СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ

Разработать вычислительное устройство, состоящее из двух взаимосвязанных частей – операционного и управляющего автоматов – и выполняющее следующие операции:

Операция №1: НОК двух целых чисел в дополнительном коде;

Операция №2: Квадрат числа, представленного в экспоненциальном формате.

Управляющий автомат: Схема с регулярной адресацией.

Функциональные схемы разрабатываются с использованием многоразрядных мультиплексоров, дешифраторов, сумматоров, регистров, счётчиков, компараторов, ПЗУ с чётким указанием информационных, управляющих и синхронизирующих входов.

2 ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ

2.1 Реализация вычислительного устройства

Первая операция – квадрат числа, представленного в экспоненциальном формате – будет реализована как возведение числа в квадрат в формате IEEE 754 для чисел с половинной точностью. Поэтому входной канал для ввода данных (INPUT) будет шестнадцатиразрядным. Так как вычислительное устройство проектируется для двух операций, то и вторая операция – НОК двух целых чисел в дополнительном коде – будет также работать с шестнадцатиразрядными числами. Наименьшее общее кратное можно найти как произведение двух операндов, делённое на НОД (Формула 2.1).

Формула 2.1 – НОК через НОД:

$$\text{НОК}(a, b) = \frac{a*b}{\text{НОД}\ (a,b)} \tag{2.1}$$

Для этой операции нет необходимости сохранять информацию об остатке от деления, поэтому воспользуемся алгоритмом деления без восстановления остатка. Поиск НОД выполним при помощи алгоритма Евклида через вычитание. Умножение будем выполнять «в столбик».

Вычисление числа в формате с плавающей точной предполагает операции над мантиссой и экспонентой, однако возведение числа в квадрат всегда даёт положительное число, поэтому учитывать знак во время выполнения операции не имеет смысла. Для работы с экспонентой потребуется операция вычитания и сложения, а для перемножения мантисс – умножение, которое также будет реализовано «в столбик».

Для этого набора операций были разработаны простые команды, такие как перемещение информации из одного регистра в другой, сложить результат двух регистров и прочее. Операционный автомат преобразует входящую команду при

помощи дешифратора и выполняет требуемую операцию. Все команды выполняются за один такт, а сами команды напоминают ассемблерный стиль. Вышеописанный проект удобно синергирует с требуемым операционным автоматом, что в значительной степени упрощает разработку. Перечень всех команд с их описанием приведен в Таблице 2.1.

Таблица 2.1 – Перечень команд вычислительного устройства

Команда	Описание
CMP	Сравнение двух чисел в беззнаковом виде. Результатом операции может быть «не равно», «равно» и «меньше». Сам результат записывается в D-триггеры и будет доступен на следующем такте. Результат сравнения хранится до следующей операции сравнения. Шины данных: P_NEQUAL, P_EQUAL, P_LESS для «не равно», «равно» и «меньше» соответственно. Пример команды: cmp rga rgb.
MOV	Копировать содержимое правого регистра в левый. Пример: mov rga rgb – здесь содержимое регистра rgb будет скопировано в регистр rga.
AND	Побитовое «И» для двух регистров. Результат операции сохраняется в регистр rgk. Пример команды: and rga rgb.
OR	Побитовое «ИЛИ» для двух регистров. Результат операции сохраняется в регистр rgk. Пример команды: or rga rgb.
NOT	Побитовое «НЕ» для одного регистра. Результат операции сохраняется в регистр rgk. Пример команды: not rga.
XOR	Побитовое «ИСКЛЮЧАЮЩЕЕ ИЛИ» для двух регистров. Результат операции сохраняется в регистр rgk. Пример команды: xor rga rgb.
ADD	Сложение двух регистров в беззнаковом виде при помощи сумматора. В случае переполнения возникает флаг OVERFLOW, который хранится в течение следующего такта в D-триггере. Результат операции записывается в регистр rgk. Пример команды: add rga rgb.
SUB	Вычитание двух регистров в дополнительном коде при помощи двух сумматоров. Первый преобразуем вычитаемое, а второй складывает два числа. В случае переполнения возникает флаг OVERFLOW, который хранится в течение следующего такта в D-триггере. Результат операции записывается в регистр rgk. Пример команды: sub rga rgb.
ROM	Установить ПЗУ, в которой хранятся константы, в заданное значение. Всего выделено 16 ячеек памяти для констант, поэтому и выбор самой константы производится среди заданных 16 значений. Номер ячейки ПЗУ задаётся в шестнадцатеричном виде с принятым соглашением о наименованиях. Пример команды: rom 0x1.
INP	Запись значения с панели ввода в заданный регистр при помощи нажатия кнопки RI (ready input). Из-за особенностей псевдоассемблера, поле для второго операнда требуется занять значением 0xF. Пример команды: inp rga 0xF.
FIN	Завершение работы вычислительного устройства. Устанавливает флаг RO (ready out) как 1. Флаг RO хранится в D-триггере. Пример команды: fin.
ERR	Установка регистра ошибок. Регистр для хранения ошибок двухразрядный, поэтому всего возможно 4 состояния, где состояние 0 – это, по соглашению, отсутствие ошибок. Остальные состояния определяются в зависимости от операции. Пример команды: err 0x1.
SHR	Побитовый логический сдвиг вправо. Пример команды: shr rga.

Продолжение Таблицы 2.1

SHL	Побитовый логический сдвиг влево. Пример команды: shr rga.
BRK	Отладочная команда, которая останавливает выполнение операций в заданной точке программы до перехода сигнала BREAKPOINT с панели ввода на высокий уровень. После такового перехода, программа продолжит выполнение. Соответственно, остальные подобные команды будут игнорироваться до тех пор, пока BREAKPOINT находится в состоянии 1. Пример команды: brk
JMP	Оператор безусловного перехода. Изменяет состояние счётчика команд за заданное значение. Само значение определяется точкой входа внутри программы. Точка входа задаётся в ассемблерном стиле – на новой строке с оператором двоеточия на конце. Пример операции: jmp is_nan Также существуют операторы безусловного перехода на основе флагов, установленных операцией cmp и and. Для «не равно», «равно» и «меньше» — это jne, jme jml соответственно, а для фрага OVERFLOW – jmo. Если флаг равен 1, то происходит безусловный переход, в противном случае данная команда игнорируется.

Для работы с этой системой команд был написан псевдоассемблер для удобного преобразования команд в данные для ПЗУ Logisim на C++ (Приложение А). По соглашению хранения данных в ПЗУ, любые данные, которые импортируются в ПЗУ извне, требуют в начале файла на первой строке набор символов «v2.0 raw», а на следующей строке через пробел перечисленные данные для записи в ПЗУ слева направо. Программа для работы двух операций, требуемых в поставленной задаче, представлена в Листинге 2.1. Пояснение к работе алгоритма оставлено в комментариях к программе, которые, по соглашению, идут после двойного слеша «//».

В вычислительном устройстве установлено 7 универсальных 16-разрядных сдвиговых регистра (Рисунок 2.1). Доступ к каждому из них для чтения возможен через шины WIRE1 и WIRE2, а запись – только через WIRE2. Включение и выключение записи реализовано при помощи дешифратора. Шины WIRE1 и WIRE2 реализованы через мультиплексоры 16-1. Сброс автомата выполняется безусловно при помощи кнопки RST.

Управляющий автомат и операционный автомат в Logisim представлены на одной схеме. Поэтому всё вычислительное устройство представлено на Рисунке 2.2.

