МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»

Высшая школа электроники и компьютерных наук
Кафедра «Электронные вычислительные машины»

РАБОТА ПРОВЕРЕНА
Рецензент
____
«___»___________________ 2019 г.

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой ЭВМ
_______________ Г.И. Радченко
«___»__________ 2019 г.

Разработка алгоритмов операций сложения чисел с плавающей запятой в ассоциативных решающих полях

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

Руководитель работы,
к.т.н., доцент каф. ЭВМ
______________И.Л. Кафтанников
«___»___________2019 г.

Автор работы,
студент группы КЭ-222
________________Д.И. Захаров
«___»___________2019 г.

Нормоконтролёр,
ст. преп. каф. ЭВМ
________________ С.В. Сяськов
«___»____________2019 г.

Челябинск-2019

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Высшая школа электроники и компьютерных наук
Кафедра «Электронные вычислительные машины»

УТВЕРЖДАЮ
Заведующий кафедрой ЭВМ
_______________ Г.И. Радченко
«___»_________2019 г.

ЗАДАНИЕ
на выпускную квалификационную работу бакалавра
студенту группы КЭ-452
Захарова Данилы Игоревича
обучающемуся по направлению
09.03.01 «Информатика и вычислительная техника»

1. **Тема работы: «**Разработка алгоритмов операций сложения чисел с плавающей запятой в ассоциативных решающих полях» утверждена приказом по университету от 25 апреля 2019 г. №899

2. **Срок сдачи студентом законченной работы:** 1 июня 2019 г.

3. **Исходные данные к работе:**

– Goldberg D. What Every Computer Scientist Should Know About Floating-Point Arithmetic – XeroxPaloAltoResearchCenter, 1991 – 94 с.;
– Н.А. Караван, Системный анализ формирования признаков для поиска информации в ассоциативных запоминающих устройствах / Н.А. Караван, Я.В. Корпань, Д.А. Лукашенко, К.С. Рудаков;

– Imani M., CAP: Configurable resistive associative processor for near-data computing [Electronic resource] / M. Imani // IEEE –2017г. – 2017 18th International Symposium on Quality Electronic Design – Режим доступа: https://www.academia.edu/32985302/CAP_Configurable_Resistive_Associative_Processor_for_Near-Data_Computing – Заглавие с экрана.

4. **Перечень подлежащих разработке вопросов:**

– рассмотрение существующих способов вычисления арифметических функций в процессорах;
– изучение архитектуры ассоциативных процессоров;
– разработка алгоритмов для вычисления арифметических функций на ассоциативном процессоре;
– анализ работоспособности разработанных алгоритмов;
– оценка целесообразности создание рабочего прототипа ассоциативного процессора для вычисления арифметических функций;

5. **Дата выдачи задания:** 1 декабря 2018 г.

Руководитель работы ________________________________/*И.Л. Кафтанников*/

Студент _________________________________/*Д.И. Захаров* /

КАЛЕНДАРНЫЙ ПЛАН

Этап	Срок сдачи	Подпись руководителя
Введение и обзор литературы	01.03.2019	
Разработка алгоритмов	01.04.2019	
Реализация демонстрационного приложения	01.05.2019	
Компоновка текста работы и сдача на нормоконтроль	24.05.2019	
Подготовка презентации и доклада	30.05.2019	

Руководитель работы ______________________________/*И.Л. Кафтанников*/

Студент ________________________________/*Д.И. Захаров*/

Аннотация

Д.И. Захаров. Разработка алгоритмов операций сложения чисел с плавающей запятой в ассоциативных решающих полях. – Челябинск: ФГАОУ ВО «ЮУрГУ (НИУ)», ВШЭКН; 2019, 89 с., 17 ил., библиогр. список – 18 наим.

В рамках выпускной квалификационной работы производится детальный анализ современных методов решения арифметических функций на базе вычислительных устройств. Проанализировано текущее состояние индустрии производства и разработки ассоциативных процессоров. Рассматриваются преимущества и недостатки вычислений арифметических функций на ассоциативных процессорах. Разработаны алгоритмы вычисления арифметических функций на ассоциативном процессоре. Доказывается целесообразность предполагаемой архитектуры для проведения вычислений.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 9
1. ПРЕДМЕТНАЯ ОБЛАСТЬ И АНАЛИЗ ИСТОЧНИКОВ ИНФОРМАЦИИ 11
1.1. ВЫПОЛНЕНИЕ АРИФМЕТИЧЕСКИХ ФУНКЦИЙ В ПРОЦЕССОРЕ 11
1.1.1. ВВЕДЕНИЕ 11
1.1.2. ЦЕЛЫЕ ЧИСЛА 12
1.1.2.1. ПРЕДСТАВЛЕНИЕ ЦЕЛЫХ ЧИСЕЛ 12
1.1.2.2. СЛОЖЕНИЕ ЦЕЛЫХ ЧИСЕЛ 12
1.1.2.3. СДВИГ ЦЕЛЫХ ЧИСЕЛ 13
1.1.3. ЧИСЛА С ПЛАВАЮЩЕЙ ТОЧКОЙ 15
1.1.3.1. ПРЕДСТАВЛЕНИЕ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ 15
1.1.3.2. СЛОЖЕНИЕ И ВЫЧИТАНИЕ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ 18
1.2. ВЫПОЛНЕНИЕ АРИФМЕТИЧЕСКИХ ФУНКЦИЙ НА БАЗЕ СУЩЕСТВУЮЩИХ АРХИТЕКТУР ПРОЦЕССОРОВ 18
1.2.1. ВИДЫ АРХИТЕКТУР И ИХ РАЗЛИЧИЯ 18
1.2.2. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ В ЦЕНТРАЛЬНОМ ПРОЦЕССОРЕ 20
1.2.3. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ В ГРАФИЧЕСКОМ ПРОЦЕССОРЕ 22
1.3. МЕТРИКИ БЫСТРОДЕЙСТВИЯ ВЫПОЛНЕНИЯ ОПЕРАЦИИ .. 25
1.3.1. ВВЕДЕНИЕ 25
1.3.2. CPI 25
1.3.3. IPS 26

1.3.4. FLOPs (GFLOPs, TFLOPs) 26
1.4. ВЫВОД 27
2. АССОЦИАТИВНЫЕ ПРОЦЕССОРЫ 28
2.1. АРХИТЕКТУРА И ПРИНЦИП РАБОТЫ АССОЦИАТИВНЫХ ПРОЦЕССОРОВ 28
2.2. ТЕКУЩЕЕ ИСПОЛЬЗОВАНИЕ 29
2.2.1. ИСПОЛЬЗОВАНИЕ В ПРОМЫШЛЕННЫХ МАСШТАБАХ 29
2.2.2. ИССЛЕДОВАНИЕ ТЕХНОЛОГИИ АССОЦИАТИВНЫХ ПРОЦЕССОРОВ 30
3. РАЗРАБОТКА АЛГОРИТМОВ ВЫПОЛНЕНИЯ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ НА БАЗЕ АССОЦИАТИВНОГО ПРОЦЕССОРА 34
3.1. ТРЕБОВАНИЯ К АЛГОРИТМАМ 34
3.2. РАЗРАБОТКА АЛГОРИТМОВ 34
3.2.1. АЛГОРИТМЫ ДЛЯ ЦЕЛЫХ ЧИСЕЛ 34
3.2.1.1. АЛГОРИТМ СЛОЖЕНИЯ ЦЕЛЫХ ЧИСЕЛ 34
3.2.1.2. АЛГОРИТМ ВЫЧИТАНИЯ ЦЕЛЫХ БЕЗЗНАКОВЫХ ЧИСЕЛ 38
3.2.1.3. АЛГОРИТМ СДВИГА ЦЕЛЫХ ЧИСЕЛ 41
4.2.1.4. АЛГОРИТМ СРАВНЕНИЯ ЦЕЛЫХ ЧИСЕЛ 46
3.2.1.5. ПОЛУЧЕНИЕ ПРОТИВОПОЛОЖНОГО ЧИСЛА 48
3.2.2. АЛГОРИТМЫ ДЛЯ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ 51
3.2.2.1. АЛГОРИТМ СЛОЖЕНИЯ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ 51
4. РАЗРАБОТКА ДЕМОНСТРАЦИОННОГО ПРИЛОЖЕНИЯ 58
4.1. АКТУАЛЬНОСТЬ ДЕМОНСТРАЦИОННОГО ПРИЛОЖЕНИЯ 58
4.2. ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ 58
4.3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ 59

4.4. РАБОТА ПРИЛОЖЕНИЯ 59
5. АНАЛИЗ БЫСТРОДЕЙСТВИЯ СОСТАВЛЕННЫХ АЛГОРИТМОВ. 63
5.1. ВЫБОР МЕТРИКИ БЫСТРОДЕЙСТВИЯ........ 63
5.2. АНАЛИЗ БЫСТРОДЕЙСТВИЯ АЛГОРИТМОВ 63
5.2.1. СЛОЖЕНИЕ ЦЕЛЫХ ЧИСЕЛ 63
5.2.2 СДВИГ........ 64
5.2.3. СЛОЖЕНИЕ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ 65
5.3. ВЫВОДЫ 66
6. ЗАКЛЮЧЕНИЕ 67
БИБЛИОГРАФИЧЕСКИЙ СПИСОК........ 68
ПРИЛОЖЕНИЕ А ИСХОДНЫЙ КОД МОДУЛЕЙ ДЕМОНСТРАЦИОННОГО ПРИЛОЖЕНИЯ........ 71

ВВЕДЕНИЕ

В настоящее время происходит поиск увеличения производительности вычислительных процессоров. Уже невозможно добиться мощностей, просто уменьшая размер транзисторов на интегральной схеме процессора. На выставке CES2019 соучредитель и генеральный директор NVIDIA подтвердил, что закон Мура, согласно которому число транзисторов в микросхеме удваивалось, не работает. [1] Затраты и сложность непрерывного размещения большого числа элементов на небольшой площади вынуждают искать новые архитектуры для более быстрого вычисления определенных алгоритмов. Одной из таких архитектур является ассоциативное запоминающее устройство (АЗУ).

Термин «ассоциативная» или «память с адресацией по содержимому» обозначается для класса запоминающих устройств, в которых к данным обеспечивается доступ не по адресу хранения, а по содержимому.

Применение АЗУ позволяет значительно ускорить поиск и обработку данных в больших массивах и обеспечивает удобное и компактное представление сложных алгоритмов решения информационно-логических задач – таких как планирование производства и материально-техническое снабжение, поиск научно-технической информации, поиск справочных данных о различных приборах, информационно-измерительных системах и других объектах.

В рамках данной работы будут рассмотрены операции хранения и сложения массива целых чисел и чисел с плавающей точкой на базе АП, а также проанализированы их скорость выполнения в сравнении с традиционной архитектурой процессоров конвейерных и матричных процессоров.

Целью создания системы обработки данных в АП является оптимизация обработки больших массивов данных. Уменьшение времени обработки операции происходит за счет:

- разработки новых алгоритмов обработки данных, которые потенциально быстрее существующих;
- параллельной обработки данных.

Для достижения поставленной цели необходимо решить следующие задачи:

- Проанализировать существующие схемотехнические методы реализации вычислительных устройств;
- Рассмотреть существующие архитектуры процессоров и то, как на них реализовано решение арифметических функций сложения целых чисел и чисел с плавающей точкой;
- Рассмотреть архитектуру ассоциативных процессоров и методы обработки информации по содержанию;
- Разработать алгоритмы вычисления арифметических функций на основе архитектуры ассоциативных процессоров;
- Проанализировать работоспособность разработанных алгоритмов;
- Произвести оценку быстродействия разработанных алгоритмов по сравнению с существующими;
- Сделать наглядную форму записи разработанных алгоритмов;
- Проанализировать целесообразность создания прототипа ассоциативного процессора;

1. ПРЕДМЕТНАЯ ОБЛАСТЬ И АНАЛИЗ ИСТОЧНИКОВ ИНФОРМАЦИИ

1.1. ВЫПОЛНЕНИЕ АРИФМЕТИЧЕСКИХ ФУНКЦИЙ В ПРОЦЕССОРЕ

1.1.1. ВВЕДЕНИЕ

Несмотря на то, что существует огромное множество различных микроархитектур процессоров, в целом они реализуют схожие арифметические алгоритмы.

Архитектура компьютера не определяет структуру аппаратного обеспечения, которое её реализует. Зачастую существуют разные аппаратные реализации одной и той же архитектуры. Например, компании Intel и Advanced Micro Devices (AMD) производят разные микропроцессоры, которые относятся к архитектуре x86. Все они могут выполнять одни и те же программы, но при этом в их основе лежит разное аппаратное обеспечение, поэтому эти процессоры имеют разное соотношение производительности, цены и энергопотребления. [2]

В рамках данной работы были рассмотрены форматы целых чисел и чисел с плавающей точкой. Целые числа необходимо рассмотреть, так как операции с числами с плавающей точкой в конечном счете сводятся к операциям сложения, вычитания и сдвига чисел с целой точкой. Числа в формате с фиксированной точкой разобраны не будут, так как операции с ними аналогичны выполнению операций с целыми числами.

В данном разделе будут рассмотрены основные принципы реализации арифметических функций на уровне микрокоманд процессора.

1.1.2. ЦЕЛЫЕ ЧИСЛА

1.1.2.1. ПРЕДСТАВЛЕНИЕ ЦЕЛЫХ ЧИСЕЛ

В памяти типовой компьютерной системы целое число представлено в виде цепочки битов фиксированного (кратного 8) размера. Эта последовательность нулей и единиц — не что иное, как двоичная запись числа, поскольку обычно для представления чисел в современной компьютерной технике используется позиционный двоичный код. Диапазон целых чисел, как правило, определяется количеством байтов в памяти компьютера, отводимых под одну переменную.

1.1.2.2. СЛОЖЕНИЕ ЦЕЛЫХ ЧИСЕЛ

Существуют два основных способа сложения в цифровых процессорах – последовательный и параллельный. При последовательном способе сложение операндов, находящихся в регистрах процессора, выполняется по тактам, начиная с младшего бита, и количество тактов равно разрядности процессора.

Достоинством последовательного сумматора является простота схемы, требующая минимального количества оборудования, недостатком – низкое быстродействие, т.к. для сложения кодов n-разрядных чисел требуется, учитывая возможность переполнения, n-1 такт работы.

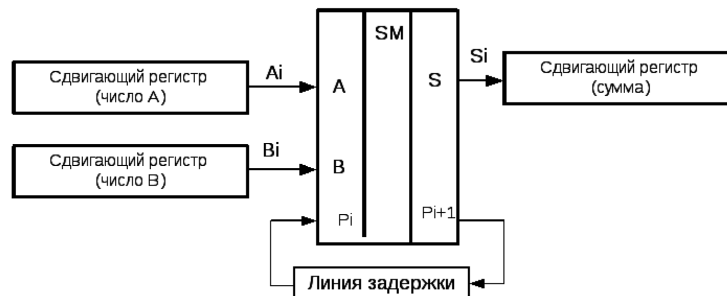

Рисунок 1 – Последовательный многоразрядный сумматор

При параллельном способе сложение операндов выполняется за один такт работы процессора, так как используется многоразрядный сумматор (число разрядов сумматора равно разрядности процессора).

