сохраняется значение 255, т.е. 7-ой бит не смог поменять своё состояние из-за неисправности. АСА засек неполадку и запустил маршевый тест March C-, который в свою очередь обнаружил неисправность в памяти.

6 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПС

Целью дипломного проекта является создание программного средства для верификации алгоритмов тестирования оперативных запоминающих устройств. Данное программное средство позволяет облегчить верификацию ОЗУ, не прибегая к тестированию реальных микросистем. Основными достоинствами программного средства являются: симуляция реальных процессов ОЗУ, что позволяет обеспечить максимальную схожесть с реальными системами на чипе, существенное увеличение удобства процесса запуска и верификации тестов динамической оперативной памяти, актуальность.

В данном разделе рассмотрим экономическую эффективность программного средства. Программный комплекс относится ко 2-ой группе сложности. Категория новизны продукта - «В». Для оценки экономической эффективности разработанного программного средства проводится расчет цены и прибыли от продажи одной системы(программы).

Расчеты выполнены на основе методического пособия [20].

6.1 Расчёт сметы затрат и цены программного продукта

Целесообразность создания коммерческого ПО требует проведения предварительной экономической оценки и расчета экономического эффекта. Экономический эффект у разработчика ПО зависит от объёма инвестиций в разработку проекта, цены на готовый программный продукт и количества проданных копий, и проявляется в виде роста чистой прибыли.

Исходные данные для разрабатываемого проекта указаны в таблице 6.1. На основании сметы затрат и анализа рынка ПО определяется плановая отпускная цена. Расчет объёма программного продукта (количества строк исходного кода) предполагает определение типа программного обеспечения, всестороннее техническое обоснование функций ПО и определение объёма каждой функций. Согласно классификации типов программного обеспечения [20, с. 59, приложение 1], разрабатываемое ПО с наименьшей ошибкой можно классифицировать как ПО методо-ориентированных расчетов.

Общий объём программного продукта определяется исходя из коли-

Таблица 6.1 – Исходные данные

Наименование	Условное обозначение	Значение
Категория сложности		2
Коэффициент сложности, ед.	K_c	1,12
Степень использования при разработке стандартных модулей, ед.	K_T	0,7
Коэффициент новизны, ед.	K_n	0,7
Годовой эффективный фонд времени, дн.	$\Phi_{эф}$	231
Продолжительность рабочего дня, ч.	$T_{ч}$	8
Месячная тарифная ставка первого разряда, Br	$T_{м1}$	298 000
Коэффициент премирования, ед.	K	1,5
Норматив дополнительной заработной платы, ед.	H_d	20
Норматив отчислений в ФСЗН и обязательное страхование, %	$H_{сз}$	34,5
Норматив командировочных расходов, %	H_k	15
Норматив прочих затрат, %	$H_{пз}$	20
Норматив накладных расходов, %	$H_{рн}$	100
Прогнозируемый уровень рентабельности, %	$U_{рп}$	35
Норматив НДС, %	$H_{дс}$	20
Норматив налога на прибыль, %	$H_{п}$	18
Норматив расхода материалов, %	$H_{мз}$	3
Норматив расхода машинного времени, ч.	$H_{мв}$	15
Цена одного часа машинного времени, Br	$H_{мв}$	5000
Норматив расходов на сопровождение и адаптацию ПО, %	$H_{рса}$	30

чества и объёма функций, реализованных в программе:

$$V_o = \sum_{i=1}^n V_i, \quad (6.1)$$

где V_i —объём отдельной функции ПО, LoC;

n —общее число функций.

На стадии технико-экономического обоснования проекта рассчитать точный объём функций невозможно. Вместо вычисления точного объёма функций применяются приблизительные оценки на основе данных по ана-

логичным проектам или по нормативам [20, с. 61, приложение 2], которые приняты в организации.

Таблица 6.2 – Перечень и объём функций программного модуля

№ функции	Наименование (содержание)	Объём функции, LoC	
		по каталогу (V_i)	уточненный (V_i^y)
101	Организация ввода информации	150	70
102	Контроль, предварительная обработка и ввод информации	450	300
111	Управление вводом/выводом	2400	1000
301	Формирование последовательного файла	290	250
305	Обработка файлов	420	350
501	Монитор ПО (управление работой компонентов)	740	700
506	Обработка ошибочных и сбойных ситуаций	410	400
507	Обеспечение интерфейса между компонентами	970	680
701	Математическая статистика и прогнозирование	9320	2800
Итог		15 150	6550

Перечень и объём функций программного модуля перечислен в таблице 6.2. По приведенным данным уточненный объём некоторых функций изменился, и общий уточненный объём ПО $V_y = 6550$ LoC.

6.2 Расчёт нормативной трудоемкости

На основании общего объема ПО определяется нормативная трудоемкость (T_n) с учетом сложности ПО. Для ПО 2-ой группы сложности, к которой относится разрабатываемый программный продукт, нормативная трудоемкость составит $T_n = 172$ чел./дн.

Нормативная трудоемкость служит основой для оценки общей трудоемкости T_o . Используем формулу (6.2) для оценки общей трудоемкости для небольших проектов:

$$T_o = T_n \cdot K_c \cdot K_T \cdot K_n, \quad (6.2)$$

где K_c —коэффициент, учитывающий сложность ПО;

K_t —поправочный коэффициент, учитывающий степень использования при разработке стандартных модулей;

K_n —коэффициент, учитывающий степень новизны ПО.

Дополнительные затраты труда на разработку ПО учитываются через коэффициент сложности, который вычисляется по формуле

$$K_c = 1 + \sum_{i=1}^n K_i, \quad (6.3)$$

где K_i —коэффициент, соответствующий степени повышения сложности ПО за счет конкретной характеристики;

n —количество учитываемых характеристик.

Наличие двух характеристик сложности позволяет [20, с. 66, приложение 4, таблица П.4.2] вычислить коэффициент сложности

$$K_c = 1 + 0,12 = 1,12. \quad (6.4)$$

Разрабатываемое ПО использует стандартные компоненты. Согласно справочным данным [20, с. 68, приложение 4, таблица П.4.5] коэффициент использования стандартных модулей для разрабатываемого приложения $K_t = 0,7$. Разрабатываемое ПО не является новым, существуют аналогичные более зрелые разработки у различных компаний и университетов по всему миру. Влияние степени новизны на трудоемкость создания ПО определяется коэффициентом новизны — K_n . Согласно справочным данным [20, с. 67, приложение 4, таблица П.4.4] для разрабатываемого ПО $K_n = 0,7$. Подставив приведенные выше коэффициенты для разрабатываемого ПО в формулу (6.2) получим общую трудоемкость разработки

$$T_o = 172 \cdot 1,12 \cdot 0,7 \cdot 0,7 \approx 94 \text{ чел./дн.} \quad (6.5)$$

На основе общей трудоемкости и требуемых сроков реализации проекта вычисляется плановое количество исполнителей. Численность исполнителей проекта рассчитывается по формуле:

$$ч_p = \frac{T_o}{T_p \cdot \Phi_{эф}}, \quad (6.6)$$

где T_o —общая трудоемкость разработки проекта, чел./дн.;
 $\Phi_{эф}$ —эффективный фонд времени работы одного работника в течение года, дн.;
 T_p —срок разработки проекта, лет.