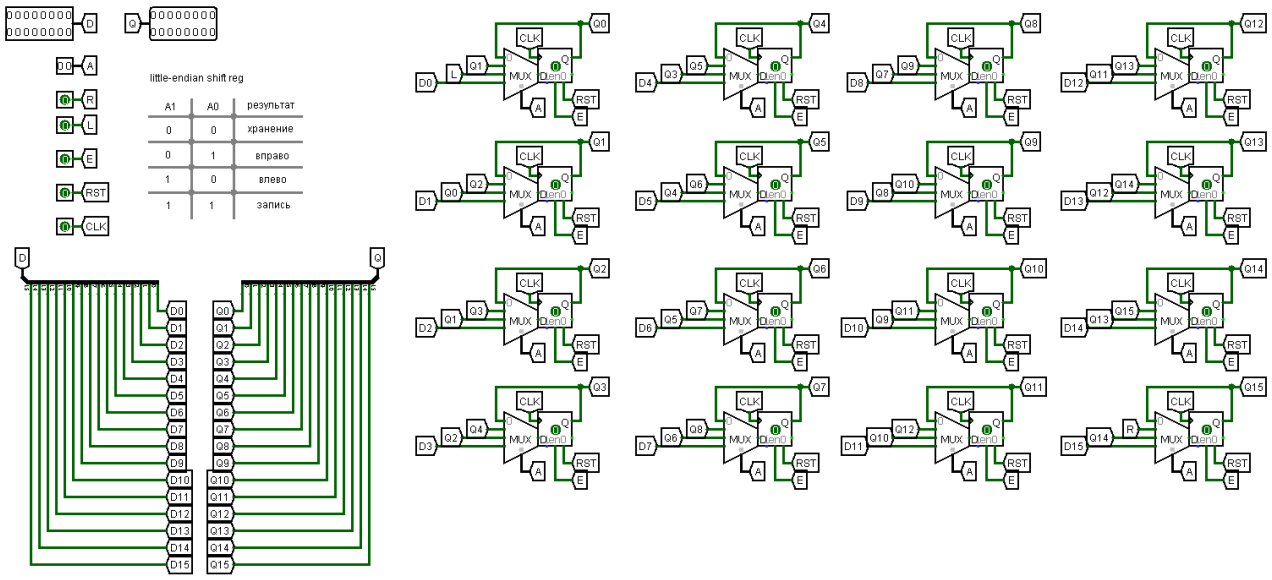

Рисунок 2.1 – Шестнадцатиразрядный универсальный сдвиговый регистр на D-триггерах

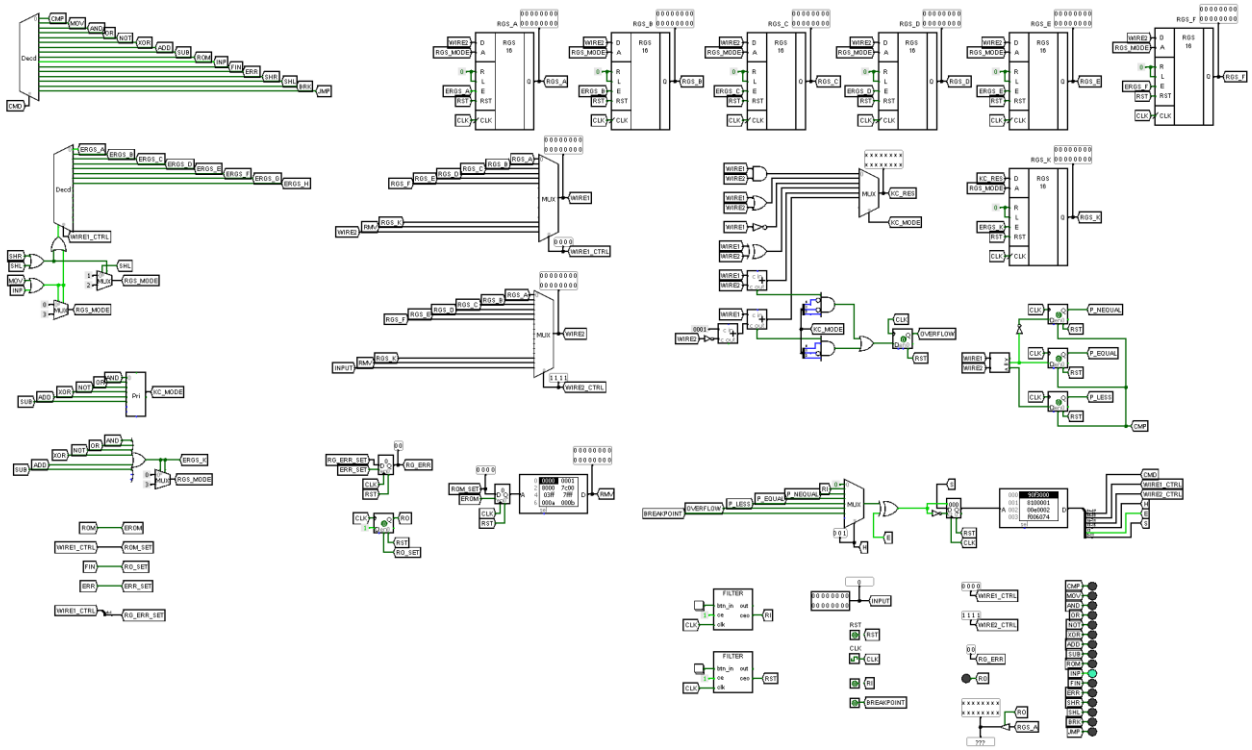

Рисунок 2.2 – Вычислительное устройство

Листинг 2.1 – Программа на псевдоассемблере для НОК и возведения числа в квадрат в экспоненциальной форме.

```
// значения для rom
// 0x0 - 0x0000 - константа 0
// 0x1 - 0x0001 - константа 1
// 0x2 - 0x8000 - битовая  маска для знака
// 0x3 - 0x7C00 - битовая маска для экспоненты
// 0x4 - 0x03FF - битовая маска для мантиссы
```

Продолжение Листинга 2.1.

```
// 0x5 - 0x7FFF - битовая маска дял экспоненты и мантиссы
// 0x6 - 0x000A - нормальное значение для экспоненты
// 0x7 - 0x000B - максимальное значение для экспоненты, если на каждом шаге
происходило переполнение
// 0x8 - 0x000F - сдвиг порядков для экспоненты ( = 15 )
// 0x9 - 0x0400 - бит по умолчанию для мантиссы
// 0xA - 0xFFE0 - маска для проверки переполнения мантиссы
// 0xB - 0x0800 - маска проверки переполнения мантиссы
// 0xC - 0xFFFF - максимальное значение регистра
// 0xD - 0x0020 - переполнение для экспоненты

// ввод номера операции
// 0 - НОК
// 1 - возведение числа в квадрат в экспоненциальном виде
inp rga 0xF
rom 0x1
cmp rga rmv
jme sqr
jmp nok

////////////////////////////////////////////////////////////////////////////

nok:
    // информация о регистрах (скорее всего уже неверная):
    // rga - регистр с первым числом. тут же будет сохранён НОД
    // rgb - регистр со вторым числом
    // rgc - счётчик
    // rgd - регистр для счёта числа сдвигов
    // rge - регистр с перемноженными числами из rga и rgb
    // rgf - регистр для сдвига числа при умножении

    // информация об ошибках
    // 0x0 - всё гуд
    // 0x1 - переполнение
    // 0x2 - один из операндов равен нулю

    inp rga 0xF // ввод первого числа
    inp rgb 0xF // ввод второго числа

    // проверка на ноль
    rom 0x0 // 0x0000
    cmp rga rmv
    jme err_0x2
    cmp rgb rmv
    jme err_0x2


    // числа даны в дополнительном коде, поэтому делаем проверку на
отрицательное число. если да, то берём его модуль
    rom 0x2 // 0x8000 - проверка на максимальное отрицательное число
    cmp rga rmv
    jme err_0x1
    cmp rgb rmv
    jme err_0x1

    // число в rga по модулю
    rom 0x2 // 0x8000
    and rga rmv
    cmp rgk rmv
    jme add1_rga

    // число в rgb по модулю
```