Длительность формирования результата в таком сумматоре определяется временем установления выходных сигналов (сумма и перенос) в каждом из одноразрядных сумматоров после установления сигнала на его входах.

Надо учитывать, что если на входы Xi и Yi всех разрядов сигналы поступают в момент начала такта, то на вход Pi сигнал переноса поступает с некоторой задержкой, которая определяется длительностью переходных процессов в сумматоре предыдущего разряда.

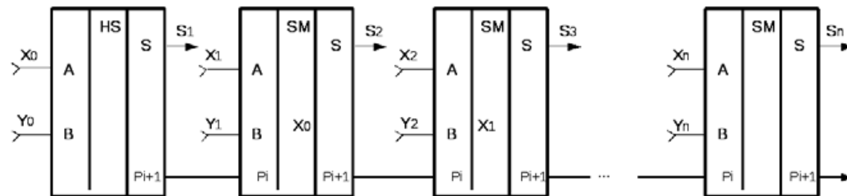

Рисунок 2 – Параллельный многоразрядный сумматор

1.1.2.3. СДВИГ ЦЕЛЫХ ЧИСЕЛ

Операция сдвига – это одновременное перемещение значений битов операнда в регистре процессора на фиксированное количество разрядов влево или вправо.

Различают три типа сдвига:

- логический;
- циклический;
- арифметический.

В нашей работе требуется арифметический сдвиг.

Арифметический сдвиг выполняется с учетом знака операнда, поэтому алгоритм его выполнения зависит от кода, в котором работает процессор. Арифметический сдвиг операнда влево на 1 бит эквивалентен увеличению его в два раза, а арифметический сдвиг вправо на 1 бит эквивалентен делению операнда на 2.

Первоначально, операция сдвига была реализована на логических элементах, однако позже была преобразована в интегральную схему на мультиплексорах.

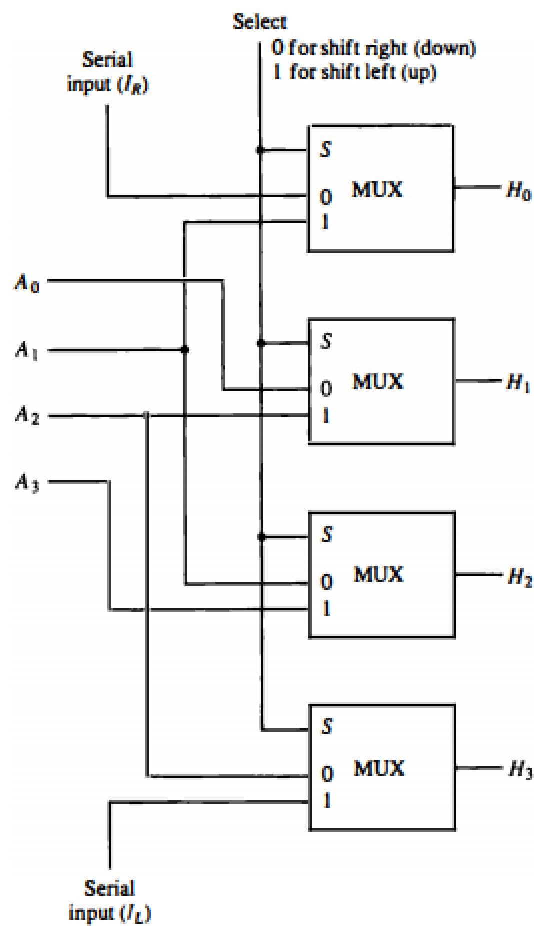

Рисунок 3 – Комбинационная схема сдвига

1.1.3. ЧИСЛА С ПЛАВАЮЩЕЙ ТОЧКОЙ

1.1.3.1. ПРЕДСТАВЛЕНИЕ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Операнды цифрового процессора в формате с плавающей точкой представляют числа в экспоненциальной форме. Такой формат используется в основном в научно-технических расчетах, когда диапазон чисел в вычислениях может варьироваться от очень малых величин до очень больших.

Число 2019.25_{10} в экспоненциальной форме можно представить как:

Математически это записывается как (1), где

- S – знак;
- M – мантисса числа;
- B – основание числа;
- E – порядок или экспонента числа;

Основание определяет систему счисления разрядов. Математически доказано, что числа с плавающей запятой с базой B=2 (двоичное представление) наиболее устойчивы к ошибкам округления, поэтому на практике встречаются только базы 2 и, реже, 10. Для дальнейшего изложения будем всегда полагать B=2, и формула числа с плавающей запятой будет иметь вид:

Примеры записи чисел с плавающей точкой представлены на таблицах 1 и 2.

Таблица 1. Компоненты записи чисел с плавающей точкой в формате IEEE 754

Число	**10,88**	**-0,3**
Двоичная форма	$1010.11100001010 = 1.01011100001010*2^{11}$	$-0.01001100110 = -1.001100110*2^{-10}$

Знак	0	1
Exp	$3 + 127 = 130_{10} =$ 0b10000010	-2 + 127 = 125 = 0b01111101
M	010 1110 0001 0100 0000 0000	001 1001 1000 0000 0000 0000

Окончание таблицы 1.

Таблица 2. Итоговая запись чисел с плавающей точкой

<table>
<tr><td></td><td>Знак</td><td colspan="8">Экспонента</td><td colspan="23">Мантисса</td></tr>
<tr><td>10,88</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>10,88</td><td>0</td><td colspan="4">8</td><td colspan="4">2</td><td colspan="3">2</td><td colspan="4">E</td><td colspan="4">1</td><td colspan="4">6</td><td colspan="4">0</td><td colspan="4">0</td></tr>
<tr><td>0,3</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0,3</td><td>1</td><td colspan="4">7</td><td colspan="4">D</td><td colspan="3">1</td><td colspan="4">9</td><td colspan="4">8</td><td colspan="4">0</td><td colspan="4">0</td><td colspan="4">0</td></tr>
</table>

При этом лишь некоторые из вещественных чисел могут быть представлены в памяти компьютера точным значением, в то время как остальные числа представляются приближёнными значениями.

Более простым вариантом представления вещественных чисел является вариант с фиксированной точкой, когда целая и вещественная части хранятся отдельно. Например, на целую часть отводится всегда X бит и на дробную отводится всегда Y бит. Такой способ в архитектурах процессоров не присутствует. Отдаётся предпочтение числам с плавающей запятой, как компромиссу между диапазоном допустимых значений и точностью.

Типы чисел с плавающей точкой:

– ***Число половинной точности*** — компьютерный формат представления чисел, занимающий в памяти половину машинного слова (в случае 32-битного компьютера — 16 бит или 2 байта). В силу невысокой точности этот формат представления чисел с плавающей запятой обычно

используется в видеокартах, где небольшой размер и высокая скорость работы важнее точности вычислений.

– ***Число одинарной точности*** — компьютерный формат представления чисел, занимающий в памяти одно машинное слово (32 бита или 4 байта). Используется для работы с вещественными числами везде, где не нужна очень высокая точность. В языках программирования представлен типом *float*.
– ***Число двойной точности*** — компьютерный формат представления чисел, занимающий в памяти два машинных слова (в случае 32-битного компьютера — 64 бита или 8 байт). Часто используется благодаря своей неплохой точности, даже несмотря на двойной расход памяти и сетевого трафика относительно чисел одинарной точности.
– ***Число четверной точности*** — компьютерный формат представления чисел, занимающий в памяти четыре машинных слова (в случае 32-битного компьютера — 128 бит или 16 байт). Используется в случае необходимости крайне высокой точности. Обычно этот формат реализуется программно, случаи аппаратной реализации крайне редки. Также не гарантируется поддержка этого типа в языках программирования, хотя кое-где она и реализована (например, компилятор gcc для архитектуры x86 позволяет использовать тип __float128, являющийся программной реализацией числа с четверной точностью). В совокупности эти факторы делают Quadruple весьма экзотичным и редко встречающимся форматом чисел с плавающей запятой.

1.1.3.2. СЛОЖЕНИЕ И ВЫЧИТАНИЕ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Сложение двух чисел X и Y происходит следующим образом: [15]

Операнды X и Y представляются как:

Напомним, что только при записи чисел в память в формате с плавающей точкой мантиссы хранятся в диапазоне 1<=M<2, а при извлечении из памяти в ОА скрытый бит восстанавливается и 0,5<=M<1.

Сначала требуется сравнить порядки и и выровнять их, если они не равны, для этого потребуется найти разность и сдвинуть мантиссу на число разрядов, равное искомой разности. Затем выполнить сложение мантисс и и записать полученное число в результирующую ячейку.

Полученная сумма Z=X+Y также должна быть представлена как

1.2. ВЫПОЛНЕНИЕ АРИФМЕТИЧЕСКИХ ФУНКЦИЙ НА БАЗЕ СУЩЕСТВУЮЩИХ АРХИТЕКТУР ПРОЦЕССОРОВ

1.2.1. ВИДЫ АРХИТЕКТУР И ИХ РАЗЛИЧИЯ

Большинство современных процессоров для персональных компьютеров основаны на той или иной версии циклического процесса последовательной обработки данных, изобретённого Джоном фон Нейманом.

Отличительной особенностью архитектуры фон Неймана является то, что инструкции и данные хранятся в одной и той же памяти. [12]

В различных архитектурах и для различных команд могут потребоваться дополнительные этапы. Например, для арифметических команд могут

потребоваться дополнительные обращения к памяти, во время которых производится считывание операндов и запись результатов.

Этапы цикла выполнения:

- Процессор выставляет число, хранящееся в регистре счётчика команд, на шину адреса и отдаёт памяти команду чтения.
- Выставленное число является для памяти адресом; память, получив адрес и команду чтения, выставляет содержимое, хранящееся по этому адресу, на шину данных и сообщает о готовности.
- Процессор получает число с шины данных, интерпретирует его как команду (машинную инструкцию) из своей системы команд и исполняет её.
- Если последняя команда не является командой перехода, процессор увеличивает на единицу (в предположении, что длина каждой команды равна единице) число, хранящееся в счётчике команд; в результате там образуется адрес следующей команды.

Данный цикл выполняется неизменно, и именно он называется процессом (откуда и произошло название устройства).

В настоящее время существует две основных архитектуры процессоров:

- RISC-процессоры (Redused instruction set computer) – вычисления с упрощённым набором команд. Характеризуются небольшим набором микрокоманд, выполняющихся за короткое время. Остальные операции должны быть приведены к набору этих микрокоманд.
- CISC-процессоры (Complex instruction set computer) – процессоры, выполняющие вычисления со сложным набором команд. Имеют расширенный набор команд, однако время их выполнения не фиксировано и может в разы превышать эквивалентный набор команд в RISC-процессорах. [2]

RISC применяется в микроконтроллерах Atmel и ARM-архитектуре, где не требуется большое количество операций (например, промышленные микроконтроллеры в электронных устройствах и мобильных устройствах)

CISC как правило применяется совместно с RISC-ядром в процессорах Intel. Процессоры Intel, начиная с процессора 486, содержат RISC-ядро, которое выполняет самые простые (и обычно самые распространенные) команды за один цикл тракта данных, а по обычной технологии CISC интерпретируются более сложные команды. В результате обычные команды выполняются быстро, а более сложные и редкие — медленно. Хотя при таком «гибридном» подходе производительность ниже, чем в архитектуре RISC, новая архитектура CISC имеет ряд преимуществ, поскольку позволяет использовать старое программное обеспечение без изменений. [8]

1.2.2. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ В ЦЕНТРАЛЬНОМ ПРОЦЕССОРЕ

Начиная с архитектуры x86 в процессорах фирмы Intel используются следующие расширения для выполнения арифметических операций:

– **SSE** (версии 1-5) (англ. Streaming SIMD Extensions — потоковое SIMD-расширение) — SIMD (англ. Single Instruction, Multiple Data — «одна инструкция — множество данных») набор инструкций. Позволяет выполнять потоковые вычисления над числами с плавающей точкой. В первой версии позволяло выполнять действия над 128 байтами, которые могли быть интерпретированы как два числа с плавающей точкой с двойной точностью, четыре числа с плавающей точкой с одинарной точностью, два 64-битных целых числа, четыре 32-битных целых числа, восемь 16-битных целых числа, шестнадцать 8-битных целых числа.

– **MMX** – Дополнительный «мультимедийный» (англ. *Multi-Media eXtensions*) набор инструкций, выполняющих по несколько характерных для процессов кодирования/декодирования потоковых аудио/видеоданных действий за одну машинную инструкцию. Обеспечивает только целочисленные вычисления.
– **FMA** (англ. *Fused Multiply-Add*, умножение-сложение с однократным округлением) — это набор опциональных 128- и 256-битных SIMD-инструкций для архитектур x86 и x64, предназначенный для выполнения операции умножения-сложения над числами в формате с плавающей запятой.

Недостатки выполнения расчетов на ЦП:

Со времени своего появления в начале 1980-х годов, персональные компьютеры развивались в основном как машины для выполнения программ, сложных по внутренней структуре, содержащих большое количество ветвлений, интенсивно взаимодействующих с пользователем, но редко связанных с потоковой обработкой большого количества однотипных данных.

Центральные процессоры ПК оптимизировались для решения именно таких задач, поэтому характеризовались следующим:

– большим количеством блоков для управления исполнением программы (кеширование данных, предсказание ветвлений и т.п.) и сравнительно малым количеством блоков для вычислений;
– архитектурой, оптимальной для программ со сложным потоком управления (обработка разнородных команд и данных, организация взаимодействия программ между собой и с пользователем);
– памятью с максимальной скоростью произвольного доступа к данным.

Увеличение производительности CPU в основном было связано с увеличением тактовой частоты и размеров высокоскоростной кэш-памяти (память, расположенная прямо на процессоре). Программирование CPU для ресурсоемких научных вычислений подразумевает тщательное структурирование данных и порядка инструкций для эффективного использования всех уровней кэш-памяти. Однако, развитие кэш-памяти породило ряд уязвимости в самой архитектуре процессора. Так, в 2017м году были открыты Meltdown и Spectre, присутствовавшие в современных процессорах в течение десятилетий, но не обнаруженные ранее из-за сложности современных вычислительных систем и, в частности, их взаимодействия с кэш-памятью. [11]

Также, на CPU ограничено многопоточное выполнение, так как изначально развитие шло в сторону выполнения мультикоманд MIMD – Multiple Instruction Multiple Data. Процессоры могут выполнять много параллельных потоков, но при этом гораздо хуже выполняют одинаковые операции, содержащие множество однотипных данных.

1.2.3. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ В ГРАФИЧЕСКОМ ПРОЦЕССОРЕ

Во второй половине 1990-х годов началось быстрое развитие графических процессоров (GPU) — дополнительных вычислительных устройств для ускоренного исполнения алгоритмов визуализации трехмерных сцен. Поскольку трехмерная визуализация допускает эффективное распараллеливание расчетов, графические процессоры разрабатывались как поточно-параллельные системы с большим количеством вычислительных блоков, конвейерной обработкой данных и памятью с максимальной пропускной способностью.