Эффективный фонд времени работы одного разработчика вычисляется по следующей формуле:

$$\Phi_{эф} = D_r - D_{п} - D_v - D_o, \quad (6.7)$$

где D_r —количество дней в году, дн.;
 $D_{п}$ —количество праздничных дней в году, не совпадающих с выходными днями, дн.;
 D_v —количество выходных дней в году, дн.;
 $D_{от}$ —количество дней отпуска, дн.

Согласно данным, приведенным в производственном календаре для пятидневной рабочей недели в 2016 году для Беларуси [21], фонд рабочего времени составит

$$\Phi_{эф} = 366 - 6 - 105 - 24 = 231 \text{ дн.} \quad (6.8)$$

Учитывая срок разработки проекта $T_p = 3 \text{ мес.} = 0,25 \text{ года}$, общую трудоемкость и фонд эффективного времени одного работника, вычисленные ранее, можем рассчитать численность исполнителей проекта

$$Ч_p = \frac{94}{0,25 \cdot 231} \approx 2 \text{ рабочих.} \quad (6.9)$$

Вычисленные оценки показывают, что для выполнения запланированного проекта в указанные сроки необходимо два рабочих.

6.3 Расчёт основной заработной платы исполнителей

Информация о работниках перечислена в таблице 6.3.

Таблица 6.3 – Работники, занятые в проекте

Исполнители	Разряд	Тарифный коэффициент	Чел./дн. занятости
Программист I-категории	13	3,04	47
Ведущий программист	15	3,48	47

Месячная тарифная ставка одного работника вычисляется по формуле

$$T_{\text{ч}} = \frac{T_{\text{м}_1} \cdot T_{\text{к}}}{\Phi_{\text{р}}}, \quad (6.10)$$

где $T_{\text{м}_1}$ —месячная тарифная ставка 1-го разряда, Br;

$T_{\text{к}}$ —тарифный коэффициент, соответствующий установленному тарифному разряду;

$\Phi_{\text{р}}$ —среднемесячная норма рабочего времени, час.

Подставив данные из таблицы 6.3 в формулу (6.10), приняв значение тарифной ставки 1-го разряда $T_{\text{м}_1} = 298\,000$ Br и среднемесячную норму рабочего времени $\Phi_{\text{р}} = 160$ часов получаем:

$$T_{\text{ч}}^{\text{прогр. I-разр.}} = \frac{298\,000 \cdot 3,04}{160} = 5662 \text{ Br/час}; \quad (6.11)$$

$$T_{\text{ч}}^{\text{вед. прогр.}} = \frac{298\,000 \cdot 3,48}{160} = 6482 \text{ Br/час}. \quad (6.12)$$

Основная заработная плата исполнителей на конкретное ПО рассчитывается по формуле:

$$Z_o = \sum_{i=1}^n T_{\text{ч}}^i \cdot T_{\text{ч}} \cdot \Phi_{\text{п}} \cdot K, \quad (6.13)$$

где $T_{\text{ч}}^i$ —часовая тарифная ставка i -го исполнителя, Br/час;

$T_{\text{ч}}$ —количество часов работы в день, час;

$\Phi_{\text{п}}$ —плановый фонд рабочего времени i -го исполнителя, дн.;

K —коэффициент премирования.

Подставив ранее вычисленные значения и данные из таблицы 6.3 в формулу (6.13) и приняв коэффициент премирования $K = 1,5$ получим

$$Z_o = (5662 \cdot 47 + 6482 \cdot 47) \cdot 8 \cdot 1,5 = 6\,849\,216 \text{ Br}. \quad (6.14)$$

Дополнительная заработная плата включает выплаты предусмотренные законодательством от труде и определяется по нормативу в процентах от основной заработной платы

$$Z_{\text{д}} = \frac{Z_o \cdot H_{\text{д}}}{100\%}, \quad (6.15)$$

где H_d —норматив дополнительной заработной платы, %.

Приняв норматив дополнительной заработной платы $H_d = 20\%$ и подставив известные данные в формулу (6.15) получим

$$Z_d = \frac{6\,849\,216 \cdot 20\%}{100\%} \approx 1\,369\,843 \text{ Br.} \quad (6.16)$$

Расчеты общей суммы расходов и прогнозируемой цены ПО, а также его себестоимости сведены в таблицу 6.4.

Таблица 6.4 – Расчет себестоимости и отпускной цены ПО

Наименование статей	Норматив, %	Методика расчета	Значение, руб.
Отчисления в фонд социальной защиты и обязательного страхования	$H_{сз} = 34,5$	$Z_{сз} = (Z_o + Z_d) \cdot H_{сз}/100$	2 835 575
Материалы и комплектующие	$H_{мз} = 3$	$M = Z_o \cdot H_{мз}/100$	205 476
Машинное время		$P_m = C_m \cdot V_o/100 \cdot H_{мв}$ $H_{мв} = 15$ машино-часов $C_m = 5000 \text{ Br}$	4 912 500
Расходы на научные командировки	$H_k = 15$	$P_k = Z_o \cdot H_k/100$	1 027 382
Прочие прямые расходы	$H_{пз} = 20$	$P_z = Z_o \cdot H_{пз}/100$	1 369 843
Накладные расходы	$H_{рн} = 100$	$P_n = Z_o \cdot H_{рн}/100$	6 849 216
Общая сумма расходов по смете		$C_p = Z_o + Z_d + Z_{сз} + M +$ $P_m + P_k + P_z + P_n$	25 419 051
Сопровождение и адаптация ПО	$H_{рса} = 30$	$P_{са} = C_p \cdot H_{рса}/100$	7 625 715
Полная себестоимость ПО		$C_{п} = C_p + P_{са}$	33 044 766

Продолжение таблицы 6.4

Наименование статей	Норматив, %	Методика расчета	Значение, руб.
Прогнозируемая прибыль	$Y_{rp} = 35$	$P_c = C_{п} \cdot Y_{rp} / 100$	11 565 668
Прогнозируемая цена без налогов		$C_{п} = C_{п} + P_c$	44 610 434
Отчисления и налоги в местный и республиканский бюджеты	$H_{mp} = 3,9$	$O_{mp} = C_{п} \cdot H_{mp} / 100 - H_{mp}$	1 810 413
Налог на добавленную стоимость	$H_{dc} = 20$	$HDC = (C_{п} + O_{mp}) \cdot H_{dc} / 100$	9 284 169
Прогнозируемая отпускная цена		$C_o = C_{п} + O_{mp} + HDC$	55 705 016

6.4 Расчёт экономической эффективности у разработчика

Важной задачей является расчет экономической эффективности проектов и выбор наиболее выгодного проекта. Разрабатываемое ПО является заказным, т.е. разрабатывается для одного заказчика на заказ. На основании анализа рыночных условий и договоренности с заказчиком об отпускной цене прогнозируемая рентабельность проекта составит $Y_{rp} = 35\%$.

