УПРАВЛЕНИЕ ОБРАЗОВАНИЯ ГОРОДА ПЕНЗЫ
ЦКОиМОУ г. Пензы
Муниципальное бюджетное общеобразовательное учреждение
«Лицей современных технологий управления №2» г. Пензы
(МБОУ ЛСТУ №2 г. Пензы)

«Старт в науку» 2020, Пенза

Posit числа:
теоретическое и практическое применения

Автор:
Нагаев Марат Тимурович, учащийся 11 класса, МБОУ «Лицей современных технологий управления №2» г. Пензы
Научные руководители:
Р. Ю. Алексеева, учитель информатики МБОУ ЛСТУ № 2 г. Пензы

Пенза
2020

Оглавление

Введение3

Описание формата Posit4

Описание класса на C++14

Веб технологии: Javascript, React, HTML16

Список используемой литературы20

Введение

Январь 2019 года был объявлен на Хабре месяцем Posit. Такое внимание к данному числовому типу не случайно. Исследователи предполагают, что новый тип данных, называемый posit, разработан в качестве прямой замены чисел с плавающей точкой. Повышение точности вычислений является актуальной и практико ориентированной проблемой, с которой мы сталкиваемся достаточно часто при организации научных вычислений.

Поэтому в своем исследовании мы решили разработать приложение калькулятор, оперирующий posit числами.

Актуальность данного проекта в создании javascript библиотеки для работы с posit числами и практически применимого калькулятора, позволяющего решать различные задачи, в том числе научные и инженерные.

Цель исследования

При проведении исследования была поставлена следующая цель: разработать калькулятор, поддерживающий операции над posit числами.

Были выполнены следующие задачи:

- Изучить posit числа
- Изучить WebAssembly и его применение
- Изучить библиотеку React
- Применить полученные знания для создания калькулятора

Описание формата Posit

В 1985 году Институт инженеров электротехники и электроники (IEEE) установил стандарт IEEE 754, отвечающий за форматы чисел с плавающей запятой и арифметики, которому суждено будет стать образцом для всего железа и ПО на следующие 30 лет.

И хотя большинство программистов использует плавающую точку в любой момент без разбора, когда им нужно проводить математические операции с вещественными числами, из-за определённых ограничений представления этих чисел, быстродействие и точность таких операций часто оставляют желать лучшего.

Много лет стандарт подвергался резкой критике со стороны специалистов по информатике, знакомых с этими проблемами, однако больше всего среди них выделялся американский инженер и специалист по компьютерным технологиям Джон Густафсон, который выступал с идеями замены плавающей запятой на что-то более подходящее. В данном случае более подходящим вариантом считается posit или unum – третий вариант результата его исследования «универсальных чисел». Он говорит, что числа формата posit решат большинство главнейших проблем стандарта IEEE 754, дадут улучшенную производительность и точность, и при этом будут использовать меньше битов. Что ещё лучше, он заявляет, что новый формат может заменить стандартные числа с плавающей запятой «на лету», без необходимости менять исходный код приложений.

Для специалистов, занимающихся суперкомпьютерами, одно из главных преимуществ формата posit состоит в том, что можно достичь большей точности и динамического диапазона, используя меньше битов, чем числа из IEEE 754. И не просто немного меньше. Дж. Густафсон сказал, что 32-битный posit заменяет 64-битное float почти во всех случаях, что может иметь серьёзные последствия для научных вычислений. Если уполовинить количество битов, можно не только уменьшить объёмы кэша, памяти и хранилища для этих чисел, но и серьёзно уменьшить ширину канала,

необходимого для передачи их на процессор и обратно. Это главная причина, по которой арифметика на базе posit, по его мнению, даст от двойного до четверного ускорения расчётов по сравнению с числами с плавающей запятой от IEEE.