Концепция программирования, заключающаяся в потоковой обработке данных, известна под аббревиатурой SIMD (от англ. Single Instruction — Multiple Data — одна инструкция для множества данных). Процессор, работающий по принципу SIMD, преобразует поток данных в поток результатов, используя программу как функцию преобразования

Использование памяти различных типов обусловлено необходимостью баланса между объемом памяти и скоростью доступа к данным. Разновидности памяти, имеющие наибольшую емкость, обычно характеризуются большим временем доступа к данным, и наоборот. Быстродействие памяти, в свою очередь, определяется двумя характеристиками — латентностью и пропускной способностью.

Латентность — это время доступа к памяти, точнее, время ожидания процессором данных после запроса. Как можно меньшая латентность памяти необходима современным центральным процессорам, работающим на очень высоких частотах, еще и потому, что таким частотам должна соответствовать высокая скорость доступа к данным.

Пропускная способность памяти характеризует объем данных, которые могут быть переданы к процессору или от процессора за единицу времени. Высокая пропускная способность оказывается эффективнее низкой латентности в задачах, позволяющих организовать последовательный доступ к памяти — считывание (или запись) данных из ячеек памяти, расположенных друг за другом, непрерывным потоком. [10]

При поточно-параллельной обработке данных предпочтителен именно последовательный доступ к памяти, поэтому видеопамять, предназначенная для обмена данными с графическим процессором, должна обладать максимальной пропускной способностью даже в ущерб латентности.

Выполнение расчётов на GPU показывает отличные результаты в алгоритмах, использующих параллельную обработку данных. То есть, когда одну и ту же последовательность математических операций применяют к большому объёму данных. При этом лучшие результаты достигаются, если отношение числа арифметических инструкций к числу обращений к памяти достаточно велико. Это предъявляет меньшие требования к управлению исполнением (flow control), а высокая плотность математики и большой объём данных отменяет необходимость в больших 24ЭШах, как на CPU.

Поскольку существует ряд задач, которые легко поддаются распараллеливанию, то эффективнее использовать для них именно GPU. Например, его польза очевидна при вычислении пикселей отрендеренного изображения. Происходит ряд одинаковых процедур для каждого пикселя, все необходимые данные об объектах хранятся в ОЗУ видеокарты. Каждый потоковый процессор может независимо от другого посчитать свою часть изображения. Это позволит повысить общую производительность при решении большинства задач подобного плана. [9]

Недостатки GPU заключаются в изначально малом количестве арифметических микрокоманд. Так, например, некоторые графические процессоры не полностью поддерживают формат чисел с плавающей точкой, что не дает выполнить расчеты с векторами в полной мере.

Также можно отметить довольно высокую стоимость графических процессоров.

1.3. МЕТРИКИ БЫСТРОДЕЙСТВИЯ ВЫПОЛНЕНИЯ ОПЕРАЦИИ

1.3.1. ВВЕДЕНИЕ

Существуют разные способы измерения быстродействия вычислительной системы, а также стоимость выполнения конкретной операции.

Оценка реальной вычислительной мощности производится путём прохождения специальных тестов (бенчмарков) — набора программ, специально предназначенных для проведения вычислений и измерения времени их выполнения. Для данной работы невозможно произвести подобные измерения, поэтому будут представлены только теоретический анализ, характеризующий порядок количества операций, который однако позволит определить приблизительную мощность ассоциативных процессоров.

Рассмотрим некоторые из метрик.

1.3.2. CPI

CPI (англ. *cycles per instruction* – циклов за операцию) – среднее количество тактовых сигналов процессора, необходимых для выполнения операции.

Формула для вычисления CPI:

———— (2),

где IC_i – количество тактов в одной операции для данного типа инструкции,

IC – количество всего тактов в инструкции

Cci – скорость выполнения i-й операции

Величина, обратная CPI и также используемая в измерении быстродействия процессора – IPC (*instructions per cycle*).

1.3.3. IPS

IPS (англ. *instructions per second*) — мера быстродействия процессора компьютера. Показывает число определённых инструкций, выполняемых процессором за одну секунду. Часто заявляемые производителями значения IPS являются пиковыми и получены на последовательностях инструкций, не характерных для реальных программ. Также на значения IPS сильно влияет пропускная способность иерархии памяти.

На данный момент для обозначения быстродействия добавляют приставки гига- и мега-, так как количество операций в секунду достигает 10^6.

1.3.4. FLOPs (GFLOPs, TFLOPs)

Одной из характеристик процессора является FLOPs – внесистемная единица, используемая для измерения производительности компьютеров, показывающая, сколько операций с плавающей запятой в секунду выполняет данная вычислительная система.

Для подсчета максимального количества флопсов для процессора нужно учитывать, что современные процессоры в каждом своём ядре содержат несколько исполнительных блоков каждого типа (в том числе и для операций с плавающей запятой), работающих параллельно, и могут выполнять более одной инструкции за такт. Данная особенность архитектуры называется суперскалярность и впервые появилась ещё в самом первом процессоре Pentium в 1993 году.

Более новые процессоры могут исполнять до 8 (например, Sandy и Ivy Bridge, 2011—2012 гг, AVX) или до 16 (Haswell и Broadwell, 2013—2014 гг, AVX2 и FMA3) операций на 64-битными числами с плавающей запятой в такт (на каждом ядре).

1.4. ВЫВОД

Таким образом, в рамках данной работы необходимо преобразовать существующие алгоритмы вычисления арифметических функций для архитектуры ассоциативных процессоров и оценить быстродействие с помощью одной из метрик измерения быстродействия процессоров.

2. АССОЦИАТИВНЫЕ ПРОЦЕССОРЫ

2.1. АРХИТЕКТУРА И ПРИНЦИП РАБОТЫ АССОЦИАТИВНЫХ ПРОЦЕССОРОВ

Разработка устройств с адресацией по содержимому (ассоциативной памяти / ассоциативных процессоров) проводилась вплоть до 1990х годов, в частности, была разработана серия советских микросхем серии РА. Однако, стоимость электронных устройств и отсутствие задач для применения делали дальнейшее развитие данного типа устройств нецелесообразным и дорогостоящим.

Современный уровень развития новой элементной и технологической базы позволяет заново рассмотреть уже с современных представлений возможности ассоциативной памяти.

Напомним, что АП относится к SIMD структурам, причем уникальной возможностью АП является возможность анализа и преобразования данных непосредственно в памяти и с различной степенью локализации: бит, строка, столбец, массив, группа массивов. [4]

Действие АЗУ основано на представлении всей информации в виде ряда зон в зависимости от свойств и характерных признаков. При этом поиск информации сводится к определению зоны по заданным признакам путём просмотра и сравнения их с признаками, хранимыми в АЗУ. Модель ассоциативной памяти содержит блоки с набором ассоциативных признаков и основную память, схему сравнения, устройство управления, регистр признака и регистр выходной информации. [13]

Устройство N b-битовых слов показано на рисунке 4.

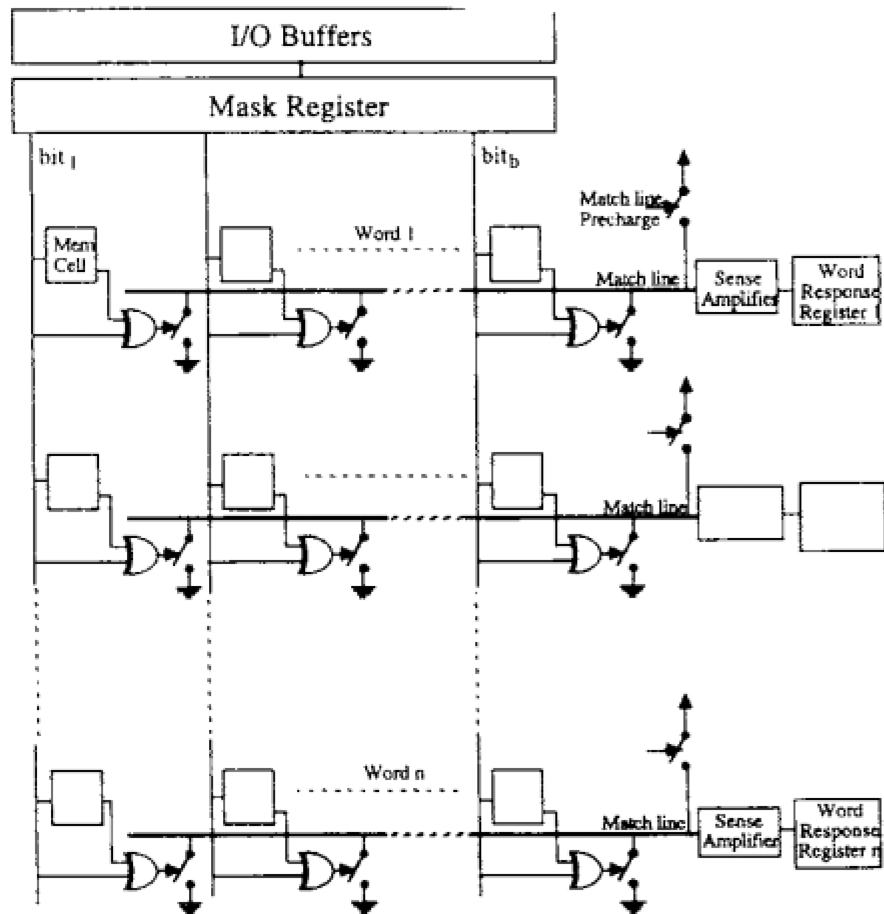

Рисунок 4 – Устройство ассоциативного процессора из N b-битовых слов

2.2. ТЕКУЩЕЕ ИСПОЛЬЗОВАНИЕ

2.2.1. ИСПОЛЬЗОВАНИЕ В ПРОМЫШЛЕННЫХ МАСШТАБАХ

На данный момент, элементы памяти с выборкой информацией по содержанию используются в средах, где необходим быстрый поиск информации из большого числа элементов.

Один из таких примеров – сетевое оборудование. Когда сетевой коммутатор получает фрейм данных на один из его портов, это обновляет внутреннюю таблицу с источником MAC-адреса фрейма и порта, на который он был получен. Потом он ищет MAC-адрес назначения в таблице, чтобы

определить, на какой порт фрейм должен быть отправлен, и отсылает его на этот порт. Таблица MAC- адресов обычно реализована на двоичной АП, таким образом порт назначения может быть найден очень быстро, уменьшая время ожидания коммутатора.

Также на данный момент ассоциативное запоминающее устройство реализуют или как сопроцессор SQL, или в виде платы, или в интегральном исполнении, или модуль встроен на кристалле центрального процессора, при этом модуль АЗУ располагается между центральным процессором и основной памятью. Как правило, модуль ассоциативной памяти является вторым уровнем кэш-памяти процессора. Также встречается частично-ассоциативная память, состоящая из многих модулей ассоциативной памяти, внутри которых идет адресация по содержанию, но доступ к каждому из модулю происходит по адресу.

2.2.2. ИССЛЕДОВАНИЕ ТЕХНОЛОГИИ АССОЦИАТИВНЫХ ПРОЦЕССОРОВ

На данный момент АП является одним из способов повышения быстродействия выполнения алгоритмов и во всем мире проходят исследования технологии АП.

В 2015 году в IIT (Israel Institute of Technology, г. Хайфа, Израиль) при поддержке Intel CollaborativeResearch Institute for Computational Intelligence вышла статья, исследовавшая зависимость энергопотребления ЦП и АП одинаковых вычислительных мощностей и быстроту обработки информации. Результаты показывают, что при использовании от $2.7*10^8$ ячеек информации начинается резкое увеличение эффективности выполнения операций, в частности, уменьшения количества тактов на выполнимую операцию (CPI). Также исследователи отмечают более стабильное энергопотребление, по

сравнению с графическими процессорами, так как узлы ассоциативного процессора распределены равномерней.

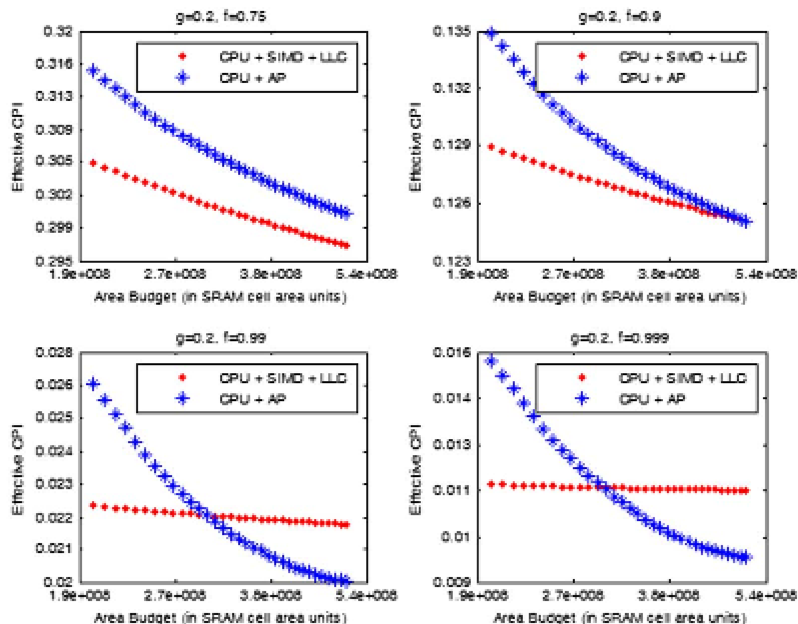

Рисунок 5 – Показатели эффективности CPI в зависимости от тактовых частот для разных архитектур процессоров

В 2017 году на международном симпозиуме IEEE была представлена работа « CAP: Configurable Resistive Associative Processor for Near-Data Computing». В данной работе были проанализированы перспективы использования ассоциативных процессоров для систем, в которых необходима обработка больших массивов информации (в частности, интернет вещей) и создан прототип надстройки ассоциативного процессора на графический процессор для уменьшения времени обработки информации. В результате, было определено, что можно добиться выигрыша в энергопотреблении в 9.4 раз, а в скорости вычислений в 4.1 раз. На рисунке 6 изображена единичная ячейка, обеспечивающая ассоциативный поиск.

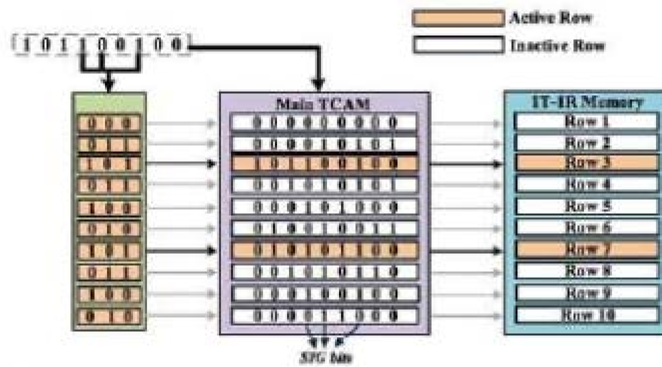

Рисунок 6 – Структура единичной ячейки, обеспечивающей ассоциативный поиск

Развитие ассоциативных процессоров, происходящее в наше время может быть обусловлено улучшением магнито-резистивной оперативной памяти, обеспечивающей достаточную быстроту извлечения данных при сравнительно небольших габаритах размещения на чипе. В работе «Resistive Content Addressable Memory for ConfigurableApproximation» был создан прототип ускорителя для оперативной памяти на базе памяти с выборкой по содержанию (ассоциативной памяти). В результате был достигнут выигрыш в энергопотреблении в 8 раз и в скорости вычислений в 4 раза.