Продолжение Листинга 2.1.

```
    check_modul_rgb:
        rom 0x2 // 0x8000
        and rgb rmv
        cmp rgk rmv
        jme add1_rgb
        jmp prepare_multiply

    add1_rga:
        not rga
        mov rga rgk
        rom 0x1 // 0x0001
        add rga rmv
        mov rga rgk
        jmp check_modul_rgb

    add1_rgb:
        not rgb
        mov rgb rgk
        rom 0x1 // 0x0001
        add rgb rmv
        mov rgb rgk
        jmp prepare_multiply

    // -----> умножение регистров <-----
    prepare_multiply:
        // подготовка регистров для сдвигов
        mov rgd rga
        mov rge rgb

    multiply:
        // умножаем столбиком. для этого каждый раз делаем проверку на ноль
для второго операнда, так как его мы будем двигать
        // rga - первый операнд, rgb - второй операнд, rgc - результат
умножения, rgd - регистр для сдвига первого операнда влево,
        // rge - регистр для сдвига второго операнда вправо

        // проверка на завершение умножения
        rom 0x0 // 0x0000
        cmp rge rmv
        jme end_multiply

        // проверяем необходимость добавлять число на текущем шаге итерации
        rom 0x1 // 0x0001
        and rge rmv
        cmp rgk rmv
        jme add_multiply
        jmp finalize_multiply

        add_multiply:
            add rgc rgd
            jmo err_0x1 // произошло переполнение
            mov rgc rgk

        finalize_multiply:
            shl rgd
            shr rge
        jmp multiply

    end_multiply:
    jmp nod

    // ошибка: переполнение регистра
```

Продолжение Листинга 2.1.

```
    err_0x1:
        err 0x1
        fin
    // ошибка: один из операндов равен нулю
    err_0x2:
        err 0x2
        fin

    nod:
        // -----> вычисление НОД по алгоритму Евклида через вычитание <-----
        cmp rga rgb
        jme end_nod
        cmp rga rgb
        jml rgb_more
        sub rga rgb
        mov rga rgk
        jmp nod
        rgb_more:
            sub rgb rga
            mov rgb rgk
        jmp nod

    end_nod:

    // -----> деление без восстановления остатка <-----
    // делим число rga - nod на rgc - результат умножения
    // сбрасываем все регистры на всякий случай
    rom 0x0 // 0x0000
    mov rgb rmv
    mov rgd rmv
    mov rge rmv
    mov rgf rmv

    // rgb - расширение делимого, rgc - делимое оригинальное, rga - делитель
оригинальный, rgd - отрицательный делитель, rge - результат
    // rgf - счётчик числа операций для деления

    // устанавливаем счётчик
    rom 0x8 // 0x000F
    mov rgf rmv

    not rga
    mov rgd rgk
    rom 0x1 // 0x0001
    add rgd rmv
    mov rgd rgk

    div:
        // устанавливаем флаг для rgb
        shl rgb // сдвиг расширенного делимого
        rom 0x2 // 0x8000
        and rgc rmv
        cmp rgk rmv
        shl rgc // сдвиг оригинального делимого
        jme positive_add
        jmp add_divisor

        // добавляем 1 к расширенному делимому
        positive_add:
            rom 0x1 // 0x0001
            add rgb rmv
            mov rgb rgk
```

Продолжение Листинга 2.1.

```
        // добавлем делитель
        add_divisor:
            shl rge // переполнение может произойти, тогда добавим 1, а пока
записываем 0 в результат

            // устанавливаем флаг для добавления положительного или
отрицательного делителя
            rom 0x2 // 0x8000
            and rgb rmv
            cmp rgk rmv
            jne add_negative

            // добавляем положительное делимое
            add rgb rga
            jmo set_one // произошло переполнение. записываем 1 в результат
            mov rgb rgk
            jmp for_cycle

            // добавляем отрицательное делимое
            add_negative:
                add rgb rgd
                jmo set_one // произошло переполнение. записываем 1 в
результат
                mov rgb rgk
                jmp for_cycle

            set_one:
                rom 0x1 // 0x0001
                add rge rmv
                mov rge rgk

        // считаем количество итераций до 0, так как надо 16 итераций, 1
итерацию уже сделали, осталось 15
        for_cycle:
            rom 0x0 // 0x0000
            cmp rgf rmv
            jme end_div
            rom 0x1 // 0x0001
            sub rgf rmv
            mov rgf rgk
            jmp div

    end_div:
        // записываем результат деления в rga
        mov rga rge
        fin

////////////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////////////

sqr:
    // значения для rom
    // 0x0 - 0x0000 - константа 0
    // 0x1 - 0x0001 - константа 1
    // 0x2 - 0x8000 - битовая  маска для знака
    // 0x3 - 0x7C00 - битовая маска для экспоненты
    // 0x4 - 0x03FF - битовая маска для мантиссы
    // 0x5 - 0x7FFF - битовая маска дял экспоненты и мантиссы
```

Продолжение Листинга 2.1.

```
    // 0x6 - 0x000A - нормальное значение для экспоненты
    // 0x7 - 0x000B - максимальное значение для экспоненты, если на каждом
шаге происходило переполнение
    // 0x8 - 0x000F - сдвиг порядков для экспоненты ( = 15 )
    // 0x9 - 0x0400 - бит по умолчанию для мантиссы
    // 0xA - 0xFFE0 - маска для проверки переполнения мантиссы
    // 0xB - 0x0800 - маска проверки переполнения мантиссы
    // 0xC - 0xFFFF - максимальное значение регистра
    // 0xD - 0x0020 - переполнение для экспоненты

    inp rga 0xF // ввод числа в формате половинной точности
    rom 0x3     // 0x7C00
    and rga rmv
    cmp rgk rmv
    jne not_special

    // ----->   проверка специальных случаев   <-----
    // проверка на NaN
    rom 0x4 // 0x03FF
    and rga rmv
    rom 0x0 // 0x0000
    cmp rgk rmv
    jne is_nan

    // результатом является +inf
    inf:
        rom 0x3     // 0x7C00
        mov rga rmv
        fin

    is_nan:
        rom 0x5 // 0x7FFF
        mov rga rmv
        err 0x1 // установка кода ошибки для NaN
        fin

    // ----------->   основная обработка   <-----------
    // rga - оригинальное число, rgb - экспонента, rgc - мантисса, rgd -
регистр для сдвига первого операнда влево при умножении,
    // rge - регистр для сдвига второго операнда вправо, rgf - регистр для
единицы округления
    not_special:
        // сначала вычисляем мантиссу. все необходимые действия по
нормализации будем производить после.
        // если при перемножении мантисс получается, что число превышает
допустимые лимиты, то делаем сдвиг.

        // вырезаем мантиссу и экспоненту
        rom 0x3 // 0x7C00
        and rga rmv
        mov rgb rgk
        rom 0x4 // 0x03FF
        and rga rmv
        mov rgc rgk

        // добавляем скрытый бит
        rom 0x0 // 0x0000
        cmp rgb rmv
        jme skip_add_hidden_bit

        rom 0x9 // 0x0400
        or  rgc rmv
```