Чистую прибыль от реализации проекта рассчитаем по формуле

$$P_{ч} = P_c \cdot \left(1 - \frac{H_{п}}{100\%} \right), \quad (6.17)$$

где $H_{п}$ — величина налога на прибыль, %.

Приняв значение налога на прибыль $H_{п} = 18\%$ и подставив известные данные в формулу (6.17) получаем чистую прибыль:

$$P_{ч} = 11\,565\,668 \cdot \left(1 - \frac{18\%}{100\%} \right) = 9\,483\,848 \text{ Br.} \quad (6.18)$$

Программное обеспечение разрабатывалось для одного заказчика в связи с этим экономическим эффектом разработчика будет являться чистая прибыль от реализации $P_{\text{ч}}$. Рассчитанные данные приведены в таблице 6.5.

Таблица 6.5 – Рассчитанные данные

Наименование	Условное обозначение	Значение
Нормативная трудоемкость, чел./дн.	$T_{\text{н}}$	172
Общая трудоемкость разработки, чел./дн.	$T_{\text{о}}$	94
Численность исполнителей, чел.	$Ч_{\text{р}}$	2
Часовая тарифная ставка программиста I-разряда, Br/ч.	$T_{\text{ч}}^{\text{прогр. I-разр.}}$	5662
Часовая тарифная ставка ведущего программиста, Br/ч.	$T_{\text{ч}}^{\text{вед. прогр.}}$	6482
Основная заработная плата, Br	$З_{\text{о}}$	6 849 216
Дополнительная заработная плата, Br	$З_{\text{д}}$	1 369 843
Отчисления в фонд социальной защиты, Br	$З_{\text{сз}}$	2 835 575
Затраты на материалы, Br	M	205 476
Расходы на машинное время, Br	$P_{\text{м}}$	4 912 500
Расходы на командировки, Br	$P_{\text{к}}$	1 027 382
Прочие затраты, Br	$P_{\text{з}}$	1 369 843
Накладные расходы, Br	$P_{\text{н}}$	6 849 216
Общая сумма расходов по смете, Br	$C_{\text{р}}$	25 419 051
Расходы на сопровождение и адаптацию, Br	$P_{\text{са}}$	7 625 715
Полная себестоимость, Br	$C_{\text{п}}$	33 044 766
Прогнозируемая прибыль, Br	$P_{\text{с}}$	11 565 668
НДС, Br	НДС	9 284 169
Прогнозируемая отпускная цена ПО, Br	$Ц_{\text{о}}$	55 705 016
Чистая прибыль, Br	$P_{\text{ч}}$	9 483 848

6.5 Выводы по технико-экономическому обоснованию

Разрабатываемое ПС является выгодным программным продуктом. Чистая прибыль от реализации ПС ($P_{\text{ч}}$ 9 483 848 рублей) остается организации-разработчику и представляет собой экономический эффект от создания нового программного средства. Таким образом, данная разработка является экономически целесообразной.

ЗАКЛЮЧЕНИЕ

В данном дипломном проекте был рассмотрен вопрос верификации алгоритмов тестирования оперативных запоминающих устройств. В рамках дипломного проекта была разработана библиотека для симуляции работы динамического оперативного запоминающего устройства. В созданной библиотеке реализованы несколько моделей функциональных неисправностей ОЗУ, а также создан механизм для верифицирования тестирующих алгоритмов ОЗУ. Для обеспечения регенерации памяти были реализованы алгоритмы встроенного самотестирования ОЗУ, основанные на адаптивном сигнатурном анализаторе. Так же был смоделирован процесс разрядки конденсаторов запоминающих элементов ОЗУ.

В целом были получены удовлетворительные результаты на хорошо изученных и известных алгоритмах маршевых тестов, выявлена покрывающая способность тестов на наиболее распространенных моделях функциональных неисправностей. В данном программном средстве удалось создать универсальный механизм для верификации любых маршевых тестов.

В итоге получилось раскрыть тему дипломного проекта и создать в его рамках программное средство.

Но за рамками рассматриваемой темы осталось еще много других алгоритмов тестирования, а также моделей неисправностей ОЗУ, которые в будущем можно будет внедрить в уже готовое программное средство.

В дальнейшем планируется развивать и улучшать существующее программное обеспечение до полноценной библиотеки, способной эффективно решать достаточно трудоёмкие задачи по моделированию многосвязных неисправностей и верификации тестирующих алгоритмов, что должно повлиять на качество новых разрабатываемых тестов и улучшить надежность современных цифровых устройств.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] B. Jacob, D. Wang. DRAM Tutorial / D. Wang B. Jacob // DRAM: Architectures, Interfaces, and Systems. — 2002.
- [2] Dynamic RAM, DRAM Memory Technology Tutorial [Электронный ресурс]. — Электронные данные. — Режим доступа: <http://www.radio-electronics.com/info/data/semicond/memory/dram-technology-basics-tutorial.php>. — Дата доступа: 24.02.2016.
- [3] DRAM – Dynamic Random Access Memory [Электронный ресурс]. — Электронные данные. — Режим доступа: <https://www.pctechguide.com/computer-memory/dram-dynamic-random-access-memory>. — Дата доступа: 03.04.2016.
- [4] A. S. Tanenbaum, T. Austin. Structured Computer Organization - 6th ed. / T. Austin A. S. Tanenbaum. — Vrije Universiteit Amsterdam, The Netherlands, 2013. — December. — 747 p.
- [5] Group, IBM. Understanding DRAM Operation / IBM Group. — 1996.
- [6] С. В. Ярмолик А. П. Занкович, А. А. Иванюк. Маршевые тесты для самотестирования ОЗУ / А. А. Иванюк С. В. Ярмолик, А. П. Занкович. — Издательский центр БГУ, 2009. — 271 с.
- [7] А. А. Иванюк, Д. С. Петроненко. Современные неразрушающие методы и алгоритмы диагностирования оперативных запоминающих устройств. / Д. С. Петроненко А. А. Иванюк. — 2004. — July.
- [8] C++ Tutorial [Электронный ресурс]. — Электронные данные. — Режим доступа: <http://www.cplusplus.com/doc/tutorial/>. — Дата доступа: 07.04.2016.
- [9] Страуструп, Б. Язык программирования C++ - 3-е изд. / Б. Страуструп. — Невский диалект - Бином, 1999. — 991 с.
- [10] Wikipedia [Электронный ресурс]. — Электронные данные. — Режим доступа: <https://ru.wikipedia.org/wiki>. — Дата доступа: 23.04.2016.
- [11] Прата, С. Язык программирования C++ (C++11). Лекции и упражнения - 6-е изд. / С. Прата. — 2011. — 688 с.
- [12] Z. Xie Y. Zhang, J. Yang L. Shi. An improved memory system simulator based on DRAMSim2 / J. Yang L. Shi Z. Xie, Y. Zhang. — 2014. — July.
- [13] P. Rosenfeld E. Cooper-Balis, B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator / B. Jacob P. Rosenfeld, E. Cooper-Balis. — 2011.