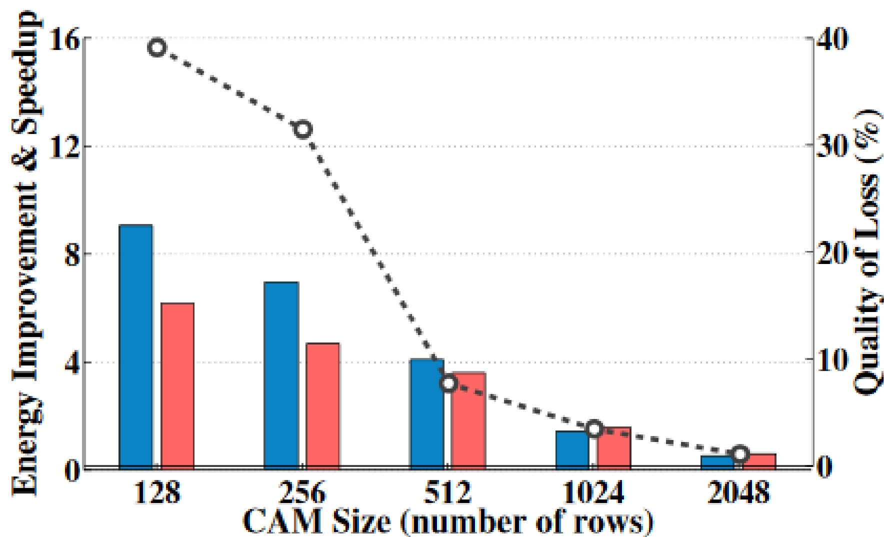

Рисунок 7 – Результаты выполнения бенчмарка BlackScholes

В России на данный момент исследованием АП занимается Институт вычислительной математики и математической геофизики СО РАН в Новосибирске. Согласно результатам работы «Решение задач на графах с помощью STAR-машины, реализованной на графических ускорителях», реализация алгоритма Уоршалла, выполненная на ассоциативном процессоре, позволяет произвести поиск кратчайшего пути в графе от 2 до 2000 раз быстрее, чем в последовательном процессоре с выборкой по адресу.

Вышеперечисленные исследования дают понять, что данная область разработки довольно перспективна и требует дальнейшего изучения. В рамках данной работы была исследована целесообразность выполнения арифметических функций на ассоциативном процессоре.

3. РАЗРАБОТКА АЛГОРИТМОВ ВЫПОЛНЕНИЯ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ НА БАЗЕ АССОЦИАТИВНОГО ПРОЦЕССОРА

3.1. ТРЕБОВАНИЯ К АЛГОРИТМАМ

Разрабатываемые алгоритмы должны отвечать следующим принципам:

- Массовость – алгоритм должен выполняться для любых чисел, хранящихся в ячейках памяти АП;
- Определенность – алгоритм обработки чисел должен быть определен однозначно;
- Дискретность – алгоритм должен быть разбит на отдельные конечные шаги, выполнение очередного начинается после завершения предыдущего;
- Конечность – исполнение алгоритма заканчивается после прохождения всех шагов, когда все данные обработаны процессором.

3.2. РАЗРАБОТКА АЛГОРИТМОВ

3.2.1. АЛГОРИТМЫ ДЛЯ ЦЕЛЫХ ЧИСЕЛ

3.2.1.1. АЛГОРИТМ СЛОЖЕНИЯ ЦЕЛЫХ ЧИСЕЛ

Согласно свойствам памяти с адресацией по содержанию, мы можем производить операции либо параллельно по разрядам и последовательно по элементам массива, либо наоборот – параллельно по элементам массива, последовательно по разрядам. Первый вариант проще реализовать на классической элементной схеме (сумматоры), разберем второй.

Схема сложения двоичных чисел следующая:

Пусть в ячейках АП хранятся следующие числа:

Таблица 3. Содержимое ячеек ассоциативного процессора

A								B								S								P							
0	0	0	1	1	1	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	0	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	0	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

где A – первый операнд, B – второй операнд, S – разряды суммы, P – разряды переноса.

Выборка опроса-записи для данных операнд соответствует таблице сумматора:

Таблица 4. Выходные значения полного сумматора.

a_j	b_j	P_{j-1}	S_j	P_j
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Где aj и bj – текущие биты операнд, pj-1 – перенос с предыдущего бита, Sj и Pj – соответственно сумма и перенос текущего бита.

Соответственно, Si = 1 при 001, 010, 100 и 111, а Pi = 1 при 011, 101, 110, 111. В дальнейшем нам потребуется проверять разряды в элементах массива на соответствие этим числам.

Последовательность действий при сложении векторов, состоящих из 16-разрядных целых чисел на АП:

1. Выбираем нулевой разряд у всех элементов обоих массивом, для этого маскируем остальные 15 разрядов. (строка «Опрос 0.1»)
2. Проверяем содержимое разрядов на соответствие таблице. Так как это начало сложения, переноса еще нет, поэтому будем действовать по упрощенной схеме:

Таблица 5. Схема заполнения разрядов S_0 и P_0 в зависимости от a_0 и b_0

a_0	b_0	S_0	P_0
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

3. Проверяем различные значения ячеек a и b:

– a = 0, b = 0. При этом значении и сумма, и перенос равны 0, значит эти значения нас не особо интересуют;
– a = 0, b = 1. При этом S = 1, производим опрос. На шинах опроса данных элементов выставляется «1»;
– a = 1, b = 0. При этом S = 1. На шинах опроса данных элементов выставляется «1»;

– Производим запись в разряд s[i]. Для этого накладываем маску на все разряды, кроме необходимого (s[0]), а затем в него производим запись «1» в те строки, на шинах которых стоит «1»;
– a = 1, b = 1. При этом S = 1, P = 1, производим поиск, на шинах опроса соответствующих строк выставляется «1»;

4. Производим запись в строки тех чисел, на шинах которых выставлена единица в служебный разряд p[0]. Для этого сначала маскируем все остальные разряды ячейки, затем записываем;
5. Производим аналогичные шаги для следующего разряда a[1], b[1]. Но при этом, мы должны учитывать бит переноса, записанный в служебном разряде;
6. Повторяем п. 3-5 пока произойдет сложение по всем разрядам;

Математическая запись алгоритма:

(3)

(4)

(5)

(6)

(7)

(8)

(9)

(10)

(11)

(12)

где

– , – результаты опроса 8 разряда;

– – результат записи в разряд суммы 8;

– – результат записи в разряд переноса 8;

– , , – результаты опроса j-го разряда;

– , – результаты записи в разряд суммы j-го разряда;

– – результаты записи в разряд переноса j-го разряда;

– – значение бита первого операнда j-го разряда;

– – значение бита второго операнда j-го разряда;

– – значение бита маскирования.

3.2.1.2. АЛГОРИТМ ВЫЧИТАНИЯ ЦЕЛЫХ БЕЗЗНАКОВЫХ ЧИСЕЛ

Алгоритм вычитания двух целых беззнаковых чисел происходит следующим образом.

Нам понадобится 8 бит S для записи результата и 1 бит C для записи займа. Начинаем опрос значений операнд с последнего разряда и движемся в начало.

Последовательность действий при вычитании векторов, состоящих из 8-разрядных целых чисел на АП:

1. Для первого шага всегда C = 0. Значения разрядов операнд может быть следующим:

Таблица 6. Схема заполнения разрядов S_0 и C_0 в зависимости от a_0 и b_0

a_0	b_0	S_0	C_0

0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Окончание таблицы 6.

2. Проверяем значения нулевых разрядов операнд:

– $a_0 = 0$, $b_0 = 0$; $a_0 =1$, $b_0 = 1$. При этом значении и сумма, и перенос равны 0, значит эти значения нас не интересуют, поэтому не производим поиск данных значений.

– a = 0, b = 1. При этом S = 1, C = 1, следовательно необходимо провести опрос. На шинах опроса данных элементов выставляется «0» для a_0, «1» для b_0. В найденные строки на шины записи выставляется «1» в разряды S_0, C_0.

– a = 1, b = 0. . При этом S = 1, производим опрос. На шинах опроса данных элементов выставляется «1» для a_0, «0» для b_0. В найденные строки на шины записи выставляется «1» в разряд S_0.

3. Проверяем следующий разряд операнд. Возможные значения:

Таблица 7. Схема заполнения разрядов S_i и C_i в зависимости от a_i, b_i и C_{i-1}

a_i	b_i	C_{i-1}	S_i	C_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1

1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Окончание таблицы 7.

- $a_i = 0$, $b_i = 0$, $C_{i-1} = 0$; $a_0 = 1$, $b_0 = 1$, $C_{i-1} = 0$; $a_0 = 1$, $b_0 = 0$, $C_{i-1} = 1$. При этом значении и разность, и перенос равны 0, значит эти значения нас не интересуют, поэтому не производим поиск данных значений.
- $a_i = 0$, $b_i = 1$, $C_{i-1} = 1$. При этом $C_i = 1$, следовательно необходимо провести опрос. На шинах опроса данных элементов выставляется «0» для a_i, «1» для b_i, «1» для C_{i-1},. В найденные строки на шины записи выставляется «1» в разряды C_i.
- $a_i = 1$, $b_i = 0$, $C_{i-1} = 0$. При этом $S_i = 1$, следовательно необходимо провести опрос. На шинах опроса данных элементов выставляется «1» для a_i, «0» для b_i, «0» для C_{i-1},. В найденные строки на шины записи выставляется «1» в разряды S_i.
- $a_i = 0$, $b_i = 0$, $C_{i-1} = 1$; $a_i = 0$, $b_i = 1$, $C_{i-1} = 0$; $a_i = 1$, $b_i = 1$, $C_{i-1} = 1$. При этом $S_i = 1$, $C_i = 1$, следовательно необходимо провести опрос. На шинах опроса данных элементов выставляются необходимые значения для a_i, b_i, и C_{i-1},. В найденные строки на шины записи выставляется «1» в разряды C_i и S_i.

4. Переходим на следующий разряд и повторяем шаг 3.

Математическая запись алгоритма:

(13)

(14)

(15)

(16)

(17)

(18)

(19)

(20)

(21)

(22)

(23)

где

– , – результаты опроса 8 разряда;

– – результат записи в разряд разности 8;

– – результат записи в разряд переноса 8;

– , , – результаты опроса j-го разряда;

– , – результаты записи в разряд суммы j-го разряда;

– , – результаты записи в разряд переноса j-го разряда;

– – значение бита первого операнда j-го разряда;

– – значение бита второго операнда j-го разряда;

– – значение бита маскирования;

3.2.1.3. АЛГОРИТМ СДВИГА ЦЕЛЫХ ЧИСЕЛ

Пусть в строке K находятся 2 числа: A из 8 разрядов и B из 3 разрядов. Необходимо вычислить число A, сдвинутое на B разрядов. Направление сдвига

значения не имеет, будут меняться лишь ячейки записи. Для примера возьмем сдвиг вправо.

Так как B – число, записанное в позиционной системе счисления, цифру, стоящую в каждом разряде этого числа можно преобразовать в число ячеек сдвига. Значение, полученное после сдвига будем записывать в ту же строку в 8-разрядное число S.

Таблица 8. Содержимое ячеек ассоциативного процессора

A								B			S							
1	0	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	1	1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0

Порядок действий при сдвиге вправо векторов A на число, указанное в B:

1. Опрашиваем все строки, в которых нулевой разряд $B_0 = 0$. В найденных строках производим запись числа A в S без маскирования разрядов:

Таблица 9. Содержимое ячеек АП после прохождения шага 1.

A								B			S							
1	0	1	1	1	1	0	1	0	0	**0**	1	0	1	1	1	1	0	1

1	0	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	1	0	1	**<u>0</u>**	1	0	1	1	1	1	0	1
1	0	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	1	1	0	**<u>0</u>**	1	0	1	1	1	1	0	1
1	0	1	1	1	1	0	1	1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	1	1	1	**<u>0</u>**	1	0	1	1	1	1	0	1
1	0	1	1	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0

Окончание таблицы 9.

2. Опрашиваем все строки, в которых нулевой разряд $B_0 = 1$. В найденных строках производим запись числа A в S, при этом необходимо произвести маскирование последнего разряда A, а при записи в S первый разряд заполнить 0:

Таблица 10. Содержимое ячеек АП после прохождения шага 2.

A								B			S							
1	0	1	1	1	1	0	1	0	0	0	1	0	1	1	1	1	0	1
1	0	1	1	1	1	0	1	0	0	**<u>1</u>**	0	1	0	1	1	1	1	0
1	0	1	1	1	1	0	1	0	1	0	1	0	1	1	1	1	0	1
1	0	1	1	1	1	0	1	0	1	**<u>1</u>**	0	1	0	1	1	1	1	0
1	0	1	1	1	1	0	1	1	0	0	1	0	1	1	1	1	0	1
1	0	1	1	1	1	0	1	1	0	**<u>1</u>**	0	1	0	1	1	1	1	0
1	0	1	1	1	1	0	1	1	1	0	1	0	1	1	1	1	0	1
1	0	1	1	1	1	0	1	1	1	**<u>1</u>**	0	1	0	1	1	1	1	0

3. Опрашиваем все строки, в которых $B_1 = 1$. В найденных строках делаем выборку числа S, при этом необходимо маскировать 2 последних разряда. Запись производится в это же число, при этом первые два разряда заполняются 0;

Таблица 11. Содержимое ячеек АП после прохождения шага 3.

B			S							
0	0	0	1	0	1	1	1	1	0	1
0	0	1	0	1	0	1	1	1	1	0
0	**1**	0	0	0	1	0	1	1	1	1
0	**1**	1	0	0	0	1	0	1	1	1
1	0	0	1	0	1	1	1	1	0	1
1	0	1	0	1	0	1	1	1	1	0
1	**1**	0	0	0	1	0	1	1	1	1
1	**1**	1	0	0	0	1	0	1	1	1

4. Опрашиваем все строки, в которых B2 = 1. В найденных строках делаем выборку числа S, при этом необходимо маскировать 4 последних разряда. Запись производится в это же число, при этом первые 4 разряда заполняются 0;

Таблица 12. Содержимое ячеек АП после прохождения шага 4.

B			S							
0	0	0	1	0	1	1	1	1	0	1
0	0	1	0	1	0	1	1	1	1	0
0	1	0	0	0	1	0	1	1	1	1

0	1	1	0	0	0	1	0	1	1	1
1	0	0	0	0	0	0	1	0	1	1
1	0	1	0	0	0	0	0	1	0	1
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Окончание таблицы 12.

Таким образом, для сдвига на i-разрядное число необходимо произвести i+1 раз опрос-запись.