Ускорения можно будет достичь через уплотнённое представление вещественных чисел. Вместо экспоненты и дробной части фиксированного размера, используемой в стандарте IEEE, posit кодирует экспоненту переменным количеством бит (комбинацией битов режима и битов экспоненты), так, что в большинстве случаев их требуется меньше. В итоге на дробную часть остаётся больше битов, что даёт большую точность. Динамическую экспоненту стоит использовать благодаря её сужающейся точности. Это означает, что величины с небольшой экспонентой, которые чаще всего используются, могут иметь большую точность, а реже используемые очень крупные и очень мелкие числа будут иметь меньшую точность. Работа Густафсона от 2017 года, описывающая формат posit, даёт подробное описание того, как это работает.

Ещё одно важное преимущество формата в том, что, в отличие от обычных чисел с плавающей запятой, posit дают одинаковые побитовые результаты на любой системе, чего часто нельзя гарантировать с форматом от IEEE (тут даже одинаковые вычисления на одной и той же системе могут дать разные результаты). Также новый формат расправляется с ошибками округления, переполнением и исчезновением значащих разрядов, денормализованными числами, и множеством значений типа not-a-number (NaN). Кроме того, posit избегает такой странности, как несовпадающие значения 0 и -0. Вместо этого формат использует для знака двоичное дополнение, как у целых чисел, что означает, что побитовое сравнение будет выполняться правильно.

С числами posit связано нечто под названием quire – механизм накопления, позволяющий программистам выполнять воспроизводимую линейную алгебру – процесс, недоступный обычным числам формата IEEE.

Он поддерживает обобщённую операцию совмещённого умножения-сложения и другие совмещённые операции, позволяющие вычислять скалярные произведения или суммы без ошибок округления или переполнений. Тесты, запущенные в Калифорнийском университете в Беркли, продемонстрировали, что операции quire проходят в 3-6 раз быстрее, чем последовательное их выполнение. Дж. Густафсон говорит, что они позволяют числам posit «драться за пределами своей весовой категории».

Хотя этот формат чисел существует всего пару лет, в сообществе специалистов по высокопроизводительным вычислениям (HPC) уже есть интерес к изучению возможностей их применения. В настоящий момент вся работа остаётся экспериментальной, и основана на предполагаемом быстродействии будущего железа или на использовании инструментов, эмулирующих posit-арифметику на обычных процессорах. Пока в производстве нет чипов, реализующих posit на аппаратном уровне.

Одно из потенциальных применений формата – строящийся радиоинтерферометр Square Kilometer Array (SKA), при проектировании которого рассматривают числа posit как способ кардинально уменьшить ширину канала и вычислительную нагрузку для обработки данных, поступающих с радиотелескопа. Необходимо, чтобы обслуживающие его суперкомпьютеры потребляли не более 10 МВт, и один из наиболее многообещающих способов достичь этого, по мнению проектировщиков, — использовать более плотный формат posit для того, чтобы вдвое урезать предполагаемую ширину канала памяти (200 ПБ/сек), канала передачи данных (10 ТБ/сек) и сетевого соединения (1 ТБ/сек). Вычислительная мощность также должна возрасти.

Разберем более подробно данный формат.

Итак, для начала вспомним как устроены традиционные вещественные числа (тип float).

Они состоят из трёх битовых частей: одного бита знака (s), N бит экспоненты (e) и M бит мантиссы (m).

Таким образом формат такого числа можно описать двумя числами: 1) сколько бит отведено под всё число в целом; 2) сколько из них отведено под поле экспоненты.

Формат битовой маски такого числа выглядит как:

s ee..ee mm..mm

Тут нужно заметить еще, что все поля содержат беззнаковые числа, а чтобы с помощью экспоненты выражать отрицательные числа, то оговорено, что если её биты все не равны нулю или единице, то из неё при трактовке вычитается середина диапазона который её битовое представление способно выразить (назовём его Emid). Кроме того поле мантиссы чаще всего (за исключением вырожденных случаев), выражая дробное число между 1.0 и 2.0 не содержит лидирующую единицу для повышенной ёмкости, выражая двоичную дробь 1.mmmmm...

Таким образом расшифровка содержимого числа выглядит так:

(-1)^s * 2^(e-Emid) * 1.mmmm...

Перейдем к описанию posit.

В отличие от IEEE чисел, экспонента и дробные части posit не имеют фиксированной длины.