- [14] Stackoverflow [Электронный ресурс]. — Электронные данные. — Режим доступа: <http://stackoverflow.com>. — Дата доступа: 02.05.2016.
- [15] Y. Kim W. Yang, O. Mutlu. Ramulator: A Fast and Extensible DRAM Simulator / O. Mutlu Y. Kim, W. Yang. — 2015.
- [16] Jeong, M. K. DrSim: A Platform for Flexible DRAM System Research / M. K. Jeong. — 2012.
- [17] N. Chatterjee R. Balasubramonian, M. Shevgoor S. H. Pugsley Aniruddha N. Udipi A. Shafiee K. Sudan M. Awasthi Z. Chishti. USIMM: the Utah SIMulated Memory Module / M. Shevgoor S. H. Pugsley Aniruddha N. Udipi A. Shafiee K. Sudan M. Awasthi Z. Chishti N. Chatterjee, R. Balasubramonian. — 2012. — February.
- [18] S. Hellebrand H.-J. Wunderlich, A. A. Ivaniuk Y. V. Klimets V. N. Yarmolik. Efficient Online and Offline Testing of Embedded DRAMs / A. A. Ivaniuk Y. V. Klimets V. N. Yarmolik S. Hellebrand, H.-J. Wunderlich. — 2002. — July.
- [19] V. N. Yarmolik S. Hellebrand, H.-J. Wunderlich. Self-Adjusting Output Data Compression: An Efficient BIST Technique for RAMs / H.-J. Wunderlich V. N. Yarmolik, S. Hellebrand. — 1998. — February.
- [20] Палицын, В. А. Техничко-экономическое обоснование дипломных проектов: Метод. пособие для студ. всех спец. БГУИР. В 4-х ч. Ч. 4: Проекты программного обеспечения / В. А. Палицын. — Минск : БГУИР, 2006. — 76 с.
- [21] Календарь праздников на 2016 год для Беларуси [Электронный ресурс]. — Электронные данные. — Режим доступа: <http://calendar.by/2016/#bkm>. — Дата доступа: 05.03.2016.

ПРИЛОЖЕНИЕ А

(обязательное)

Исходный код программного средства

```
#pragma once
#include "SimulatorObject.h"
#include "BusPacket.h"
#include "Rank.h"
#include "Discharger.h"
#include "FaultController.h"
#include "Address.h"

using namespace DRAMSim;

namespace DRAMSim
{
    class MemoryController;
    class DRAMDevice : public SimulatorObject
    {
        friend FaultController;
    public:
        DRAMDevice(MemoryController *mc, std::string faultFilePath = "");
        ~DRAMDevice();

    public:
        void receiveFromBus(BusPacket *packet);
        void attachRanks(vector<Rank> *ranks);
        void update() override;
        void step() override;

        void setRefreshWaitingFlag(int rank);
        bool getRefreshWaitingFlag(int rank);
        void powerDown(int rank);
        void powerUp(int rank);

        void dischargeCells(int cycleCount, int rank);

    public:
        uint16_t read(Address addr);
        void write(Address addr, uint16_t data);
        void invertBit(Address addr);

    private:
        uint16_t readBit(Address addr);

    private:
        vector<Rank> *ranks;
        Discharger discharger;
        FaultController faultController;
    };
}
```

```

#include "DRAMDevice.h"
#include "MemoryController.h"
#include "SystemConfiguration.h"
#include "Parsing.h"
#include <vector>

using namespace std;
using namespace DRAMSim;

DRAMDevice::DRAMDevice(MemoryController *mc, string faultFilePath)
: faultController(this)
{
    currentClockCycle = 0;
    ranks = new vector<Rank>();

    for (size_t i = 0; i<NUM_RANKS; i++)
    {
        Rank r = Rank();
        r.setId(i);
        r.attachMemoryController(mc);
        ranks->push_back(r);
    }
    discharger.Initialize();
    if (!faultFilePath.empty())
    {
        vector<Fault> faults = ParseCSV(faultFilePath);
        faultController.SetFaults(faults);
    }
}

DRAMDevice::~~DRAMDevice()
{
    ranks->clear();
    delete(ranks);
}

void DRAMDevice::attachRanks(vector<Rank> *ranks)
{
    this->ranks = ranks;
}

void DRAMDevice::update()
{
    //updates the state of each of the objects
    // NOTE - do not change order
    for (size_t i = 0; i<NUM_RANKS; i++)
    {
        (*ranks)[i].update();
    }
}

void DRAMDevice::step()

```

```

{
    for (size_t i = 0; i<NUM_RANKS; i++)
    {
        (*ranks)[i].step();
    }
    SimulatorObject::step();
}

void DRAMDevice::receiveFromBus(BusPacket *packet)
{
    if (packet->busPacketType == DATA)
    {
        faultController.DoFaults(packet->address, packet->data->getData())
        ;
        (*ranks)[packet->address.rank].receiveFromBus(packet);
    }
    else
    {
        (*ranks)[packet->address.rank].receiveFromBus(packet);
    }
}

uint16_t DRAMDevice::read(Address addr)
{
    return (*ranks)[addr.rank].banks[addr.bank].read(addr);
}

uint16_t DRAMDevice::readBit(Address addr)
{
    return (read(addr) & (1 << addr.bit)) ? 1 : 0;
}

void DRAMDevice::write(Address addr, uint16_t data)
{
    faultController.DoFaults(addr, data);
}

void DRAMDevice::invertBit(Address addr)
{
    uint16_t oldBitValue = readBit(addr);
    uint16_t newBitValue = oldBitValue ? 0 : 1;
    faultController.DoOperationOnBit(addr, newBitValue, oldBitValue);
}

void DRAMDevice::setRefreshWaitingFlag(int rank)
{
    (*ranks)[rank].refreshWaiting = true;
}

bool DRAMDevice::getRefreshWaitingFlag(int rank)
{
    return (*ranks)[rank].refreshWaiting;
}

```

```

void DRAMDevice::powerDown(int rank)
{
    (*ranks)[rank].powerDown();
}

void DRAMDevice::powerUp(int rank)
{
    (*ranks)[rank].powerUp();
}

void DRAMDevice::dischargeCells(int cycleCount, int rank)
{
    std::vector<int> addrList = discharger.GetRandomAddressList(cycleCount);
    (*ranks)[rank].dischargeCells(addrList);
}

#pragma once
#include "BusPacket.h"
#include "DRAMDevice.h"
#include "Address.h"
#include "Rank.h"

class SAODCCController
{
public:
    SAODCCController();
    ~SAODCCController();

public:
    void SetDRAMDevice(DRAMDevice *dram);
    void UpdateRef(DRAMSim::BusPacket* packet);

    void UpdateRef(Address addr, uint16_t data);
    void UpdateTestSig(Address addr, uint16_t data);

    int GetSignaturesSum();

    bool ParityBitInZero(int signature);
    int CalculateTestAndCompare();

    void ClearTestSig();
    void SetRefSignature(int signature);
    int GetTestSignature();

private:
    // function calculates address sum of each changed bit in word
    int GetAddress(Address addr, uint16_t data);
    void AddParityBit(int &address);

private:
    int RefSignature;
    int TestSignature;