Математическая запись алгоритма

(24)

(25)

(26)

(27)

(28)

(29)

(30)

(31)

(32)

где

– , – результаты опроса 0 разряда B;

– , – результаты записи S в зависимости от 0 разряда B;

– – результаты опроса j-го разряда B;

– – результаты записи S в зависимости от j-го разряда B;

– – значение переменной;

– , , – значения масок;

4.2.1.4. АЛГОРИТМ СРАВНЕНИЯ ЦЕЛЫХ ЧИСЕЛ

Пусть в строке K хранятся два целых числа A и B. Для вычисления результата нам потребуется три дополнительных разряда: E – для хранения флага равенства двух чисел и T – для хранения флага, значения, какое число больше, St – для хранения флага о том, что больше числа не нужно проверять.

- E = 1, если A = B;
- T = 0, E = 0, если A > B;
- T = 1, E = 0 если B > A.

Пусть в памяти хранятся следующие числа:

Таблица 13. Содержимое ячеек ассоциативного процессора

A								B								E	T	St
0	1	1	1	0	0	1	1	0	0	0	1	1	1	1	1	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0	0
0	1	1	1	0	0	1	1	1	0	0	1	1	1	1	1	0	0	0
0	1	1	1	0	0	1	1	0	0	0	1	1	1	1	1	0	0	0

Порядок действий при сравнении чисел A и B:

6. Ищем строки, в которых $A_0 = 0$, $B_0 = 1$, St = 0. В найденных строках записываем T = 1, St = 1

Таблица 14. Содержимое ячеек АП после прохождения шага 1.

A								B								E	T	St
0	1	1	1	0	0	1	1	0	0	0	1	1	1	1	1	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0	0

<u>0</u>	1	1	1	0	0	1	1	**<u>1</u>**	0	0	1	1	1	1	1	0	1	1
1	1	1	1	0	0	1	1	0	0	0	1	1	1	1	1	0	0	0

Окончание таблицы 14.

7. Ищем строки, в которых A_0 = 1, B_0 = 0, St = 0. В найденных строках записываем T = 0, St = 1

Таблица 15. Содержимое ячеек АП после прохождения шага 2.

A								B								E	T	St
0	1	1	1	0	0	1	1	0	0	0	1	1	1	1	1	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0	0
0	1	1	1	0	0	1	1	1	0	0	1	1	1	1	1	0	1	1
<u>1</u>	1	1	1	0	0	1	1	**<u>0</u>**	0	0	1	1	1	1	1	0	0	1

8. Повторяем шаги 1, 2 столько раз, сколько разрядов в числах A и B
9. Проводим поиск строк, в которых St = 0. Записываем в данные строки E = 1, St = 1

Таблица 16. Содержимое ячеек АП после прохождения шага 4.

A								B								E	T	St
0	1	1	1	0	0	1	1	0	0	0	1	1	1	1	1	0	0	1
0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0	**<u>0</u>**
0	1	1	1	0	0	1	1	1	0	0	1	1	1	1	1	0	1	1
1	1	1	1	0	0	1	1	0	0	0	1	1	1	1	1	0	0	1

Таблица 17. Итоговое содержимое ячеек ассоциативного процессора

A								B								E	T	St
0	1	1	1	0	0	1	1	0	0	0	1	1	1	1	1	0	0	1

0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	1	0	1
0	1	1	1	0	0	1	1	1	0	0	1	1	1	1	1	0	1	1
1	1	1	1	0	0	1	1	0	0	0	1	1	1	1	1	0	0	1

Окончание таблицы 17.

Математическая форма записи алгоритма:

3.2.1.5. ПОЛУЧЕНИЕ ПРОТИВОПОЛОЖНОГО ЧИСЛА

В цифровых процессорах алгоритм получения противоположного числа – А при наличии какого-то числа А выполняется путем инвертирования всех разрядов данного числа и последующим прибавлением единицы.

В ассоциативном процессоре можно совместить данные операции, получив выигрыш во времени. Для этого понадобится N бит для исходного числа, N бит для результирующего числа и 2 бит C для переноса.

Пусть число, которое необходимо инвертировать, хранится в разрядах A, запись противоположного числа будем производить в разряды O:

Таблица 18. Содержимое ячеек ассоциативного процессора

A								O								C_1	C_2
0	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0

Порядок действий при получении противоположного числа:

10. На первом шаге в разряде переноса всегда хранится 0, в операнде A_8 могут быть следующие числа

Таблица 19. Схема заполнения ячеек O_8 и C_2 в зависимости от a_8.

a_8	C_1	O_8	C_2
0	0	0	1
1	0	1	0

После выполнения шага в таблицу будут занесены следующие значения:

Таблица 20. Содержимое ячеек АП после прохождения шага 1.

A								O								C_1	C_2
0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1
0	1	1	1	0	0	1	1	0	0	0	0	0	0	0	1	0	0
1	1	1	1	0	0	1	1	0	0	0	0	0	0	0	1	0	0

Окончание таблицы 20.

11. Последующие разряды заполняются в зависимости от значений a_i и разряда переноса C_i

Таблица 21. Схема заполнения ячеек O_i и C_{i+1} в зависимости от a_i и и C_i

a_i	C_i	S_i	C_{i+1}
0	0	1	0
0	1	0	1
1	0	0	0
1	1	1	0

После прохождения оставшихся 7 разрядов в таблице будут записаны следующие значения:

Таблица 22. Итоговое содержимое ячеек АП.

A								O								C_1	C_2
0	1	1	1	0	0	0	0	1	1	0	0	1	0	0	0	0	0
0	0	0	0	0	1	1	0	1	1	1	1	1	0	1	0	0	0
0	1	1	1	0	0	1	1	1	0	0	0	1	1	0	1	0	0
1	1	1	1	0	0	1	1	0	0	0	0	1	1	0	1	0	0

3.2.2. АЛГОРИТМЫ ДЛЯ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ

3.2.2.1. АЛГОРИТМ СЛОЖЕНИЯ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Допустим, что числа для сложения хранятся в ассоциативной памяти в формате IEEE754. Тогда для их сложения необходимо сделать следующее:

1. Вычислить разность между экспонентами.
2. Выполнить сдвиг мантиссы меньшего числа вправо.
3. Выполнить сложение 23-битовых мантисс.
4. Нормализовать результат сложения.

Все операции должны выполняться в базисе микроопераций «опрос-считывание» и «опрос-запись», поэтому алгоритм сложения чисел с плавающей точкой необходимо преобразовать.

Для сложения двух чисел с плавающей точкой A и B необходимо произвести следующую последовательность действий:

1. Сравниваем значения порядков для обоих чисел (E_A и E_B) и находим число с меньшей экспонентой. Выставляем флаг NUM в 0, если меньше первое число и в 1, если второе. Если порядки чисел равны, то ставим 1
2. Находим разность порядков двух чисел

Записываем D в эту же строку АП. Разность порядков также будет восьмибитным числом.

В формате IEEE 754 к экспонентам (порядкам) чисел прибавляется 127 для удобства сравнивания и вычисления. Таким образом, область значений экспоненты – диапазон [-127_{10} ; 128_{10}] включительно (-127 = 0b00000000, 128 = 0b11111111). На практике, однако, эти числа используются для хранения 0, бесконечности и NaN.

Соответственно:

где – настоящие значения экспоненты.

В целом, при вычитании из одного числа другого, разница в 127 взаимоуничтожится, однако в случае, если A меньше B, мы получим отрицательную разность в дополнительном коде, и для того, чтобы получить правильное значение для последующего переноса мантиссы, нужно его инвертировать.

3. Проверяем *D*. Если *NUM* = 0, то *D* должно быть меньше 0 (первый бит *D* = 1). Необходимо умножить число на -1 (инвертировать все биты числа и прибавить 1)
4. Мантиссы обеих чисел необходимо сдвинуть на 1 разряд вправо, чтобы не потерялась неявная единица. При этом в сдвинутый разряд записываем 1, порядки сдвинутых чисел увеличиваем на 1
5. Проверяем получившееся число *D*. Если , то сдвигаем мантиссу меньшего числа (получаем из NUM) на *D* разрядов вправо, добавляя при этом единицу, которая в стандарте IEEE 754 хранится неявно. Записываем преобразованную мантиссу в отдельные 23 ячейки, чтобы не потерять первоначальные операнды.

Если *D* > 23, то сдвиг будет слишком большим: второе число много меньше первого, чтобы повлиять на результат сложения. Результатом суммы будет первое число.

6. Сдвигаем числа еще на разряд вправо для приведения к денормализованной форме. В сдвинутый разряд записываем 0, порядки сдвинутых чисел увеличиваем на 1;
7. Складываем мантиссы;
8. Проверяем полученный результат. Если первый бит полученной мантиссы равен 1 и производилось вычитание/сложение отрицательных чисел, то полученное число также отрицательное, нужно его инвертировать и прибавить 1, а также записать 1 в знаковый разряд полученной суммы;
9. Вычисляем порядок полученной суммы и нормализуем число.

Для удобства чтения покажем получившийся алгоритм в виде нескольких блок-схем:

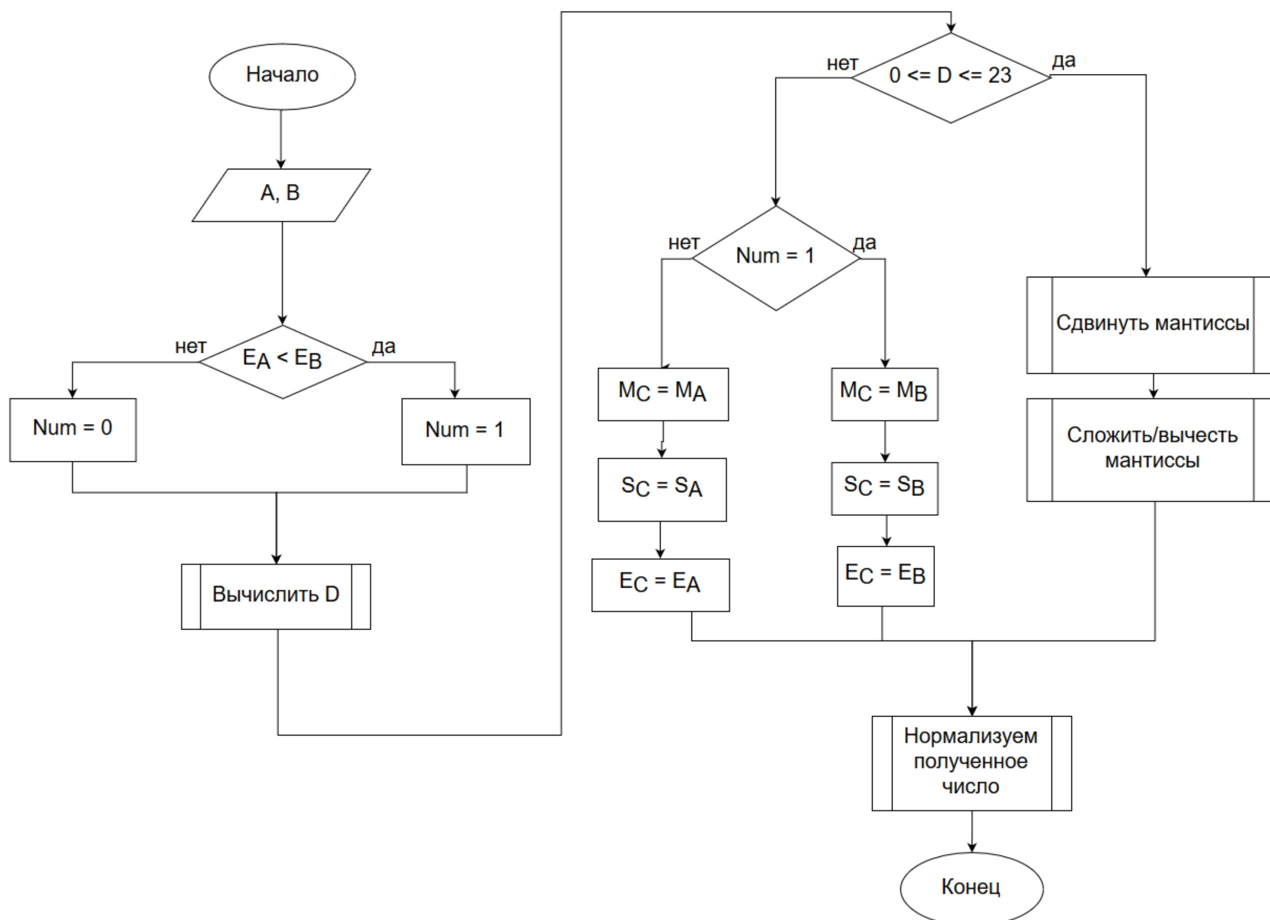

Рисунок 8 – Алгоритм сложения чисел с плавающей точкой

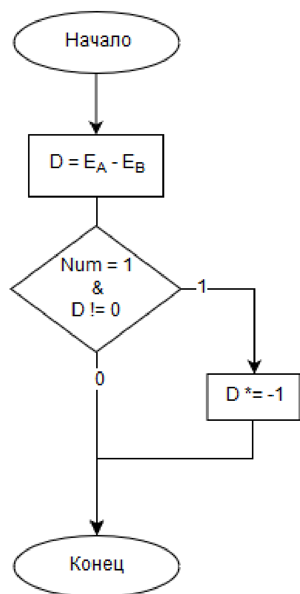

Рисунок 9 – Функция «Вычислить D»

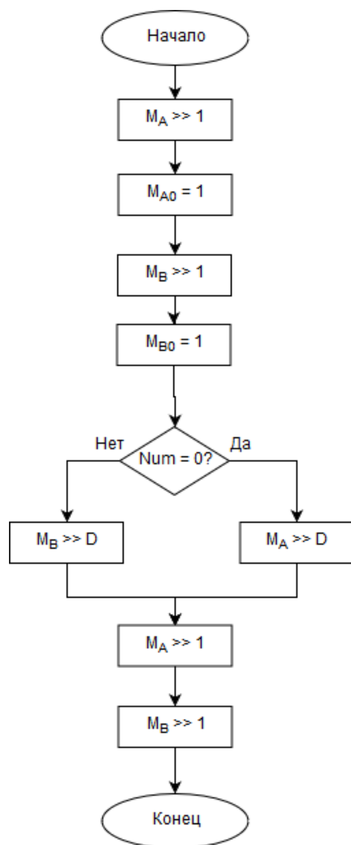

Рисунок 10 – Функция «Сдвинуть мантиссы»