Продолжение Листинга 2.1.

```
        mov rgc rgk

        skip_add_hidden_bit:
            // подготовка к умножению
            mov rgd rgc
            mov rge rgc
            // обнуляем мантиссу
            rom 0x0 // 0x0000
            mov rgc rmv
            // обнуляем экспоненту
            mov rgb rmv

        mantiss_multiply:
            // умножаем столбиком. для этого каждый раз делаем проверку на
ноль для второго операнда, так как его мы будем двигать
            // если вышли за рамки умножения (то есть дальше скрытого бита),
то делаем сдвиг вправо всех регистров и вычитаем 1 из экспоненты

            // проверка на завершение умножения
            rom 0x0 // 0x0000
            cmp rge rmv
            jme end_mantiss_multiply

            // проверяем необходимость добавлять число на текущем шаге
итерации
            rom 0x1 // 0x0001
            and rge rmv
            cmp rgk rmv
            jme add_mantiss_multiply
            jmp finalize_mantiss_multiply

            add_mantiss_multiply:
                add rgc rgd
                mov rgc rgk
                jmp finalize_mantiss_multiply

            mantiss_multiply_overflow:
                // сохраняем единицу для округления
                rom 0x1 // 0x0001
                and rgc rmv
                mov rgf rgk

                // при переполнении сдвигаем регистры вправо и добавляем к
экспоненте 1
                shr rgc
                shr rgd
                rom 0x1 // 0x0001
                add rgb rmv
                mov rgb rgk

            finalize_mantiss_multiply:
                // проверка на переполнение при сдвиге
                rom 0xB // 0x0800
                cmp rmv rgd // для числа, на которое умножаем
                jml mantiss_multiply_overflow

                shl rgd
                shr rge
            jmp mantiss_multiply

        end_mantiss_multiply:
```

Продолжение Листинга 2.1.

```
            // добавляем 1 для округления результата, если не было
переполнения мантиссы
            // если было переполнение, то нужно округлить мантиссу с учётом
этой единицы
            // проверяем на переполнение мантиссы
            rom 0xB // 0x0800
            and rgc rmv // для результата умножения
            cmp rgk rmv
            jne without_overflow_mantissa

            // если было переполнение
            rom 0x1 // 0x0001
            and rgc rmv
            mov rgf rmv
            add rgc rgf
            mov rgc rgk

            // сдвиг мантиссы вправо, добавляем 1 к экспоненте за этот сдвиг
            rom 0x1 // 0x0001
            add rgb rmv
            mov rgb rgk
            shr rgc
            jmp exp

            without_overflow_mantissa:
                add rgc rgf
                mov rgc rgk

        // -----> считаем экспоненту <-----
        exp:
            // нормализуем мантиссу и экспоненту. для посчитанной экспоненты
нормальным значением является 10 - 0x000A.
            // максимально возможное - 11 - 0x000B, минимально возможное - 0 -
0x0000
            // если в результате из финального значения экспоненты вычесть 10,
то получем конечный результат, если там было 11
            // если там было меньше, то выполним денормализацию позже

            // вырезаем экспоненту
            rom 0x3 // 0x7C00
            and rga rmv
            mov rgd rgk

            // выравниваем экспоненту для вычислений. rgf - счётчик
            rom 0x6 // 0x000A
            mov rgf rmv
            rshift_exp_loop:
                rom 0x0 // 0x0000
                cmp rgf rmv
                jme end_rshift_exp_lopp
                shr rgd
                rom 0x1 // 0x0001
                sub rgf rmv
                mov rgf rgk
                jmp rshift_exp_loop
            end_rshift_exp_lopp:
            // удвоение изначальной экспоненты
            add rgd rgd
            mov rgd rgk

            // вычитаем сдвиг экспоненты
            rom 0x8 // 0x000F
```

Продолжение Листинга 2.1.

```
            sub rgd rmv
            mov rgd rgk

            // добавляем результирующую экспоненту после умножения
            add rgd rgb
            mov rgd rgk

            // вычитаем сдвиг для нормализованной экспоненты
            rom 0x6 // 0x000A
            sub rgd rmv
            mov rgd rgk

            // проверка на положительное значение экспоненты. если она
отрицательная,
            // то проводим нормализацию до тех пор, пока не получим нулевую
экспоненту
            rom 0x2 // 0x8000
            and rgd rmv
            cmp rgk rmv
            jme normalize

            skip_denormal_shift:
            // проверка на переполнение экспоненты. результатом будет +inf
            rom 0xD // 0x0020
            and rgd rmv
            cmp rmv rgk
            jme inf

            brk
            // безусловный сдвиг вправо для экспоненты, если вид экспоненты
денормализованный????????
            rom 0x0 // 0x0000
            cmp rgd rmv
            jne skip_shift
            shr rgc

            skip_shift:

            // задвигаем экспоненту обратно.
            lshift_exp:
                rom 0x0 // 0x0000
                mov rgf rmv
                rom 0x6 // 0x000A
                mov rgf rmv
                lshift_exp_loop:
                    rom 0x0 // 0x0000
                    cmp rgf rmv
                    jme end_lshift_exp_lopp
                    shl rgd
                    rom 0x1 // 0x0001
                    sub rgf rmv
                    mov rgf rgk
                    jmp lshift_exp_loop
                end_lshift_exp_lopp:
                jmp finalize

    finalize:
        // финальная проверка на переполнение экспоненты
        rom 0x3 // 0x7C00
        cmp rgd rmv
        jme inf
```

Продолжение Листинга 2.1.

```
        // обрезаем мантиссу
        rom 0x4 // 0x03FF
        and rgc rmv
        mov rgc rgk

        // собираем результат
        rom 0x0 // 0x0000
        mov rga rmv
        or  rgd rgc
        mov rga rgk
        fin

    normalize:
        brk

        normalize_loop:
            // двигаем мантиссу до тех пор, пока экспонента не станет нулём
            rom 0x0 // 0x0000
            cmp rgd rmv
            jme end_normalize_loop

            rom 0x1 // 0x0001
            add rgd rmv
            mov rgd rgk

            // сохраняем последний бит для округления
            and rgc rmv
            mov rgf rgk
            shr rgc

            jmp normalize_loop

        end_normalize_loop:
            // добавляем округляющую единицу
            add rgc rgf
            mov rgc rgk

            // двигаем на 1 разряд, так как число денормализованное ( с
округлением?! )?????????
            rom 0x1
            and rgc rmv
            mov rgf rgk
            add rgc rgf
            mov rgc rgk
            shr rgc
            jmp finalize

////////////////////////////////////////////////////////////////////////////
```

2.2 Тестирование вычислительного устройства

После запуска тактового генератора необходимо выбрать нужную операцию. При вводе любого значения, кроме 0000 0000 0000 0001, устройство ожидает значения для поиска НОК для чисел в дополнительном коде. В

противном случае, значение для возведения числа в квадрат в экспоненциальной форме. На Рисунках 2.3 – 2.7 приведены результаты тестирования для НОК, а на рисунках 2.8 – 2.12 – для возведения числа в квадрат в экспоненциальном формате. Название рисунка содержит входные значения и ожидаемый результат.