Числа типа Posit вводят третье поле в свой формат, который называется regime (r), которое размещается между битами s и e:

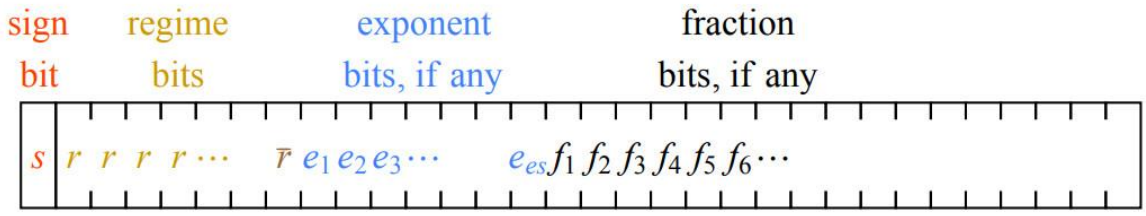


Рис 1. Обобщённый формат posit format для конечных ненулевых значений

Тип posit задается двумя числами: общее количество битов n, и максимальное количество битов, выделенных экспоненте, es. Вместе мы говорим, что у нас есть posit < n, es >.

Первый бит posit – бит знака. Если бит знака равен 1, то всё число трактуется как отрицательное целое с дополнением до двух – то есть биты инвертируются и добавляется единица. Это значит, что нет +0 и -0, например.

После бита знака приходят биты regime (режима). Число битов режима является переменным. Может быть от 1 до n-1 битов режима.

Поле regime сути своей это некий "буст" экспоненты. Это поле начинается с бита 0 или 1, которые повторяются до m раз, пока не встретится бит другого значения. В этом месте regime кончается, и его численное значение определяется так: если regime начиналось с 0, то оно равно – m, а если оно начиналось с 1, то оно равно m-1.

Если первый бит после бита знака равен 0, то биты режима продолжаются до тех пор, пока не закончатся биты или не появится 1. Аналогично, если первым битом после знака является 1, то биты режима продолжаются до тех пор, пока не закончатся биты или не появится 0. Бит, указывающий на конец прогона, не включается в режим; Режим является строкой всех нулей или всех единиц.

Для понимания битов режима, рассмотрим бинарные строки, показанные в таблице (рис.2), в которых k означает длину ведущей последовательности, а x в битовой стоке означает безразличное состояние.

Binary	0000	0001	001x	01xx	10xx	110x	1110	1111
Numerical meaning, k	−4	−3	−2	−1	0	1	2	3

Рис.2. Представление бинарной строки

Будем называть длину ведущей последовательности режимом числа. Бинарные строки начинаются с определённого количества нулей или единиц, идущих подряд, за которыми идёт противоположный бит, или достигается конец строки. Биты режима выделены янтарным цветом, противоположный бит выделен коричневым.

Биты экспоненты могут отсутствовать. Максимальное число битов экспоненты задается числом es. Если после бита знака, бита режима и бита

завершения режима имеются, по меньшей мере, биты, следующие биты принадлежат экспоненте. Если осталось менее es битов, то какие биты остаются принадлежащими экспоненте.

Если после бита знака, бита режима, бита завершения режима и бита экспоненты остались какие-либо биты, то все они принадлежат дроби.

Так как regime сам кодирует свою длину, то числа типа posit как ни странно характеризуются ровно теми же числами, что и IEE-754 – сколько бит отведено под всё число и сколько бит отведено под экспоненту:

posit< bits, es >,

однако в силу той же переменной длины regime он может занимать хоть все оставшиеся после знака биты числа, поэтому более точно параметр es означает "максимальное число бит отведенное под экспоненту". Если regime займёт много места, то фактическое число бит в экпоненте может быть меньше es, вплоть до нуля (тогда экспонента принимается за 0). И только если после regime и экспоненты в Posit останутся еще биты – они все отходят под мантиссу. Кроме того экспонента всегда воспринимается как число без знака, т.к. в отрицательные степени показателя число теперь заводит regime.

В отличие от IEEE-754, где много исключительных случаев - плюс-минус нули, денормализованные числа, плюс-минус бесконечности и параметризуемые NaN в числах posit существует только два особых значения:

- все биты числа равны нулю - обозначает ноль

- все биты числа равны единице - обозначает бесконечность (и "плюс" и "минус" для упрощения обработки).