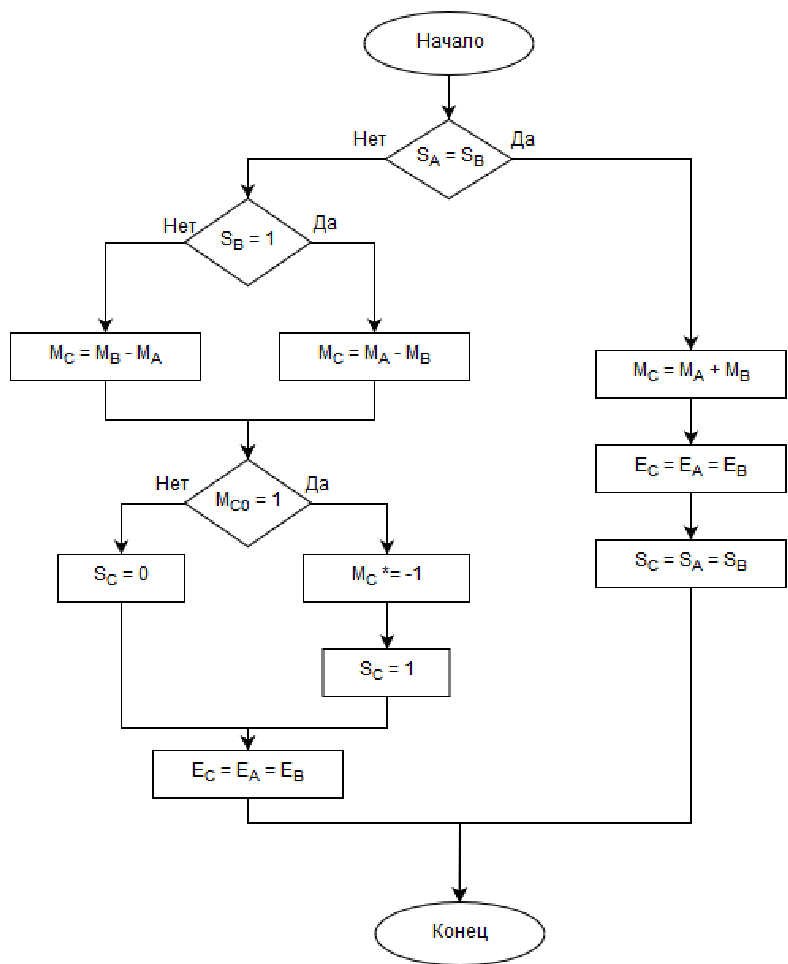

Рисунок 11 – Функция «Сложить мантиссы»

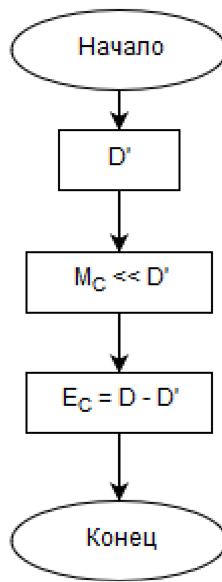

Рисунок 12 – Функция «Нормализовать результат»

4. РАЗРАБОТКА ДЕМОНСТРАЦИОННОГО ПРИЛОЖЕНИЯ

4.1. АКТУАЛЬНОСТЬ ДЕМОНСТРАЦИОННОГО ПРИЛОЖЕНИЯ

Алгоритмы, описанные в предыдущем разделе не имеют аппаратной реализации, следовательно, полученные значения не могут быть проверены на практике.

Полученные алгоритмы записаны как в табличной, так и в математической форме. Однако, данная форма представления не до конца отражает порядок опроса, чтения и записи в ячейки отдельных бит информации. Для наглядного отображения алгоритмов, необходимого для дальнейшей работы с ними, было решено создать визуальное приложение, показывающее ход работы алгоритма, пояснения к каждому шагу и значения, хранящиеся в ячейках на каждый момент времени.

Данное приложение поможет при дальнейшем проектировании рабочего прототипа на языках аппаратного описания аппаратуры, так как в них необходимо задать именно побитовые входные и выходные значения для каждого модуля процессора в каждый момент времени.

4.2. ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Демонстрационное приложение должно реализовывать следующие функции:

- наглядное отображение всех задействованных разрядов строки в данном алгоритме;
- последовательное пошаговое прохождение алгоритма с описанием протекающего шага и выделением разрядов строки, над которыми происходит преобразование;
- возможность поставить прохождение алгоритма на паузу;

- возможность выбрать один из представленных алгоритмов;
- для чисел с плавающей точкой: отображение чисел в формате IEEE 754 и вычисление суммы чисел с плавающей точкой, так как работа с данным алгоритмом затруднена в силу его неинтуитивного хранения информации;
- для чисел с плавающей точкой: проверка результата сложения чисел с плавающей точкой.

4.3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

Демонстрационное приложение было реализовано в программной среде Unity. Данное окружение позволяет создавать достаточно быстрые и понятные приложения с использованием пользовательского интерфейса и последовательного прохождения операций в виде корутин.

Скрипты для работы приложения были написаны на языке C#. Шрифты и кнопки в интерфейсе приложения были взяты из стандартных пресетов.

4.4. РАБОТА ПРИЛОЖЕНИЯ

Скриншоты работы приложения представлены на рисунках 4.1-4.5.

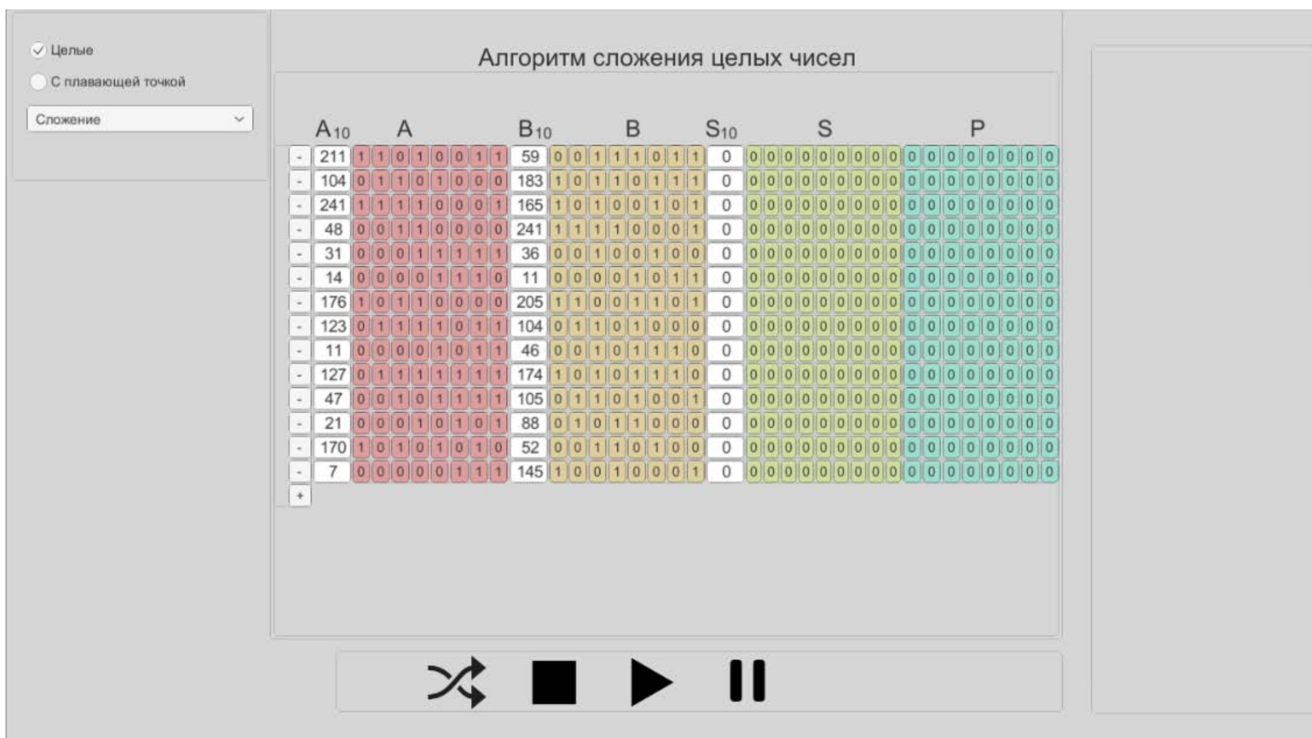

Рисунок 13 – Начало работы алгоритма «Сложение целых чисел»

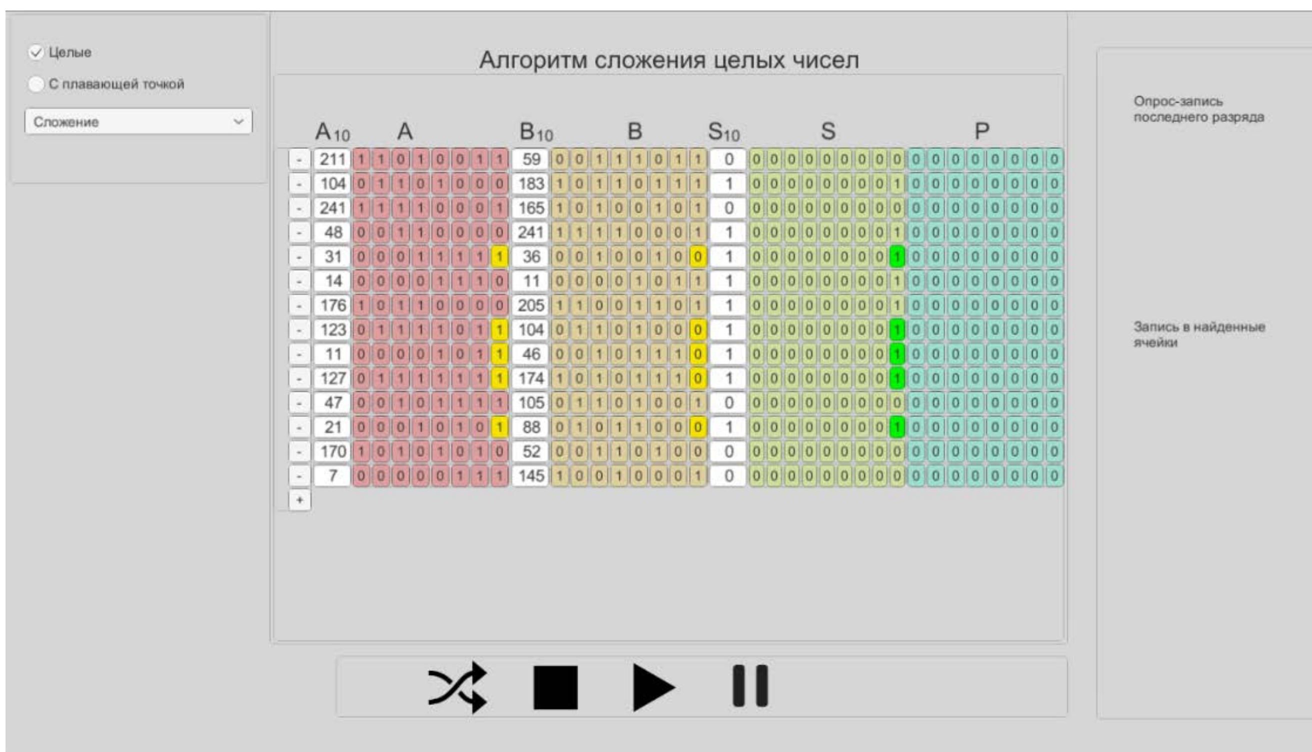

Рисунок 14 – Прохождение алгоритма «Сложение целых чисел», пауза

Рисунок 15 – Результат прохождения алгоритма «Сложение целых чисел»

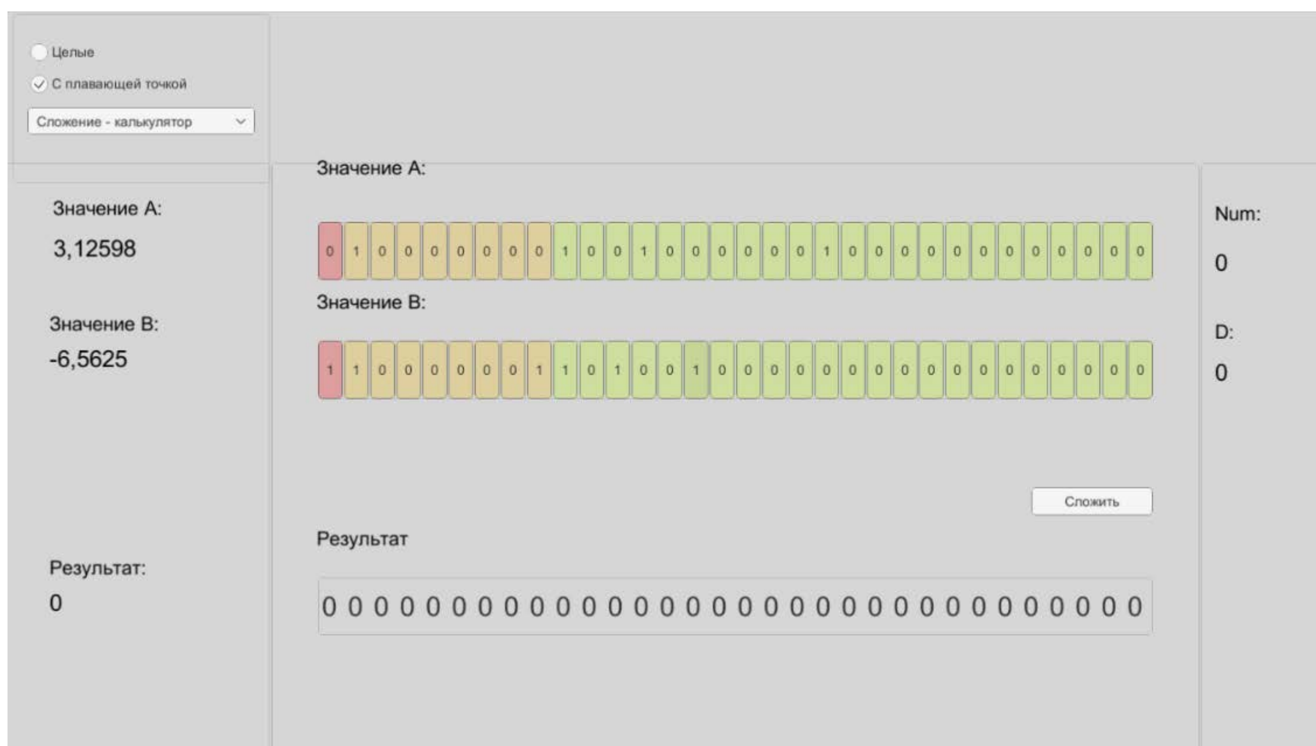

Рисунок 16 – Режим работы калькулятора чисел с плавающей точкой

Рисунок 17 – Результат работы калькулятора

5. АНАЛИЗ БЫСТРОДЕЙСТВИЯ СОСТАВЛЕННЫХ АЛГОРИТМОВ

5.1. ВЫБОР МЕТРИКИ БЫСТРОДЕЙСТВИЯ

Для данных алгоритмов необходимо вычислить их быстродействие относительно последовательных процессоров и графических процессоров. Следовательно, нам необходимо вычислить для каждого алгоритма количество циклов за инструкцию (CPI).

В описании алгоритмов были приведены операции типа «опрос-запись», однако при оценке быстродействия некорректно рассматривать их как одну операцию, их требуется разделить на «опрос-чтение» и «опрос-запись». Соответственно, одна операция «опрос-запись» на самом деле занимает 2 такта процессора.

5.2. АНАЛИЗ БЫСТРОДЕЙСТВИЯ АЛГОРИТМОВ

5.2.1. СЛОЖЕНИЕ ЦЕЛЫХ ЧИСЕЛ

Согласно документации Intel, сложение целых чисел происходит занимает 1 CPI за счет использования параллельных сумматоров. Более того, за счет частичного распараллеливания, эта величина иногда достигает 0.5 CPI.