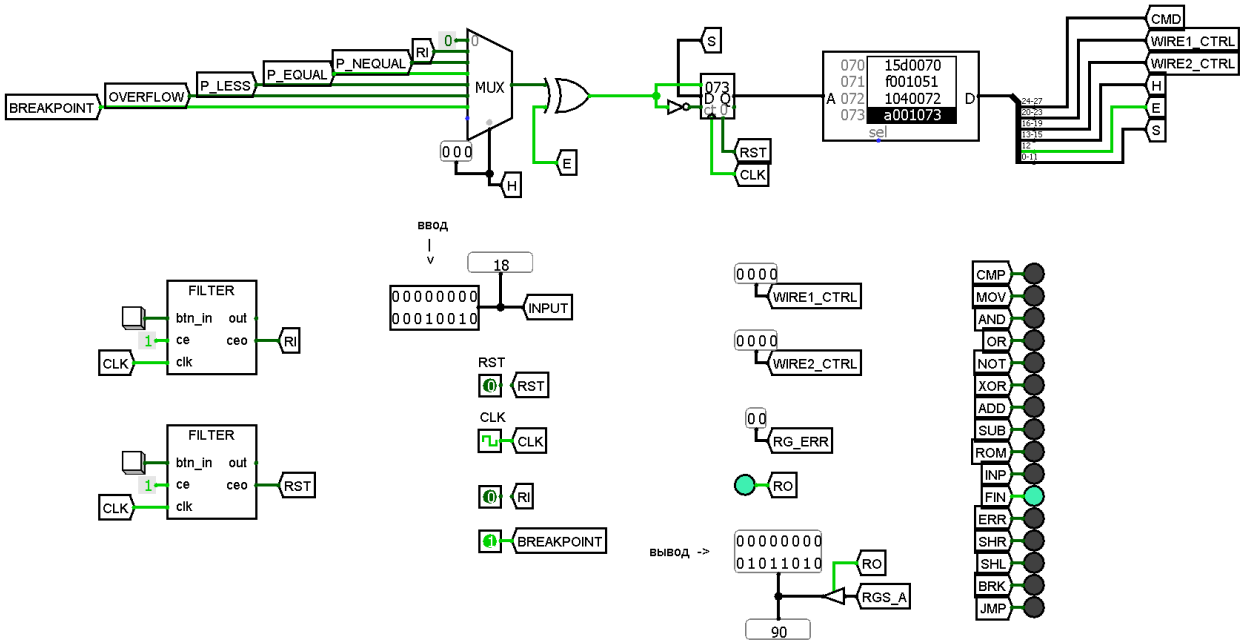

Рисунок 2.3 – Операция НОК для чисел 30 и 18. Ожидаемый результат – 90

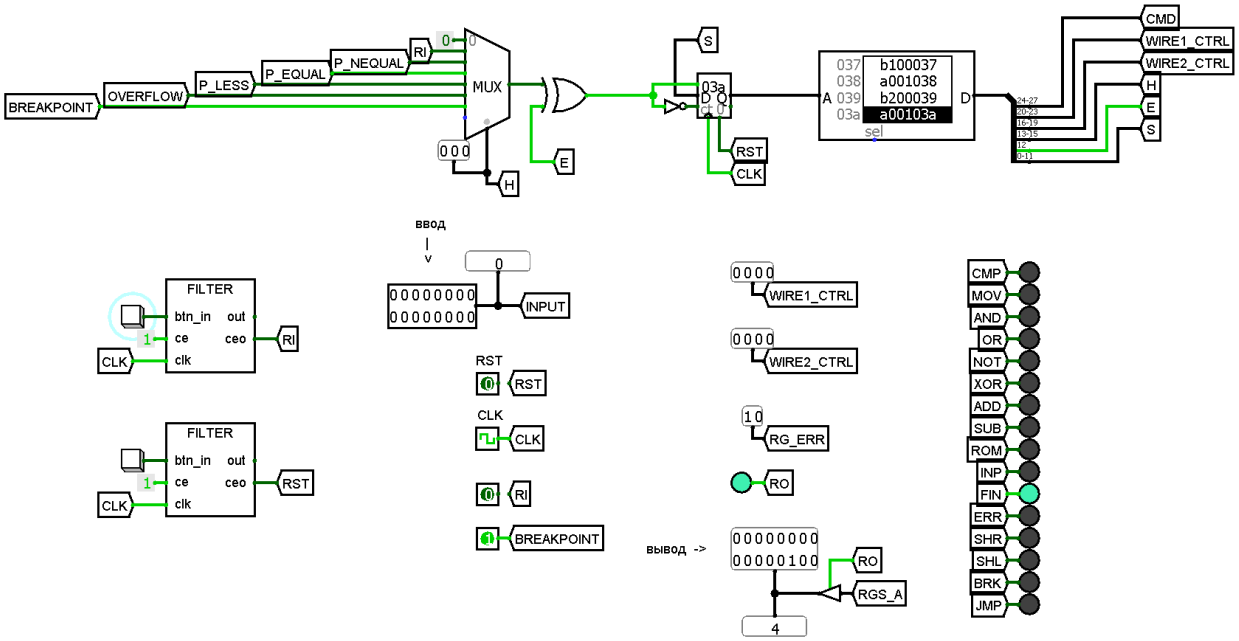

Рисунок 2.4 – Операция НОК для чисел 4 и 0. Ожидаемый результат – ошибка 0x2 – ввод нуля

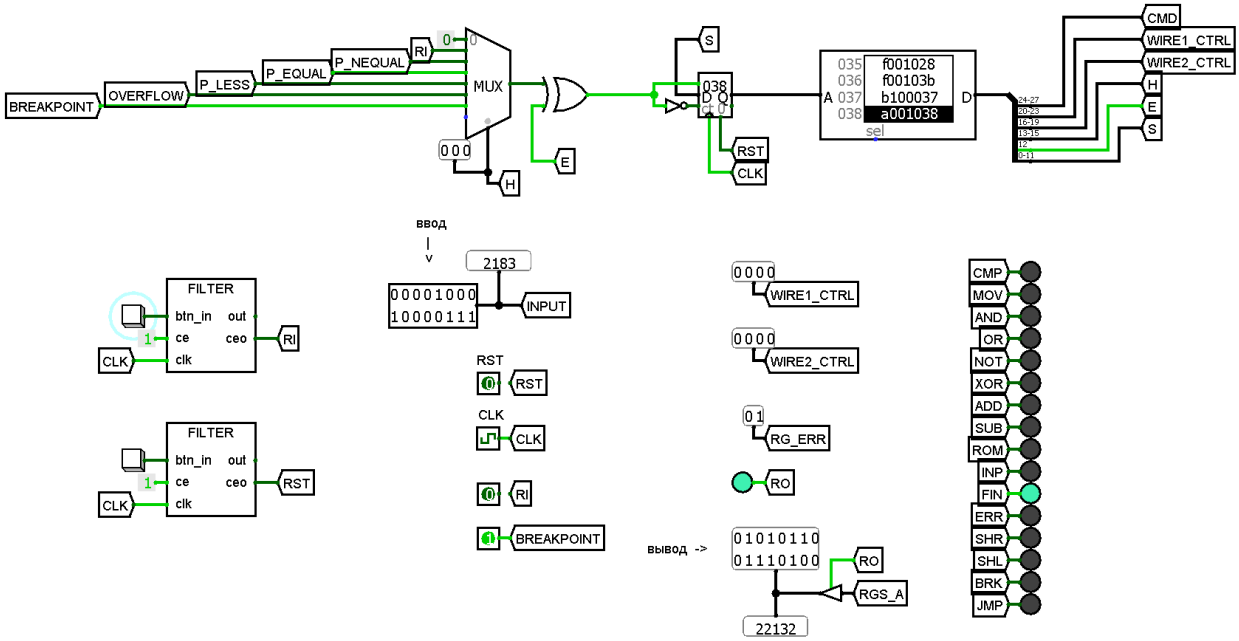

Рисунок 2.5 – Операция НОК для чисел -22132 и 2183. Ожидаемый результат – ошибка 0x1 – переполнение регистра при выполнении операции