Технически реализуемы даже числа posit< 1, 0 >, то есть однобитовые позиты, но выражать они смогут только 0 и бесконечность по правилу выше.

posit< 2, 0 > уже может содержать 4 значения: 0, -1, +1 и бесконечность.

Далее рассмотрим, как описанные выше компоненты представляют собой действительное число.

Пусть b – бит знака в posit. Знак s числа, представленного битовой комбинацией, является положительным, если этот бит равен 0, и отрицательным в противном случае.

Пусть m – количество битов в режиме, то есть длина пробега одинаковых битов, следующих за знаковым битом. Тогда пусть k = -m, если режим состоит из всех 0, и пусть k = m-1 в противном случае.

Нужно ввести еще понятие useed (или просто u) – это число равное

Это число, которое возводится в степень regime для формирования части posit.

Рис. 3 показывает пример значений useed.

es	0	1	2	3	4
useed	2	$2^2 = 4$	$4^2 = 16$	$16^2 = 256$	$256^2 = 65536$

Рис.3. Представление posit

Дробь f равна 1 + биты дроби, интерпретируемых как следующие за двоичной точкой. Например, если дробные биты равны 10011, то f = 1.10011 в двоичном формате.

Итак, вооружившись всеми вышеприведенными битовыми полями приведём теперь математическую расшифровку числа posit:

Заметьте, что если под мантиссу не хватило бит, то она естественным образом просто становится равно единице, а если не осталось бит под экспоненту, то она принимается равной нулю и 2^e тоже становится равной единице, таким образом даже только с помощью знака и regime позиты продолжают нам сообщать какие-то числа.

На рис. 4 приведен пример posit.

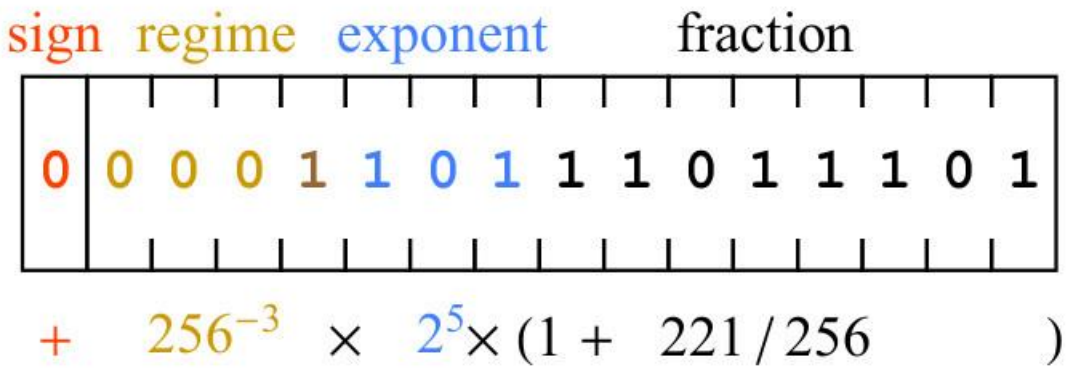


Рис. 4. Пример битовой строки posit и её математический смысл

Бит знака 0 означает, что значение положительное. Биты режима 0001 имеют последовательность из трёх нулей, что означает, что k=-3, следовательно, масштабирующий коэффициент, вносимый битами режима, равен 256^{-3}. Биты экспоненты,101, представляют 5 как беззнаковое двоичное целое, и вносимый масштабирующий коэффициент равен 2^5. Наконец, биты дробной части 11011101 представляют число 221, то есть дробная часть равна 1+221/256. Выражение, записанное под битовым полем на рис.4. приводит нас к результату $477/134217728 \approx 3.55393 \times 10^{-6}$