Вычисление сложения на нашем алгоритме занимает 3 опроса-записи для последнего разряда и 7 опросов-записи для остальных. Если принять, что мы складываем 32-разрядные числа, наш алгоритм занимает 440 циклов на операцию сложения. Однако, так как мы одновременно проводим вычисления на массиве чисел, выигрыш в производительности начинается на вычислении суммы 880 целых чисел.

На рис. 5.1 показана зависимость количества тактов операции в зависимости от числа складываемых чисел: зеленым – на ассоциативном процессоре, синим – на стандартной архитектуре

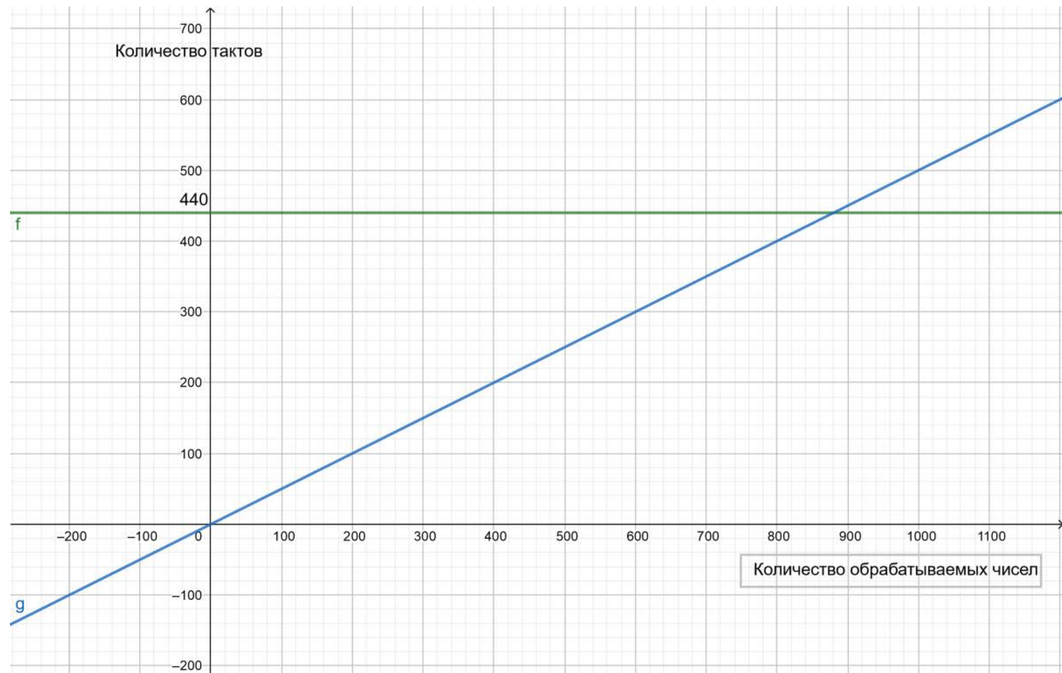

Рисунок 18 – Зависимость количества тактов на операцию от количества обрабатываемых чисел

5.2.2 СДВИГ

Сдвиг также занимает 0.5-1 такт за операцию на процессорах стандартной архитектуры.

В ассоциативном процессоре количество тактов зависит от разрядности чисел. 2^N-разрядные числа будут сдвигаться за (N+1) такт. Соответственно, сдвиг 32-разрядного целого числа будет выполнен за 6 тактов.

5.2.3. СЛОЖЕНИЕ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Распишем число тактов для каждой операции, необходимой для сложения чисел с плавающей точкой:

Таблица 23. Количество тактов для каждой операции алгоритма сложения чисел с плавающей точкой

Операция	**Количество тактов**
Сравнение двух восьмиразрядных чисел	34 такта (17 «опрос-запись»)
Взятие противоположных чисел для отрицательных восьмиразрядных чисел	16 тактов (8 ОЗ)
Сдвиг 2 операнд на 1 разряд вправо для получения неявной единицы	2х 2 такта
Сдвиг мантиссы на разность экспонент (вплоть до 5 разрядов)	22 такта
Сложение 23-разрядных мантисс	314 тактов
Вычитание мантисс, у которых разный знак S	314
Нормализация мантисс (обратный сдвиг)	22 такта
Сумма:	**726 тактов**

Таким образом, для сложения чисел с плавающей точкой требуется 726 тактов ассоциативного процессора. Согласно документации, для выполнения этой операции на процессорах Intel требуется от 1 до 3 тактов ЦП. При использовании 128-битных векторных операций SSE можно одновременно оперировать с 4 числами с плавающей точкой одинарной точности (float). Соответственно, будем считать, что за 1 такт стандартный процессор вычисляет 2 суммы чисел с ПТ.

Приняв во внимание вышеперечисленное, получаем, что ассоциативный процессор целесообразно использовать для массивов чисел не менее чем из 1500 элементов.

5.3. ВЫВОДЫ

Ассоциативный процессор неэффективно использовать для арифметических функций в случаях, если происходит обработка небольшого количества чисел. Параллельная обработка разрядов по значению в данных случаях только увеличивает время обработки. Однако время обработки не изменяется с увеличением количества обрабатываемых чисел, что делает его использование целесообразным при вычислении больших массивов данных.

Данный вывод основывается только на измерении количества тактов на операцию и не учитывает время выполнения операции при физической реализации данных алгоритмов.

6. ЗАКЛЮЧЕНИЕ

В представленной работе были разобраны основные архитектуры цифровых процессоров и методы решения арифметических задач на существующих процессорах. Изучена архитектура ассоциативных процессоров и текущее состояние отрасли изучения и производства ассоциативных процессоров.

Разработаны алгоритмы вычисления сложения чисел с плавающей точкой и некоторые арифметический функции с целыми числами для ассоциативных процессоров.

Разработано приложения для демонстрации алгоритмов в наглядной форме. Для разработки приложения были использованы среды Unity и Visual Studio 2017.

Проанализирована целесообразность использования ассоциативных процессоров для работы с целыми числами и числами с плавающей точкой. Ассоциативный процессор для вычисления арифметических функций целесообразно использовать при обработке больших массивов чисел. Для дальнейшего исследования требуется создать рабочий прототип и произвести измерение быстродействия на реальной схеме.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Moore's Law is dead, says NVidia's CEO. – https://www.cnet.com/news/moores-law-is-dead-nvidias-ceo-jensen-huang-says-at-ces-2019. Дата обращения: 19.01.2019.
2. Харрис Д.М., Цифровая схемотехника и архитектура компьютера / Д.М. Харрис, С.Л. Харрис – Издательство Morgan Kaufman © English Edition 2013 – 1662 с.
3. Н.П. Вашкевич, Основы арифметики цифровых процессоров: учебное пособие / Вашкевич Н.П. – Пенза: Издательство ПГУ, 2010 – 162 с
4. Кохонен Т., Ассоциативные запоминающие устройства:– Мир, 1982, 384 с.
5. Что нужно знать про арифметику с плавающей запятой? [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/112953/, свободный. – Заглавие с экрана.
6. Goldberg D. What Every Computer Scientist Should Know About Floating-Point Arithmetic / D. Goldberg – XeroxPaloAltoResearchCenter, 1991 – 94 с.
7. Танненбаум Э., Архитектура компьютера. – СПб.: Питер, 2019. — 816 с.
8. Собственная платформа. Часть 0.1 Теория. Немного о процессорах [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/316520/, свободный. – Заглавие с экрана.
9. Оптимизация ресурсоёмких вычислительных задач на графическом процессоре / сост. А.А. Сытник, С.П. Ивженко, И.В. Гвоздюк. – Известия Самарского научного центра РАН, 2016 – 7 с.
10. Параллельные вычисления общего назначения на графических процессорах: учебное пособие / К. А. Некрасов, С. И. Поташников, А. С.

Боярченков, А. Я. Купряжкин. — УрГУ : Изд-во Урал ьского университета, 2016.— 104 с.

11. Что такое кэш процессора и как он работает? [Электронный ресурс] – Режим доступа: https://compress.ru/artic le.aspx?id=23541, свободный – Заглавие с экрана.
12. Мюллер С., Модернизация и ремонт ПК – М.: Вильямс, 2007. —241 с.
13. Н.А. Караван, Системный анализ формирования признаков для поиска информации в ассоциативных запоминающих устройствах / Н.А. Караван, Я.В. Корпань, Д.А. Лукашенко, К.С. Рудаков – УрГУ : Изд-во Уральского университета, 2011.— 8 с.
14. IEEE 754 - стандарт двоичной арифметики с плавающей точкой [Электронный ресурс] – Режим доступа: http://www.softelectro.ru/ieee754.html, свободный – Заглавие с экрана.
15. Imani M. Resistive Content Addressable Memory for ConfigurableApproximation [Electronic resource] / M. Imani // IEEE –2018г. –International Symposium on Quality Electronic Design – Режим доступа: https://pdfs.semanticscholar.org/d134/975e44c47786278865eda04070f5e963a7a2.pdf – Заглавие с экрана.
16. Imani M., CAP: Configurable resistive associative processor for near-data computing [Electronic resource] / M. Imani // IEEE –2017г. – 2017 18th International Symposium on Quality Electronic Design – Режим доступа: https://www.academia.edu/32985302/CAP_Configurable_Resistive_Associative_Processor_for_Near-Data_Computing – Заглавие с экрана.
17. Yavitz L., Computer Architecture with Associative Processor Replacing Last-Level Cache and SIMD Accelerator [Electronic resource] / L. Yavitz // IEEE Journals and Magazines – 2013г. – Volume: 64 Issue: 2 – Режим доступа: https://ieeexplore.ieee.org/abstract/document/6671575/ – Заглавие с экрана.

18. Стоимость операций в тактах ЦП [] – Режим доступа: https://habr.com/ru/company/otus/blog/343566/, свободный – Заглавие с экрана.
19. Репозиторий с проектом демонстрационного приложения: https://bitbucket.org/danila_zi/assosiative/src/master/

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД МОДУЛЕЙ ДЕМОНСТРАЦИОННОГО ПРИЛОЖЕНИЯ

.Листинг 1 – исходный код файла «Cell.cs»

```
public class Cell : MonoBehaviour
{
    public int Value;
    public Text BtnText;

    void Start()
    {
        BtnText = GetComponentInChildren<Text>();
        BtnText.text = Value.ToString();
    }

    public void ChangeValue()
    {
        if (Value == 0)
        {
            Value = 1;
        }
        else if (Value == 1)
        {
            Value = 0;
        }
        BtnText.text = Value.ToString();
    }

    public void SetValue(int value)
    {
        Value = value;
        BtnText.text = value.ToString();
    }
}
```

Листинг 2 – исходный код файла «IntegerAddAlgo.cs»

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using UnityEngine;
using UnityEngine.Events;
using UnityEngine.UI;
using Random = System.Random;


public class IntegerAddAlgo : BaseAlgo
{
    private Color AColor = new Color(0.9056604f, 0.6450694f, 0.6450694f);
    private Color BColor = new Color(0.9058824f, 0.8447574f, 0.6431373f);
    private Color SColor = new Color(0.8420867f, 0.9058824f, 0.6431373f);
    private Color PColor = new Color(0.6431373f, 0.9058824f, 0.844094f);
    public List<CellArray> AList;
    public List<InputField> ADecInput;
    public List<string> AString;
    public List<CellArray> BList;
    public List<InputField> BDecInput;
    public List<string> BString;
    public List<CellArray> SList;
    public List<InputField> SDecOutput;
    public List<string> SString;
    public CellArray SingleArray9;
    public List<CellArray> PList;
    public Transform APlace, BPlace, ADecPlace, BDecPlace, SPlace, SDecPlace, PPlace;
    private int lineMaxIndex = 0;

    // Start is called before the first frame update
    void Start()
    {
        SingleArray8.InitArray(8);
        SingleArray9.InitArray(9);
```

```
        InitButtons();
    }

    public override void InitButtons()
    {
        for (int i = 0; i < 10; i++)
        {
            var tempNum = i;
            AList.Add(Instantiate(SingleArray8, APlace));
            foreach (var cell in AList.Last().Element)
            {
                cell.GetComponent<Button>().onClick.AddListener(delegate
                    {
                        RenewDecimal(tempNum, ADecInput, AList, AString);
                    });
            }
            BList.Add(Instantiate(SingleArray8, BPlace));
            foreach (var cell in BList.Last().Element)
            {
                cell.GetComponent<Button>().onClick.AddListener(delegate
                {
                    RenewDecimal(tempNum, BDecInput, BList, BString);
                });
            }
            SList.Add(Instantiate(SingleArray9, SPlace));
            PList.Add(Instantiate(SingleArray8, PPlace));
            AString.Add("00000000");
            BString.Add("00000000");
            SString.Add("00000000");
            ADecInput.Add(Instantiate(Input, ADecPlace));
            ADecInput.Last().onEndEdit.AddListener(delegate   {RenewRow(tempNum,   ADecInput,
AList, AString);});
            BDecInput.Add(Instantiate(Input, BDecPlace));
            BDecInput.Last().onEndEdit.AddListener(delegate  {  RenewRow(tempNum,  BDecInput,
BList, BString); });
            SDecOutput.Add(Instantiate(Input, SDecPlace));
            ADecInput[i].text = "0";
            BDecInput[i].text = "0";
            SDecOutput[i].text = "0";
```

```
        DeleteButtons.Add(Instantiate(DeleteButton, DeletePlace));
        DeleteButtons[i].onClick.AddListener(delegate {DeleteRow(tempNum);});
        lineMaxIndex++;
    }
    CreateAddButton();
    ColorCells(AList, AColor);
    ColorCells(BList, BColor);
    ColorCells(SList, SColor);
    ColorCells(PList, PColor);
}

private void ColorCells(List<CellArray> cells, Color color)
{
    foreach (var row in cells)
    {
        foreach (var cell in row.Element)
        {
            cell.GetComponent<Image>().color = color;
        }
    }
}

public override void AddNewRow()
{
    if (AList.Count < 18)
    {
        lineMaxIndex++;
        AList.Add(Instantiate(SingleArray8, APlace));
        foreach (var cell in AList.Last().Element)
        {
            cell.GetComponent<Button>().onClick.AddListener(delegate
            {
                RenewDecimal(lineMaxIndex - 1, ADecInput, AList, AString);
            });
        }
        BList.Add(Instantiate(SingleArray8, BPlace));
        foreach (var cell in BList.Last().Element)
        {
```

```
                cell.GetComponent<Button>().onClick.AddListener(delegate
                {
                    RenewDecimal(lineMaxIndex - 1, BDecInput, BList, BString);
                });
            }
            SList.Add(Instantiate(SingleArray9, SPlace));
            PList.Add(Instantiate(SingleArray8, PPlace));
            //PList[i] = Instantiate(SingleArray8, PPlace);

            AString.Add("00000000");
            BString.Add("00000000");
            SString.Add("00000000");