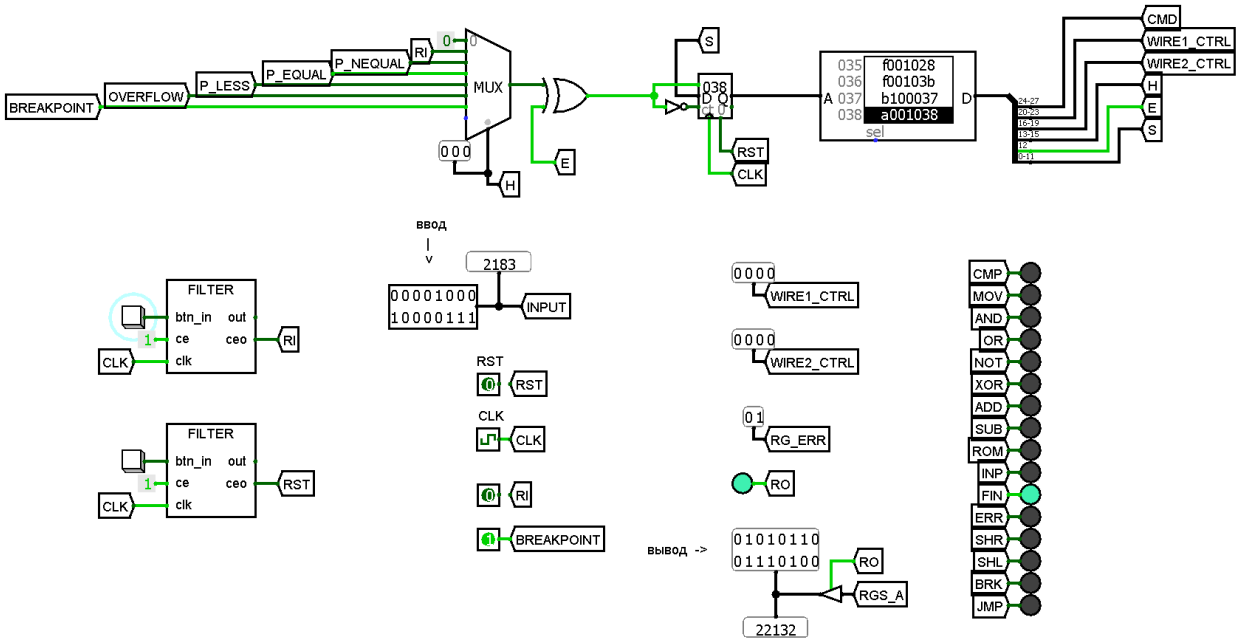

Рисунок 2.6 – Операция НОК для чисел 209 и -47. Ожидаемый результат – 9823

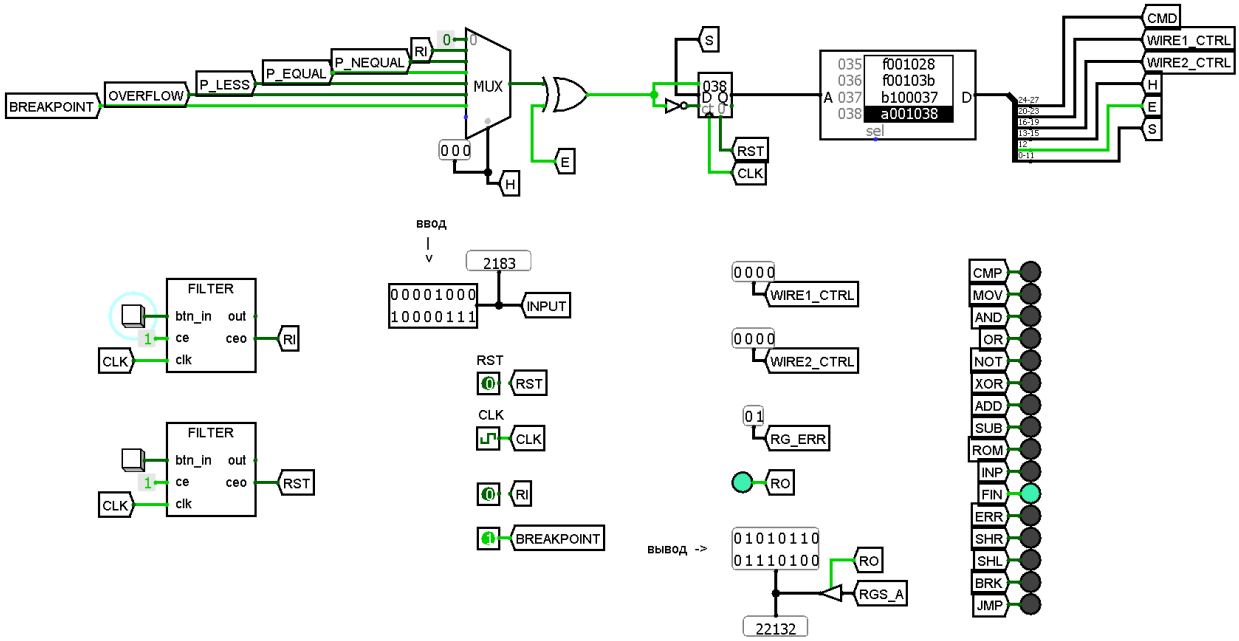

Рисунок 2.7 – Операция НОК для чисел 10195 и -5. Ожидаемый результат – 10195

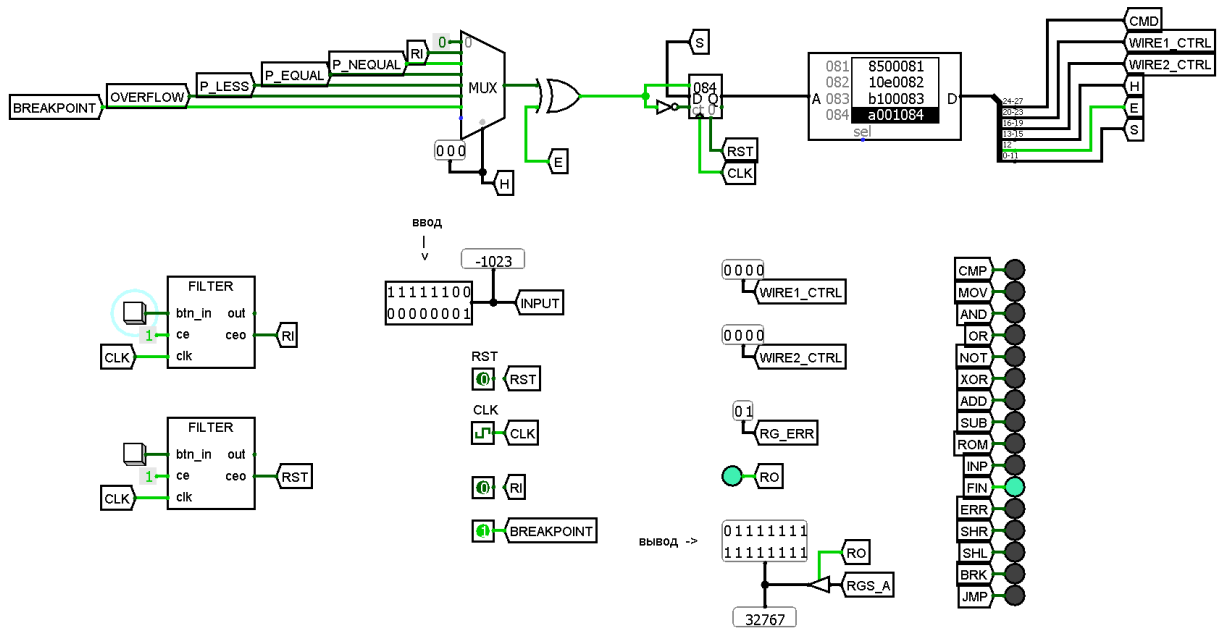

Рисунок 2.8 – Операция возведения числа в квадрат в экспоненциальной форме для NaN. Ожидаемый результат – ошибка 0x1 – ввод NaN