Качественное сравнение форматов Float и Posit

В формате posit нет “NaN” (нечисел), вместо этого, вычисления прерываются, и обработчик прерывания должен либо сообщить об ошибке, либо каким-либо образом обработать ошибку и продолжить вычисления, но числа posit не допускают присваивание некоего значения, сигнализирующего о логической ошибке, чем, по определению, и является нечисло. Это существенно упрощает аппаратное обеспечение. Если программист видит необходимость в использовании значений NaN, это показывает, что программа пока не завершена, и в отладочном окружении должны использоваться числа valid для поиска и устранения подобных ошибок. Также, posit не имеет $+\infty$ и $-\infty$, как float, однако, числа valid поддерживают открытые интервалы (maxpos,$+\infty$) и ($-\infty$,$-$maxpos), которые дают возможность представить неограниченную величину любого знака, и необходимость в

знаковой бесконечности будет означать лишь то, что вместо чисел posit нужно применять значения valid.

Также в представлении posit нет «отрицательного нуля», отрицательный ноль, это ещё один логический недостаток, существующий в стандарте IEEE float. С числами posit, если a=b, то f(a)=f(b). Стандарт IEEE 754 говорит, что число, обратное к -0 равно −∞, а число, обратное к +0, равно +∞, но также говорит, что -0 равно +0. Следовательно, подразумевается, что −∞=+∞?

Числа float имеют сложный алгоритм сравнения a=b. Если любое из (a, b) равно NaN, результат сравнения всегда отрицательный, даже если их битовое представление одинаковое. Если битовое представление разное, то по-прежнему есть возможность, чтобы a было равно b, так как отрицательный ноль равен положительному нулю! В posix проверка равенства такая же, как у целых чисел: если биты равны, числа равны. Если любой бит отличается, они не равны. Числа posit имеют такое же отношение (a < b), как и знаковые целые, как и со знаковыми целыми, вы должны следить, чтобы не произошло переполнение с изменением знака, но вам не нужны отдельные машинные инструкции для сравнения posit, если у вас есть инструкции для сравнения знаковых целых.

В формате posit нет денормализованных чисел, то есть нет специальной комбинации бит, показывающей, что скрытый бит равен 0 вместо 1. Posit не использует антипереполнение, вместо этого, используется постепенное снижение точности, что обеспечивает функциональность антипереполнения и его симметричного случая, переполнения (в отличие от posit, стандарт float ассиметричен, и использует эти битовые паттерны для представления большого и бесполезного множества NaN-значений).

Формат float имеет одно преимущество перед posit, при разработке аппаратного обеспечения фиксированное расположение битов экспоненты и дробной части позволяет декодировать их параллельно. В формате posit, нужно соблюдать некоторую последовательность, сначала декодируя биты режима, а потом остальные биты. Есть простой способ обойти это

ограничение, похожий на трюк, используемый для увеличения скорости обработки исключений во float: несколько дополнительных бит присоединяются к каждому значению так, чтобы сохранить в них информацию о размере при декодировании инструкции.

Описание класса на C++

C++ проект состоит из нескольких файлов:

posit.c

op1.c

op2.c

pack.c

posit_types.h

util.c

А также вспомагательные файлы-заголовки, описывающие структуру классов.

В файле posit_types.h содержится описание основных типов posit:

```
#include <stdint.h>
#include <stdbool.h>

#define POSIT_LUTYPE uint64_t
#define POSIT_UTYPE uint32_t
#define POSIT_STYPE int32_t
#define POSIT_WIDTH 32
#define POSIT_ZERO ((POSIT_UTYPE)0x00000000)
#define POSIT_ONE ((POSIT_UTYPE)0x40000000)
#define POSIT_NAR ((POSIT_UTYPE)0x80000000)
#define POSIT_MSB ((POSIT_UTYPE)0x80000000)
#define POSIT_MASK ((POSIT_UTYPE)0xFFFFFFFF)
```

В файлах op1.c и op2.c содержатся такие операции над «сырыми» данными как сложение, вычитание, извлечения квадратного корня и прочее. Под сырыми данными имеется ввиду следующая структура:

```
struct unpacked_t
{
bool neg;
```

```
POSIT_STYPE exp;
POSIT_UTYPE frac;
};
```

Структура содержит бит знака, мантиссу и экспоненту. Описание находится в файле pack.h