```

```

        DRAMDevice *dramDevice;
};

#include "SystemConfiguration.h"
#include "SAODCController.h"

using namespace DRAMSim;

SAODCController::SAODCController()
{
    RefSignature = TestSignature = 0;
    dramDevice = 0;
}

SAODCController::~SAODCController()
{
}

void SAODCController::SetDRAMDevice(DRAMDevice *dram)
{
    dramDevice = dram;
}

void SAODCController::UpdateRef(BusPacket* packet)
{
    if (packet->busPacketType == DATA)
    {
        uint16_t oldData = dramDevice->read(packet->address);
        uint16_t newData = packet->data->getData();
        if (oldData != newData)
            UpdateRef(packet->address, newData ^ oldData);
    }
}

void SAODCController::UpdateRef(Address addr, uint16_t data)
{
    RefSignature ^= GetAddress(addr, data);
}

void SAODCController::UpdateTestSig(Address addr, uint16_t data)
{
    TestSignature ^= GetAddress(addr, data);
}

int SAODCController::GetSignaturesSum()
{
    return TestSignature ^ RefSignature;
}

int SAODCController::GetAddress(Address addr, uint16_t data)
{
    //get 'bit address sum'

```

```

        if (!data) return 0;

        int bitSum = 0;
        int bitCount = 0; // '1' bit count
        for (int i = 0; (unsigned)i < DEVICE_WIDTH; i++)
        {
            if (data & (1 << i))
            {
                bitSum ^= i;
                bitCount++;
            }
        }

        if (bitCount % 2 == 0)
            addr.Clear();

        addr.bit = bitSum;
        int address = addr.GetPhysical();

        if (bitCount % 2 != 0)
            AddParityBit(address);

        return address;
    }

    int SAODCCController::CalculateTestAndCompare()
    {
        TestSignature = 0;
        for (int r = 0; r < NUM_RANKS; r++)
        {
            for (int b = 0; b < NUM_BANKS; b++)
            {
                for (int row = 0; row < NUM_ROWS; row++)
                {
                    for (int col = 0; col < NUM_COLS; col++)
                    {
                        Address addr(r, b, row, col);
                        uint16_t data = dramDevice->read(addr);
                        UpdateTestSig(addr, data);
                    }
                }
            }
        }

        return GetSignaturesSum();
    }

    void SAODCCController::ClearTestSig()
    {
        TestSignature = 0;
    }

    void SAODCCController::SetRefSignature(int signature)

```



```

{
    RefSignature = signature;
}

int SAODCController::GetTestSignature()
{
    return TestSignature;
}

void SAODCController::AddParityBit(int &address)
{
    address <<= 1;
    address |= 1;
}

bool SAODCController::ParityBitInZero(int signature)
{
    return (signature & 1) == 0;
}

#pragma once
#include "SimulatorObject.h"
#include "SAODCController.h"
#include "MarchTestController.h"
#include "BusPacket.h"
#include "AddressTranslator.h"
#include "DRAMDevice.h"
#include "Rank.h"

class RegenerationController : public DRAMSim::SimulatorObject
{
public:
    RegenerationController();
    ~RegenerationController();

public:
    void Initialize(DRAMDevice* dram);
    void UpdateRefSignature(DRAMSim::BusPacket* packet);
    void SetMarchTest(string marchTest);
    void StartRefresh();
    void update();

private:
    int GetMarchTestType(string marchTest);

private:
    MarchTestController marchController;
    SAODCController saodcController;
    AddressTranslator addrTranslator;

    DRAMDevice* dramDevice;

    int lastError;

```

```

    int refreshEndCycle;
    int startMarchTestCycle;

    bool needTest;

};

#include "RegenerationController.h"
#include "SystemConfiguration.h"
#include "MarchDef.h"
#include <iostream>

RegenerationController::RegenerationController()
{
    currentClockCycle = 0;
    lastError = 0;
    dramDevice = 0;
}

RegenerationController::~RegenerationController()
{
}

void RegenerationController::Initialize(DRAMDevice* dram)
{
    dramDevice = dram;
    refreshEndCycle = 0;
    startMarchTestCycle = 0;
    saodcController.SetDRAMDevice(dram);
    marchController.Initialize(dram);
    needTest = false;
}

void RegenerationController::SetMarchTest(string marchTest)
{
    marchController.SetMarchTest(GetMarchTestType(marchTest));
}

int RegenerationController::GetMarchTestType(string marchTest)
{
    if (marchTest.compare("March C-") == 0)
        return MARCH_C_MINUS;
    if (marchTest.compare("March A") == 0)
        return MARCH_A;
    if (marchTest.compare("March B") == 0)
        return MARCH_B;
    if (marchTest.compare("March X") == 0)
        return MARCH_X;
    if (marchTest.compare("March Y") == 0)
        return MARCH_Y;
    if (marchTest.compare("MATS") == 0)
        return MATS;
}

```

```

        if (marchTest.compare("MATS+") == 0)
            return MATS_PLUS;
        if (marchTest.compare("MATS++") == 0)
            return MATS_PLUS_PLUS;
        return MATS;
    }

void RegenerationController::StartRefresh()
{
    refreshEndCycle = currentClockCycle + tRFC - 1;
    int err = saodcController.CalculateTestAndCompare();
    if (err)
    {
        std::cout << "[Regeneration Controller] Detected error(s) in memory.\n"
            "Signature Sum = " << err << "(" << addrTranslator.GetDescription(err
            >>1, true) << ")" << std::endl;
        if (lastError)
        {
            needTest = true;
            startMarchTestCycle = currentClockCycle + 1;
        }
        else
        {
            if (saodcController.ParityBitInZero(err))
            {
                //multiple errors
                needTest = true;
                startMarchTestCycle = currentClockCycle + 1;
            }
            else
            {
                //if it first time - let's decide that it was 1 error.
                Invert bit
                lastError = err;
                std::cout << "[Regeneration Controller] Try to restore
                    the damaged cell." << std::endl;
                Address addr(err >> 1, true);
                dramDevice->invertBit(addr);
            }
        }
    }
    else
    {
        lastError = 0;
    }
}

void RegenerationController::update()
{
    if (currentClockCycle >= refreshEndCycle)
        return;

    if (needTest)

```

```

{
    if (startMarchTestCycle == currentClockCycle)
    {
        std::cout << "[Regeneration Controller] March Test Started." << std::
            endl;
        marchController.RunTest(saodcController.GetTestSignature());
    }
    else if (startMarchTestCycle < currentClockCycle)
    {
        marchController.Update();
    }

    if (marchController.TestCompleted())
    {
        if (!marchController.TestPassed())
        {
            std::cout << "[Regeneration Controller] March Test Failed." << std
                ::endl;
            //do smth in case of errors
        }
    }
    else
    {
        std::cout << "[Regeneration Controller] March Test Passed." << std
            ::endl;
    }

    marchController.Reset();
    needTest = false;
}
}

void RegenerationController::UpdateRefSignature(DRAMSim::BusPacket* packet)
{
    saodcController.UpdateRef(packet);
}

#pragma once

#include <vector>
#include "DRAMDevice.h"
#include "AddressTranslator.h"
#include "SAODCController.h"
#include "MarchDef.h"
#include <vector>

class MarchTestController
{
    struct State
    {
        bool testStarted;
        bool testCompleted;
        int phase;
        bool testPassed;
    }
};