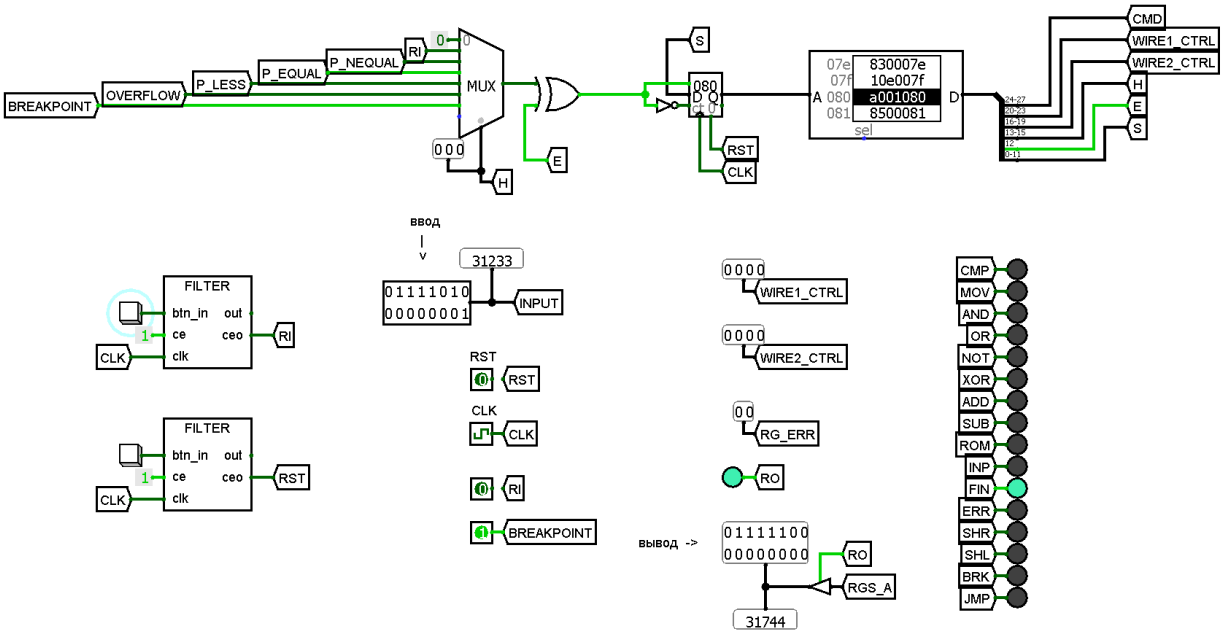

Рисунок 2.9 – Операция возведения числа в квадрат в экспоненциальной форме для 0111 1010 0000 0001 (4.1E4). Ожидаемый результат – 0111 1100 0000 0000 (inf)

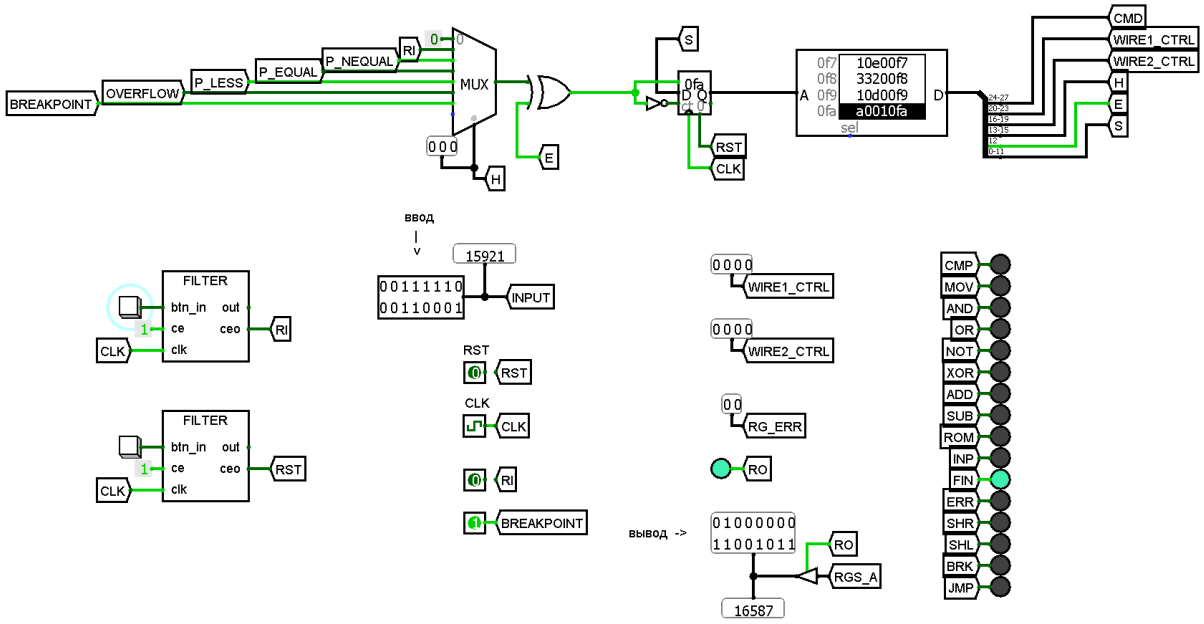

Рисунок 2.10 – Операция возведения числа в квадрат в экспоненциальной форме для 0011 1110 0011 0001 (1.548). Ожидаемый результат – 0100 0000 1100 1011 (2.396)