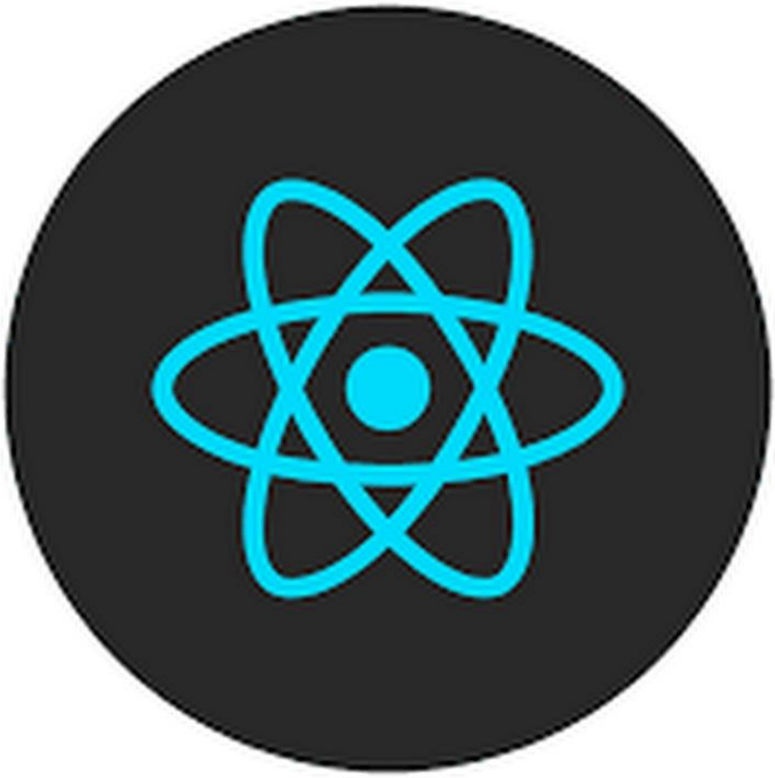
Файл pack.c содержит функции, связанные с распаковкой и запаковкой позита. pack_posit создает позит из битов мантиссы и экспоненты, pack_float аналогично создает float.

util.c содержит утилиты, такие как проверка на равенство NaR, 0, <0.

Веб технологии: Javascript, React, HTML

Как известно, основой любого сайта служат язык разметки гипертекста HTML, скриптовый язык Javascript и язык таблицы стилей CSS.

У каждого из них своя роль. Так, с помощью html формируется «каркас» будущего веб-приложения, с помощью Javascript функциональная составляющая и css для определения стиля элементов. Однако в нашем проекте использован более современный подход. Для создания интерфейса было решено использовать библиотеку React от компании Facebook.



Немного истории. React был создан Джорданом Валке, разработчиком программного обеспечения из Facebook. В первый раз реакт был использован в новостной ленте Facebook в 2011 году. Исходный код React был открыт в 2013 году на конференции JSConf US. С самого начала реакт предназначался для разработки пользовательских интерфейсов, причем не только в браузере. Например, с помощью библиотеки React Native можно писать мобильные приложения. Но вернемся к обычному реакту. В его основе лежит

функциональность, компоненты и декларативный подход к описанию интерфейсов.

Компонент - чистая функция или класс, возвращающая код, который должен быть отрисован. Рассмотрим пример объявления компонента:

```
class App() {
  render(){
    return (
    <div className="App">
      <p>Hello World</p>
    </div>
  );

  }
}
```

Здесь мы объявляем класс App, в котором есть метод render.Он возвращает html код.

Однако, функции в JS не могут возвращать html. На самом деле, это не html код, а JSX – расширение языка JavaScript. После процедуры компиляции JSX превращается в обычный JS код. Для создания компонентов есть два способа: класс и функция. У нас в проекте используется функциональный подход, так как он считается более современным и лаконичным.

Virtual DOM

Еще одним преимуществом React является виртуальный DOM. DOM – Document Object Model, программный интерфейс для контроля html кода.