```

```

        State() { Clear(); }

        void Clear()
        {
            testStarted = false;
            testCompleted = false;
            phase = 0;
            testPassed = false;
        }
    };

    struct MarchPhase
    {
        MarchDirection direction;
        std::vector<MarchOperation> elements;
    };

public:
    MarchTestController();
    ~MarchTestController();

public:
    void Initialize(DRAMDevice* dram);
    void SetMarchTest(int marchTest);

    void RunTest(int refSignature);
    void Update();
    bool TestCompleted();
    bool TestPassed();
    void Reset();

private:
    void RunElement(int element, int address, uint16_t &buffer, uint16_t&
        oldValue);

private:
    AddressTranslator addrTranslator;
    SAODCCController saodc;
    State state;
    std::vector<MarchPhase> phases;
    DRAMDevice* dramDevice;
    bool testPass;
};

#include "MarchTestController.h"
#include "SystemConfiguration.h"
#include <iostream>

MarchTestController::MarchTestController()
{
    dramDevice = 0;
}

```

```

MarchTestController::~MarchTestController()
{}

void MarchTestController::Initialize(DRAMDevice* dram)
{
    dramDevice = dram;
}

void MarchTestController::SetMarchTest(int marchTest)
{
    phases.clear();
    switch (marchTest)
    {
        case MARCH_C_MINUS:
        {
            phases.push_back({ MD_UP, { MO_RD, MO_WDC } });
            phases.push_back({ MD_UP, { MO_RDC, MO_WD } });
            phases.push_back({ MD_DOWN, { MO_RD, MO_WDC } });
            ;
            phases.push_back({ MD_DOWN, { MO_RDC, MO_WD } });
            ;
            phases.push_back({ MD_BOTH, { MO_RD } });
            break;
        }
        case MARCH_B:
        {
            phases.push_back({ MD_UP, { MO_RD, MO_WDC,
                MO_RDC, MO_WD, MO_RD, MO_WDC } });
            phases.push_back({ MD_UP, { MO_RDC, MO_WD,
                MO_WDC } });
            phases.push_back({ MD_DOWN, { MO_RDC, MO_WD,
                MO_WDC, MO_WD } });
            phases.push_back({ MD_DOWN, { MO_RD, MO_WDC,
                MO_WD } });
            break;
        }
        case MARCH_A:
        {
            phases.push_back({ MD_UP, { MO_RD, MO_WDC, MO_WD
                , MO_WDC } });
            phases.push_back({ MD_UP, { MO_RDC, MO_WD,
                MO_WDC } });
            phases.push_back({ MD_DOWN, { MO_RDC, MO_WD,
                MO_WDC, MO_WD } });
            phases.push_back({ MD_DOWN, { MO_RD, MO_WDC,
                MO_WD } });
            break;
        }
        case MARCH_X:
        {
            phases.push_back({ MD_UP, { MO_RD, MO_WDC } });
            phases.push_back({ MD_DOWN, { MO_RDC, MO_WD } })

```

```

        ;
        phases.push_back({ MD_BOTH, { MO_RD } });
        break;
    }
    case MARCH_Y:
    {
        phases.push_back({ MD_UP, { MO_RD, MO_WDC,
            MO_RDC } });
        phases.push_back({ MD_DOWN, { MO_RDC, MO_WD,
            MO_RD } });
        phases.push_back({ MD_BOTH, { MO_RD } });
        break;
    }
    case MATS:
    {
        phases.push_back({ MD_UP, { MO_RD, MO_WDC } });
        phases.push_back({ MD_DOWN, { MO_RDC } });
        break;
    }
    case MATS_PLUS:
    {
        phases.push_back({ MD_UP, { MO_RD, MO_WDC } });
        phases.push_back({ MD_DOWN, { MO_RDC, MO_WD } });
        ;
        break;
    }
    case MATS_PLUS_PLUS:
    {
        phases.push_back({ MD_UP, { MO_RD, MO_WDC } });
        phases.push_back({ MD_DOWN, { MO_RDC, MO_WD,
            MO_RD } });
        break;
    }
    default:
        break;
    }
}

void MarchTestController::RunTest(int refSignature)
{
    Reset();
    state.testStarted = true;
    saodc.SetRefSignature(refSignature);
}

void MarchTestController::Update()
{
    if (!state.testStarted)
        return;

    saodc.ClearTestSig();

    MarchPhase& phase = phases[state.phase];

```

```

int addrStart = 0;
int counter = 0;

int bottom = 0;
int top = NUM_BANKS * NUM_COLS * NUM_ROWS - 1;

if (phase.direction == MD_UP || phase.direction == MD_BOTH)
{
    addrStart = bottom;
    counter = 1;
}
else if (phase.direction == MD_DOWN)
{
    addrStart = top;
    counter = -1;
}

for (int addr = addrStart; addr >= bottom && addr <= top; addr +=
counter)
{
    uint16_t buffer = 0;
    uint16_t old = 0;
    for (auto& el : phase.elements)
    {
        RunElement(el, addr, buffer, old);
    }
}

int err = saodc.GetSignaturesSum();
if (err)
{
    state.testPassed = false;
    cout << "[MarchTest] Errors are detected while running phase!\n
Signature sum is " << err << "(" << addrTranslator.GetDescription(
err>>1, true) << ")" << endl;
}

state.phase++;
if (state.phase == phases.size())
{
    state.testCompleted = true;
    state.testStarted = false;
}
}

void MarchTestController::RunElement(int element, int address, uint16_t &
buffer, uint16_t& oldValue)
{
    Address addr(address, false);
    uint16_t data = 0;

    switch (element)

```



```

    {
    case MO_RD:
        data = dramDevice->read(addr);
        buffer = data;
        break;
    case MO_RDC:
        data = dramDevice->read(addr);
        buffer = ~data;
        break;
    case MO_WD:
        dramDevice->write(addr, buffer);
        break;
    case MO_WDC:
        dramDevice->write(addr, ~buffer);
        break;
    }

    //convertData
    if (element == MO_RD || element == MO_RDC)
    {
        saodc.UpdateTestSig(addr, data ^ oldValue);
        oldValue = data;
    }
}

bool MarchTestController::TestCompleted()
{
    return state.testCompleted;
}

bool MarchTestController::TestPassed()
{
    return state.testPassed;
}

void MarchTestController::Reset()
{
    state.testStarted = false;
    state.testPassed = true;
    state.testCompleted = false;
    state.phase = 0;
    saodc.ClearTestSig();
}

#pragma once

#include <vector>

enum MarchDirection
{
    MD_UP,
    MD_DOWN,
    MD_BOTH