            ADecInput.Add(Instantiate(Input, ADecPlace));
            ADecInput.Last().onEndEdit.AddListener(delegate  {  RenewRow(lineMaxIndex  -  1,
ADecInput, AList, AString); });
            BDecInput.Add(Instantiate(Input, BDecPlace));
            BDecInput.Last().onEndEdit.AddListener(delegate  {  RenewRow(lineMaxIndex  -  1,
BDecInput, BList, BString); });
            SDecOutput.Add(Instantiate(Input, SDecPlace));

            ADecInput.Last().text = "0";
            BDecInput.Last().text = "0";
            SDecOutput.Last().text = "0";

            Destroy(DeletePlace.GetChild(DeleteButtons.Count - 1).gameObject);
            DeleteButtons.RemoveAt(DeleteButtons.Count - 1);
            DeleteButtons.Add(Instantiate(DeleteButton, DeletePlace));
            DeleteButtons.Last().onClick.AddListener(delegate { DeleteRow(lineMaxIndex - 1);
});
            CreateAddButton();
            ColorCells(AList, AColor);
            ColorCells(BList, BColor);
            ColorCells(SList, SColor);
            ColorCells(PList, PColor);
        }

    }
```

```
public override void DeleteRow(int i)
{
    //Debug.Log("i " + i );
    AString.RemoveAt(i);
    BString.RemoveAt(i);
    SString.RemoveAt(i);

    lineMaxIndex--;
    ClearMatrix();

    for (int j = 0; j < lineMaxIndex; j++)
    {
        var index = j;

        AList.Add(Instantiate(SingleArray8, APlace));
        SetCellArray(AList[index], AString[index]);
        foreach (var cell in AList.Last().Element)
        {
            cell.GetComponent<Button>().onClick.AddListener(delegate
            {
                RenewDecimal(index, ADecInput, AList, AString);
            });
        }
        BList.Add(Instantiate(SingleArray8, BPlace));
        SetCellArray(BList[index], BString[index]);
        foreach (var cell in BList.Last().Element)
        {
            cell.GetComponent<Button>().onClick.AddListener(delegate
            {
                RenewDecimal(index, BDecInput, BList, BString);
            });
        }
        SList.Add(Instantiate(SingleArray9, SPlace));
        PList.Add(Instantiate(SingleArray8, PPlace));



        ADecInput.Add(Instantiate(Input, ADecPlace));
        SetInputDecimal(ADecInput[index], AString[index]);
```

```
            ADecInput.Last().onEndEdit.AddListener(delegate   {   RenewRow(index,   ADecInput,
AList, AString); });

            BDecInput.Add(Instantiate(Input, BDecPlace));
            SetInputDecimal(BDecInput[index], BString[index]);
            BDecInput.Last().onEndEdit.AddListener(delegate   {   RenewRow(index,   BDecInput,
BList, BString); });

            SDecOutput.Add(Instantiate(Input, SDecPlace));
            SDecOutput.Last().text = "0";

            DeleteButtons.Add(Instantiate(DeleteButton, DeletePlace));
            DeleteButtons[j].onClick.AddListener(delegate { DeleteRow(index); });

        }
        CreateAddButton();
        ColorCells(AList, AColor);
        ColorCells(BList, BColor);
        ColorCells(SList, SColor);
        ColorCells(PList, PColor);
    }

    public void ClearMatrix()
    {
        foreach (Transform child in APlace)
        {
            Destroy(child.gameObject);
        }
        foreach (Transform child in BPlace)
        {
            Destroy(child.gameObject);
        }
        foreach (Transform child in PPlace)
        {
            Destroy(child.gameObject);
        }
        foreach (Transform child in SPlace)
        {
            Destroy(child.gameObject);
```

```
    }
    foreach (Transform child in ADecPlace)
    {
        Destroy(child.gameObject);
    }
    foreach (Transform child in BDecPlace)
    {
        Destroy(child.gameObject);
    }
    foreach (Transform child in SDecPlace)
    {
        Destroy(child.gameObject);
    }
    foreach (Transform child in DeletePlace)
    {
        Destroy(child.gameObject);
    }
    ADecInput.Clear();
    BDecInput.Clear();
    SDecOutput.Clear();
    AList.Clear();
    BList.Clear();
    PList.Clear();
    SList.Clear();
    DeleteButtons.Clear();
}

public void DestroyObjectsAtRow(int rowNumber)
{
    Destroy(APlace.GetChild(rowNumber).gameObject);
    Destroy(BPlace.GetChild(rowNumber).gameObject);
    Destroy(SPlace.GetChild(rowNumber).gameObject);
    Destroy(PPlace.GetChild(rowNumber).gameObject);
    Destroy(ADecPlace.GetChild(rowNumber).gameObject);
    Destroy(BDecPlace.GetChild(rowNumber).gameObject);
    Destroy(SDecPlace.GetChild(rowNumber).gameObject);
    Destroy(DeletePlace.GetChild(rowNumber).gameObject);
}
```

```
private int curStep = 0;
private int aValue, bValue, pValue;
private int curColumnNumber;
private bool onPause;
private bool isWriting;
private bool PNotSet = false;

public override void StartAlgo()
{
    if (onPause) onPause = false;
    else
        AddOperands();
}
public void AddOperands()
{
    curColumnNumber = SingleArray8.Element.Length - 1;
    curStep = 0;
    StartCoroutine(Adding());
    //разделить на опрос и запись
}

List<int> rows = new List<int>();

public IEnumerator Adding()
{
    while (curColumnNumber >= 0)
    {
        if (!onPause)
        {
            //curStep = 0;
            if (curColumnNumber == 7)
            {
                while (curStep < 3)
                {
                    if (!isWriting)
                    {
                        Description.text = "Опрос-запись последнего разряда";
                        //rows.Clear();
```

```
                    if (rows != null && rows.Count != 0)
                    {
                        //Debug.Log("!!!");
                        HighlightCellsInColumn(AList, rows, 7, AColor);
                        HighlightCellsInColumn(BList, rows, 7, BColor);
                        HighlightCellsInColumn(SList, rows, 8, SColor);
                        HighlightCellsInColumn(PList, rows, 7, PColor);
                    }

                    SetABValues();
                    Operation.text = String.Format("Опрос ячеек, в которых a = {0},
b = {1}", aValue, bValue);
                    rows = GetMatchingRows(aValue, bValue, 7);
                    HighlightCellsInColumn(AList, rows, 7, Color.yellow);
                    HighlightCellsInColumn(BList, rows, 7, Color.yellow);
                    isWriting = true;
                    if (onPause)
                        break;
                    yield return new WaitForSeconds(1f);
                }
                if (isWriting)
                {
                    Operation.text = "Запись в найденные ячейки";
                    if (aValue == 1 && bValue == 1)
                    {
                        HighlightCellsInColumn(PList, rows, 7, Color.green);
                        SetValueInCells(PList, rows, 7, 1);
                    }
                    else
                    {
                        HighlightCellsInColumn(SList, rows, 8, Color.green);
                        SetValueInCells(SList, rows, 8, 1);
                    }

                    if (onPause)
                        break;
                    yield return new WaitForSeconds(1f);
                    isWriting = false;
```

```
                        curStep++;
                        if (curStep == 3)
                        {
                            curColumnNumber--;
                            curStep = 0;
                            PNotSet = true;
                            break;
                        }
                    }

                }
            }
            else
            {
                Debug.Log("CurColumn: " + curColumnNumber);
                Description.text    =    String.Format("Опрос-запись    {0}    разряда",
curColumnNumber);
                while (curStep < 7)
                {
                    if (!isWriting)
                    {
                        if (rows != null && PNotSet)
                        {
                            HighlightCellsInColumn(AList, rows, 7, AColor);
                            HighlightCellsInColumn(BList, rows, 7, BColor);
                            HighlightCellsInColumn(SList, rows, 8, SColor);
                            HighlightCellsInColumn(PList, rows, 7, PColor);
                        }
                        else
                        {
                            HighlightCellsInColumn(AList,     rows,     curColumnNumber,
AColor);
                            HighlightCellsInColumn(BList,     rows,     curColumnNumber,
BColor);
                            HighlightCellsInColumn(PList,  rows,  curColumnNumber  +  1,
PColor);
                            HighlightCellsInColumn(PList,     rows,     curColumnNumber,
PColor);
```

```
                    HighlightCellsInColumn(SList,  rows,  curColumnNumber  +  1,
SColor);
                }



                SetABPValues();
                PNotSet = false;
                Operation.text = String.Format("Опрос ячеек, в которых a = {0},
b = {1}, p = {2}", aValue, bValue, pValue);
                rows = GetMatchingRows(aValue, bValue, pValue, curColumnNumber);
                HighlightCellsInColumn(AList,      rows,      curColumnNumber,
Color.yellow);
                HighlightCellsInColumn(BList,      rows,      curColumnNumber,
Color.yellow);
                HighlightCellsInColumn(PList,   rows,   curColumnNumber   +   1,
Color.yellow);
                isWriting = true;
                if (onPause)
                    break;
                yield return new WaitForSeconds(1f);
            }
            if (isWriting)
            {
                Operation.text = "Запись в найденные ячейки";
                if (curStep < 3)
                {
                    HighlightCellsInColumn(SList,  rows,  curColumnNumber  +  1,
Color.green);

                    SetValueInCells(SList, rows, curColumnNumber + 1, 1);

                }
                else if (curStep < 6)
                {
                    HighlightCellsInColumn(PList,    rows,    curColumnNumber,
Color.green);

                    SetValueInCells(PList, rows, curColumnNumber, 1);

                }
                else
```

```
                        {
                            HighlightCellsInColumn(SList,  rows,  curColumnNumber  +  1,
Color.green);
                            SetValueInCells(PList, rows, curColumnNumber, 1);
                            HighlightCellsInColumn(PList,     rows,     curColumnNumber,
Color.green);
                            SetValueInCells(SList, rows, curColumnNumber + 1, 1);
                        }


                        if (onPause)
                            break;
                        yield return new WaitForSeconds(1f);
                        curStep++;
                        isWriting = false;
                        if (curStep == 7)
                        {
                            curColumnNumber--;
                            curStep = 0;
                            break;
                        }

                    }



                    //HighlightCellsInColumn(PList, rows, 0, Color.white);
                    //HighlightCellsInColumn(SList, rows, 0, Color.white);
                }
            }
        }
        yield return null;

    }

    //опрашиваем последний разряд PList и заполняем 0й в SList
    rows.Clear();
    Description.text = "Заключительный опрос единиц в ячейках переноса";
    Operation.text = "Опрос ячеек, в которых p = 1";
```

```
        rows = GetMatchingRows(1, 0);
        HighlightCellsInColumn(PList, rows, 0, Color.yellow);
        yield return new WaitForSeconds(0.5f);

        Operation.text = "Запись в 0й разряд ячеек суммы";
        HighlightCellsInColumn(SList, rows, 0, Color.green);
        SetValueInCells(SList, rows, 0, 1);
        yield return new WaitForSeconds(0.5f);

        ColorCells(AList, AColor);
        ColorCells(BList, BColor);
        ColorCells(SList, SColor);
        ColorCells(PList, PColor);
    }

    private void SetABValues()
    {
        switch (curStep)
        {
            case 0:
                aValue = 0;
                bValue = 1;
                break;
            case 1:
                aValue = 1;
                bValue = 0;
                break;
            case 2:
                aValue = 1;
                bValue = 1;
                break;
        }
    }

    private void SetABPValues()
    {
        switch (curStep)
        {
            case 0:
```

```
            aValue = 0;
            bValue = 0;
            pValue = 1;
            break;
        case 1:
            aValue = 0;
            bValue = 1;
            pValue = 0;
            break;
        case 2:
            aValue = 1;
            bValue = 0;
            pValue = 0;
            break;
        case 3:
            aValue = 1;
            bValue = 0;
            pValue = 1;
            break;
        case 4:
            aValue = 0;
            bValue = 1;
            pValue = 1;
            break;
        case 5:
            aValue = 1;
            bValue = 1;
            pValue = 0;
            break;
        case 6:
            aValue = 1;
            bValue = 1;
            pValue = 1;
            break;
    }
}

private List<int> GetMatchingRows(int aValue, int bValue, int columnNumber)
{
```

```
        var result = new List<int>();
        for (int i = 0; i < AList.Count; i++)
        {
            if (AList[i].Element[columnNumber].Value == aValue
                && BList[i].Element[columnNumber].Value == bValue)
                result.Add(i);
        }
        return result;
    }

    private List<int> GetMatchingRows(int aValue, int bValue, int pValue, int columnNumber)
    {
        var result = new List<int>();
        for (int i = 0; i < AList.Count; i++)
        {
            if (AList[i].Element[columnNumber].Value == aValue
                && BList[i].Element[columnNumber].Value == bValue
                && PList[i].Element[columnNumber + 1].Value == pValue)
                result.Add(i);
        }
        return result;
    }

    private List<int> GetMatchingRows(int pValue, int columnNumber)
    {
        var result = new List<int>();
        for (int i = 0; i < AList.Count; i++)
        {
            if (PList[i].Element[columnNumber].Value == pValue)
                result.Add(i);
        }
        return result;
    }

    private   void   SetValueInCells(List<CellArray>   cells,   List<int>   rowNumbers,   int
columnNumber, int value)
    {
        for (int i = 0; i < rowNumbers.Count; i++)
        {
```

```
            cells[rowNumbers[i]].Element[columnNumber].SetValue(value);

            if (cells == SList)
            {
                SString[rowNumbers[i]] = GetStringByArray(cells[rowNumbers[i]]);
                SetInputDecimal(SDecOutput[rowNumbers[i]], SString[rowNumbers[i]]);
            }
        }
    }

    public override void HighlightCellsInColumn(List<CellArray> cells, List<int> rowNumbers,
int columnNumber, Color color)
    {
        if (rowNumbers != null)
        {
            for (int i = 0; i < rowNumbers.Count; i++)
            {
                cells[rowNumbers[i]].Element[columnNumber].GetComponent<Image>().color      =
color;
            }
        }

        //base.HighlightCellsInColumn(rowNumbers, columnNumber, color);
    }

    public override void PauseAlgo()
    {
        onPause = true;
    }

    public override void NextStep()
    {
        if (!isWriting)
        {

        }
        else
        {
```

```
    }
}


Random rnd = new Random();
private string randString;
public override void SetRandom()
{
    //Debug.Log("a " + AString.Count + " b " + BString.Count);
    for (int i = 0; i < AString.Count; i++)
    {
        //Debug.Log(AList[i].Element.Length);
        AString[i] = GetRandomString(8);
        //Debug.Log(AString[i]);
        SetCellArray(AList[i], AString[i]);
        SetInputDecimal(ADecInput[i], AString[i]);

        BString[i] = GetRandomString(8);
        SetCellArray(BList[i], BString[i]);
        SetInputDecimal(BDecInput[i], BString[i]);

        //Debug.Log("//" + i);
    }

}

public string GetRandomString(int strLength)
{
    randString = "";
    for (int i = 0; i < strLength; i++)
        randString += rnd.Next(0, 2).ToString();
    return randString;
}

public string GetStringByArray(CellArray array)
{
    var str = "";
    foreach (var cell in array.Element)
    {
```

```
            str = String.Concat(str, cell.Value.ToString());
        }
        return str;
    }
}
```