Изменение дерева элементов – «дорогой» процесс, занимающий много ресурсов. Вместо этого можно использовать виртуальный DOM. Мы можем вносить изменения в копию, исходя из наших потребностей, а после этого применять изменения к реальному DOM.
При этом происходит сравнение DOM-дерева с его виртуальной копией, определяется разница и запускается перерисовка того, что было изменено. Такой подход работает быстрее, потому как не включает в себя все тяжеловесные части реального DOM.

Состояние – внутренний объект компонента, один из краеугольных камней React. Состояние доступно только внутри компонента и является

изменяемым. Пример функционального компонента с применением состояния:

```
function App() {

        const [value,setValue] = useState(0);//состояние: в переменной value 0
        var inc = ()=> setValue(value+1);
        var dec = ()=> setValue(value-1);
        return (
           <div className="App">
                <p>{counter}</p>
                <button onClick={inc}>+</button>
                <button onClick={dec}>-</button>
             </div>

        );
}
```

Результат работы кода:

При нажатии на кнопку результат автоматически обновится.

Здесь используется хук useState. Хуки – это новые функции для управления состоянием приложения и другие возможности React для функциональных компонентов.

Еще одним хуком является useEffect.

useEffect(()=>{

document.addEventListener('keydown',keyboardHandler,false); //до примонтирования

return ()=>document.removeEventListener('keydown',keyboardHandler,false); //после отмонтирования

});

В этом примере хук используется для однократной привязки обработчика нажатия на клавиатуру.

WebAssembly

Еще одной технологией, которая была использована в проекте является WebAssembly (сокращено wasm). WebAssembly появился как развитие проекта инженеров Mozilla asm.js.

Главной целью WebAssembly является поддержка работы программ, написанных на C и C++ в браузере путем компиляции кода в особый бинарный формат, поддерживаемый и исполняемый браузером. Это позволяет очень быстро адаптировать C++ проекта для работы в браузере. Например, движок Unity уже можно запустить в браузере, который поддерживает эту технологию. Одним из преимуществ WASM является его скорость. По оценкам, выполнение байткода WebAssembly всего на 20% медленнее машинного кода.

Для компиляции C++ в WebAssembly используется компилятор emcc, развиваемый сообществом. Команда для компиляции выглядит так:

emcc –std=c++11 main.cpp –o main.html WASM=1

Где main.cpp – название входного файла, а main.html – выходного.

Помимо html кода автоматически добавляется файл main.js, содержащий вспомогательный JS код.

Для вызова функции C++ из браузера используется метод

WebAssembly.ccall, которому передается название функции, тип аргументов, массив аргументов и тип возвращаемого значения.

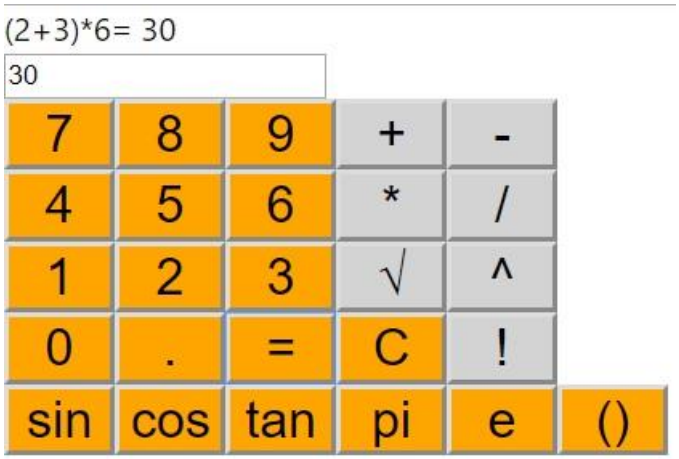


Рис.5. Скриншот приложения

Список используемой литературы

1. C/C++. Программирование на языке высокого уровня / Т. А. Павловская. – СПб.: Питер, 2003. – 461 с.
2. Michael Feldman. Новый подход может помочь нам избавиться от вычислений с плавающей запятой математикой [Электронный ресурс] // Режим доступа https://habr.com/ru/post/462385/
3. John L. Gustafson, Isaac Yonemoto. Posit-арифметика: победа над floating point на его собственном поле математикой [Электронный ресурс] // Режим доступа https://habr.com/ru/post/465723/