```

```

};

enum MarchOperation
{
    MO_RD,
    MO_RDC,
    MO_WD,
    MO_WDC
};

enum MarchTest
{
    MARCH_C_MINUS,
    MARCH_B,
    MARCH_A,
    MARCH_Y,
    MARCH_X,
    MATS,
    MATS_PLUS,
    MATS_PLUS_PLUS
};

#pragma once

enum FaultType
{
    NONE = 0,
    SAF,
    TF,
    CFin,
    CFid,
    CFst,
    AF
};

struct Fault
{
    FaultType type;
    int victimValue;
    int agressorValue;
    int agressorAddress;
    int victimAddress;
};

#pragma once
#include "FaultDef.h"
#include "Address.h"
#include <stdint.h>
#include <vector>

namespace DRAMSim
{
    class DRAMDevice;

```

```

}

class FaultController
{

public:
    FaultController(DRAMSim::DRAMDevice* dram);
    ~FaultController();

public:
    void SetFaults(std::vector<Fault>& faults);
    bool IsFaulty(int address);
    bool IsAgressor(int address);
    void DoFaults(Address addr, uint16_t data);
    void DoOperationOnBit(Address addr, uint16_t newVal, uint16_t oldVal);

    Fault GetCellAttributes(int address, bool isAgressor);

private:
    void WriteBit(Address address, int val);
    void InitSAFs();

private:
    DRAMSim::DRAMDevice* dramDevice;
    AddressTranslator translator;
    std::vector<Fault> faultyCells;
};

#include "FaultController.h"
#include "DRAMDevice.h"
#include <algorithm>
using namespace std;

FaultController::FaultController(DRAMDevice* dram) : dramDevice(dram)
{
}

FaultController::~~FaultController()
{
}

void FaultController::SetFaults(std::vector<Fault>& faults)
{
    faultyCells = faults;
    auto it = find_if(faultyCells.begin(), faultyCells.end(), [](Fault f){
        return f.type == SAF; });
    if (it != faultyCells.end())
    {
        InitSAFs();
    }
}

void FaultController::InitSAFs()

```

```

{
    for (auto& f : faultyCells)
    {
        if (f.type == SAF)
        {
            Address addr(f.victimAddress, true);
            (*dramDevice->ranks)[addr.rank].banks[addr.bank].writeBit(
                addr, f.victimValue == 1);
        }
    }
}

bool FaultController::IsFaulty(int address)
{
    auto it = find_if(faultyCells.begin(), faultyCells.end(),
        [address](Fault f){return f.victimAddress == address && f.type !=
            AF; });
    return it != faultyCells.end();
}

bool FaultController::IsAgressor(int address)
{
    auto it = find_if(faultyCells.begin(), faultyCells.end(),
        [address](Fault f){return f.agressorAddress == address; });
    return it != faultyCells.end();
}

void FaultController::DoFaults(Address addr, uint16_t data)
{
    uint16_t oldData = dramDevice->read(addr);
    uint16_t bitMask = oldData ^ data;

    for (unsigned i = 0; i < DEVICE_WIDTH; i++)
    {
        if (bitMask & (1 << i))
        {
            addr.bit = i;
            DoOperationOnBit(addr, data & (1 << i) ? 1 : 0, oldData & (1
                << i) ? 1 : 0);
        }
    }
}

void FaultController::DoOperationOnBit(Address address, uint16_t newVal,
    uint16_t oldVal)
{
    int addr = address.GetPhysical();
    if (IsFaulty(addr))
    {
        Fault fault = GetCellAttributes(addr, false);
        switch (fault.type)
        {
            case SAF: break;
        }
    }
}

```

```

    case TF:
    {
        if (oldVal != fault.victimValue)
        {
            WriteBit(address, newVal);
        }
        break;
    }
    case CFin:
    {
        WriteBit(address, newVal);
        break;
    }
    case CFid:
    {
        WriteBit(address, newVal);
        break;
    }
    case CFst:
    {
        if (newVal != fault.victimValue)
            WriteBit(address, newVal);
        else
        {
            Address agrAdr(fault.agressorAddress,
                true);
            if (dramDevice->readBit(agrAdr) == fault.
                agressorValue)
                WriteBit(address, newVal);
        }
        break;
    }
    default: break;
}
else
{
    if (IsAgressor(addr))
    {
        Fault fault = GetCellAttributes(addr, true);
        switch (fault.type)
        {
            case CFin:
            {
                if (newVal == fault.agressorValue &&
                    newVal != oldVal)
                {
                    Address vicAdr(fault.victimAddress,
                        true);
                    (*dramDevice->ranks)[vicAdr.rank].
                        banks[vicAdr.bank].invertBit(
                            vicAdr);
                }
            }
        }
    }
}

```

```

        break;
    }
    case CFid:
    {
        if (newVal == fault.agressorValue &&
            newVal != oldVal)
        {
            Address vicAdr(fault.victimAddress,
                           true);
            WriteBit(vicAdr, fault.victimValue)
                ;
        }
        break;
    }
    case AF:
    {
        Address vicAdr(fault.victimAddress, true);
        WriteBit(vicAdr, newVal);
        break;
    }
    default :
        break;
    }
    WriteBit(address, newVal);
}
else
{
    WriteBit(address, newVal);
}
}

Fault FaultController::GetCellAttributes(int address, bool isAgressor)
{
    vector<Fault>::iterator it;
    if (isAgressor)
    {
        it = find_if(faultyCells.begin(), faultyCells.end(), [address](
            Fault f){return f.agressorAddress == address; });
    }
    else
    {
        it = find_if(faultyCells.begin(), faultyCells.end(), [address](
            Fault f){return f.victimAddress == address && f.type != AF; })
            ;
    }
    if (it != faultyCells.end())
        return *it;
    else return Fault();
}

void FaultController::WriteBit(Address address, int val)
{

```

```

        (*dramDevice->ranks)[address.rank].banks[address.bank].writeBit(address,
            val == 1);
    }

#pragma once
#include <random>
#include <vector>

using namespace std;
class default_random_engine;
class exponential_distribution;

class Discharger
{
public:
    Discharger();
    ~Discharger();

public:
    void Initialize();
    vector<int> GetRandomAddressList(int curCycle);

private:
    std::random_device rd;
    std::mt19937* gen;
    std::uniform_int_distribution<>* uniformDist;
};

#include "Discharger.h"
#include "SystemConfiguration.h"
#include <math.h>

using namespace std;

Discharger::Discharger() : gen(0), uniformDist(0)
{

}

Discharger::~Discharger()
{
    delete uniformDist;
}

void Discharger::Initialize()
{
    gen = new std::mt19937(rd());
    int totalBits = NUM_RANKS * NUM_BANKS * NUM_ROWS * NUM_COLS * DEVICE_WIDTH;
    uniformDist = new std::uniform_int_distribution<>(0, totalBits - 1);
}

vector<int> Discharger::GetRandomAddressList(int curCycle)
{

```

```

vector<int> addrList;
double number = exp(sqrt(curCycle)/5.2);
int addrCount = (int)number;

for (int i = 0; i < addrCount; i++)
{
    addrList.push_back((*uniformDist)(*gen));
}
return addrList;
}

#pragma once
#include <string>

class AddressTranslator
{
public:
    AddressTranslator();
    ~AddressTranslator();

    void Init();

    void Translate(int address, int& rank, int& bank, int& row, int& col);
    void Translate(int address, int& rank, int& bank, int& row, int& col, int&
        bit);

    int TranslateToAddr(int rank, int bank, int row, int col, int bit = -1);

    std::string GetDescription(int address, bool includeBit);

private:
    int GetValue(int width, int& address);
    void SetValue(int width, int value, int& address);

private:
    int RankWidth;
    int BankWidth;
    int RowWidth;
    int ColWidth;
    int CellWidth;
};

#include "AddressTranslator.h"
#include "SystemConfiguration.h"
#include <sstream>

using namespace DRAMSim;

AddressTranslator::AddressTranslator()
{
    Init();
}

```



```

AddressTranslator::~AddressTranslator()
{}

void AddressTranslator::Init()
{
    RankWidth = dramsim_log2(NUM_RANKS);
    BankWidth = dramsim_log2(NUM_BANKS);
    RowWidth = dramsim_log2(NUM_ROWS);
    ColWidth = dramsim_log2(NUM_COLS);
    CellWidth = dramsim_log2(DEVICE_WIDTH);
}

int AddressTranslator::GetValue(int width, int& address)
{
    int temp = address;
    temp >>= width;
    temp <<= width;
    int n = temp ^ address;
    address >>= width;
    return n;
}

void AddressTranslator::SetValue(int width, int value, int& address)
{
    address <<= width;
    address |= value;
}

void AddressTranslator::Translate(int address, int& rank, int& bank, int& row,
    int& col)
{
    col = GetValue(ColWidth, address);
    row = GetValue(RowWidth, address);
    bank = GetValue(BankWidth, address);
    rank = GetValue(RankWidth, address);
}

void AddressTranslator::Translate(int address, int& rank, int& bank, int& row,
    int& col, int& bit)
{
    bit = GetValue(CellWidth, address);
    col = GetValue(ColWidth, address);
    row = GetValue(RowWidth, address);
    bank = GetValue(BankWidth, address);
    rank = GetValue(RankWidth, address);
}

int AddressTranslator::TranslateToAddr(int rank, int bank, int row, int col,
    int bit)
{
    int address = 0;
    SetValue(RankWidth, rank, address);

```

```

        SetValue(BankWidth, bank, address);
        SetValue(RowWidth, row, address);
        SetValue(ColWidth, col, address);
        if (bit >= 0)
            SetValue(CellWidth, bit, address);
        return address;
    }

std::string AddressTranslator::GetDescription(int address, bool includeBit)
{
    std::ostringstream stream;
    int r = 0, b = 0, row = 0, col = 0, bit = 0;
    if (includeBit)
        Translate(address, r, b, row, col, bit);
    else
        Translate(address, r, b, row, col);

    stream << "r:" << r << ", b:" << b << ", row:" << row << ", col:" << col;
    if (includeBit)
        stream << ", bit:" << bit;
    return stream.str();
}

#pragma once
#include "AddressTranslator.h"

struct Address
{
public:
    Address(int r = 0, int b = 0, int row = 0, int col = 0, int bit = 0);
    Address(int address, bool full);

    void Clear();
    int GetPhysical();

    friend bool operator==(const Address& a, const Address& b)
    {
        return a.rank == b.rank &&
            a.bank == b.bank &&
            a.row == b.row &&
            a.col == b.col &&
            a.bit == b.bit;
    }

public:
    int rank;
    int bank;
    int row;
    int col;
    int bit;

public:
    static void InitTranslator();

```

```

        static AddressTranslator translator;
};

#include "Address.h"
#include "SystemConfiguration.h"

AddressTranslator Address::translator;

Address::Address(int r, int b, int row, int col, int bit)
: rank(r), bank(b), row(row), col(col), bit(bit)
{
}

Address::Address(int address, bool full)
{
    Clear();
    if (full)
        translator.Translate(address, rank, bank, row, col, bit);
    else
        translator.Translate(address, rank, bank, row, col);
}

void Address::Clear()
{
    rank = bank = row = col = bit = 0;
}

int Address::GetPhysical()
{
    return translator.TranslateToAddr(rank, bank, row, col, bit);
}

void Address::InitTranslator()
{
    translator.Init();
}

FaultType GetFaultTypeFromString(string type)
{
    if (type == "SAF") return SAF;
    if (type == "TF") return TF;
    if (type == "CFin") return CFin;
    if (type == "CFid") return CFid;
    if (type == "CFst") return CFst;
    if (type == "AF") return AF;
    return NONE;
}

vector<Fault> ParseCSV(string filePath)
{
    vector<Fault> faults;
    ifstream file(filePath);
    while (file.good())

```

```

{
    string line;
    getline(file, line);

    Fault* fault = 0;
    int paramCount = 0;

    stringstream strstr(line);
    string value = "";

    while (getline(strstr, value, '\t'))
    {
        if (value == "SAF" || value == "TF" || value == "CFid" ||
            value == "CFin" || value == "CFst" || value == "AF")
        {
            if (GetFaultTypeFromString(value) != NONE)
            {
                faults.push_back(Fault());
                fault = &faults.back();
                fault->type = GetFaultTypeFromString(value);
            }
        }
        else
        {
            if (fault)
            {
                paramCount++;
                switch (fault->type)
                {
                    case SAF:
                    case TF:
                        if (paramCount == 1)
                        {
                            istringstream iss(value.substr(2));
                            iss >> hex >> fault->victimAddress;
                        }
                        if (paramCount == 2)
                        {
                            istringstream iss(value);
                            iss >> dec >> fault->victimValue;
                        }
                        break;
                    case CFid:
                    case CFst:
                        if (paramCount == 1 || paramCount == 3)
                        {
                            istringstream iss(value.substr(2));
                            if (paramCount == 1)
                                iss >> hex >> fault->
                                    victimAddress;
                            else
                                iss >> hex >> fault->
                                    agressorAddress;
                        }
                    }
                }
            }
        }
    }
}

```

```

    }
    if (paramCount == 2 || paramCount == 4)
    {
        stringstream iss(value);
        if (paramCount == 2)
            iss >> dec >> fault->
                victimValue;

        else
            iss >> dec >> fault->
                agressorValue;
    }
    break;
case CFin:
    if (paramCount == 1 || paramCount == 3)
    {
        stringstream iss(value.substr(2));
        if (paramCount == 1)
            iss >> hex >> fault->
                victimAddress;

        else
            iss >> hex >> fault->
                agressorAddress;
    }
    if (paramCount == 4)
    {
        stringstream iss(value);
        iss >> dec >> fault->agressorValue;
    }
    break;
case AF:
    if (paramCount == 1 || paramCount == 3)
    {
        stringstream iss(value.substr(2));
        if (paramCount == 1)
            iss >> hex >> fault->
                victimAddress;

        else
            iss >> hex >> fault->
                agressorAddress;
    }
}
}
}
}
}
return faults;
}

```