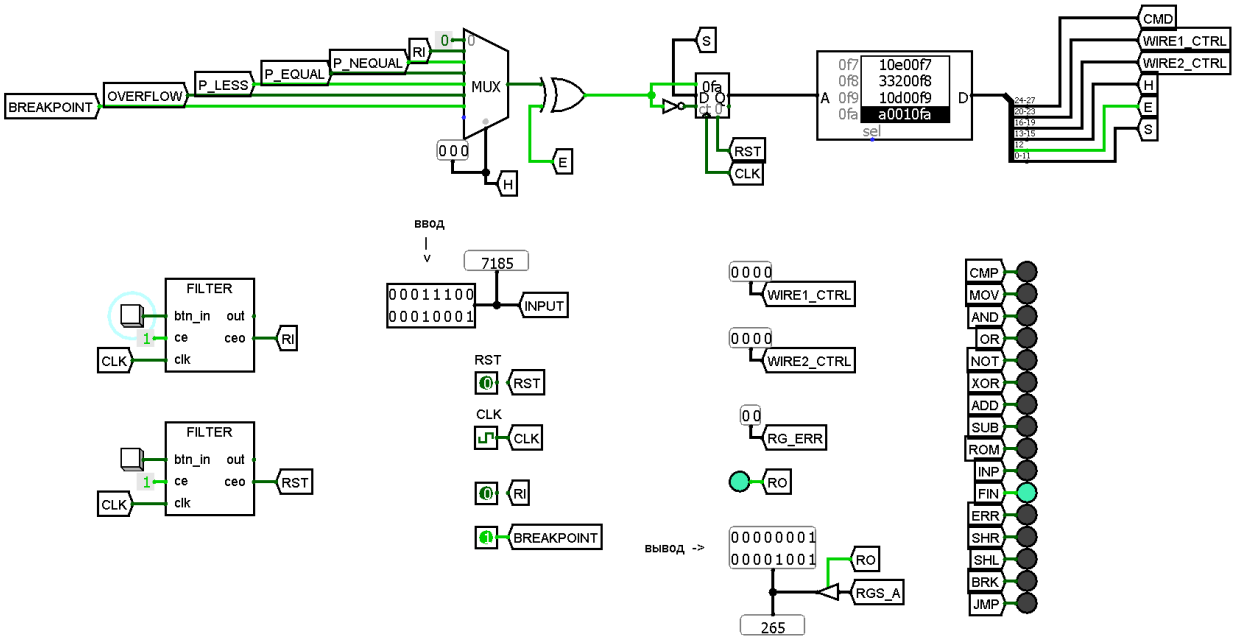

Рисунок 2.11 – Операция возведения числа в квадрат в экспоненциальной форме для 0001 1100 0001 0001 (0.00397). Ожидаемый результат – 0000 0001 0000 1001 (1.58E-5)

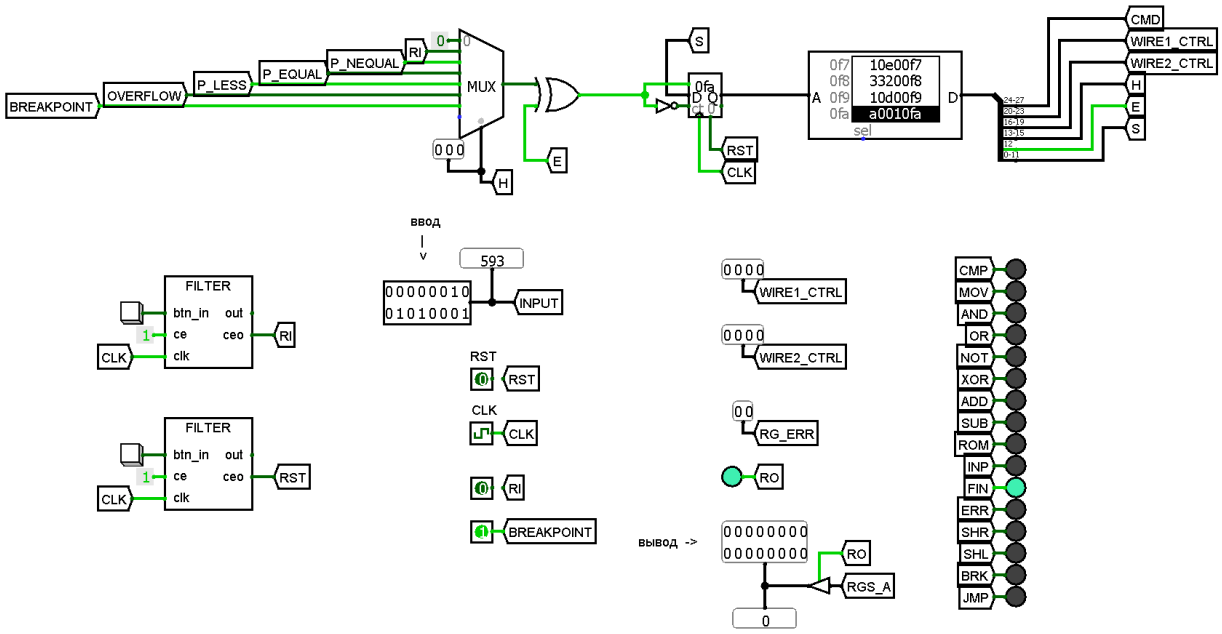

Рисунок 2.12 – Операция возведения числа в квадрат в экспоненциальной форме для 0000 0010 0101 0001 (3.535E-5). Ожидаемый результат – 0000 0000 0000 0000 (0)

3 ЗАКЛЮЧЕНИЕ

В ходе работы было построено вычислительное устройство (Рисунок 2.2) для двух команд: поиск НОК для двух чисел в дополнительном коде и возведение чисел в квадрат в экспоненциальной форме. Все вычисления производились для шестнадцатиразрядных чисел, соответственно, возведение чисел в квадрат производилось для чисел с половинной точностью в соответствии со стандартом IEEE 754. Была разработана система команд для управления операционным автоматом (Таблица 2.1), которая в значительной степени упростила реализацию всех операции, так как это стало задачей программирования (Листинг 2.1) на псевдоассемблере и преобразования этих команд в машинный код при помощи программы на C++ (Приложение А). Тестирование вычислительного устройства показало (Рисунки 2.3 – 2.12), что последнее работает корректно.

ПРИЛОЖЕНИЕ

Приложение А – Программа для преобразования команд в машинный код на языке программирования C++.

Приложение А

Программа для преобразования команд в машинный код на языке программирования C++

Листинг А.1 – Программа для преобразования команд в машинный код.

```
#include <iostream>
#include <bitset>
#include <vector>
#include <fstream>
#include <unordered_map>
#include <string>
#include <sstream>
#include <math.h>
#include <stdio.h>
#include <exception>
#include <set>
#include <stdint.h>

const std::size_t ROM_SIZE = std::pow(2, 9);
const std::size_t ROM_CMD = 28;
using bit_cmd = std::bitset<ROM_CMD>;

std::string to_hex_string(const bit_cmd& _cmd) {
    const std::size_t num_bits = _cmd.size();
    char* hex_str = new char[num_bits/4 + 1];
    std::size_t s = 0;
    hex_str[num_bits/4] = '\0';

    for (std::size_t i = 0; i < num_bits; i += 4) {
        std::size_t b = 0;
        for (std::size_t j = 0; j < 4; j++) {
            int v = num_bits - i - j - 1;
            b += _cmd[num_bits - i - j - 1] << 3 - j;
        }
        if (0 <= b && 9 >= b)
            hex_str[s++] = static_cast<char>(b) + '0';
        else if (10 <= b && 15 >= b)
            hex_str[s++] = static_cast<char>(b - 10) + 'a';
    }

    std::string hex_string(hex_str);
    delete[] hex_str;
    return hex_string;
}

const std::unordered_map<std::string, unsigned short> cmd = {
    {"cmp", 0x0}, // compare 2 register:
cmp rga rgb
    {"mov", 0x1}, // move data from second register to first:
mov rga rgb
    {"and", 0x2}, // bitwise AND. result saved on rgk:
and rga rgb
    {"or",  0x3}, // bitwise OR. result saved on rgk:
or  rga rgb
    {"not", 0x4}, // bitwise NOT. result saved on rgk:
not rga rgb
```

Продолжение Листинга А.1.

```
    {"xor", 0x5}, // bitwise XOR. result saved on rgk:
xor rga rgb
    {"add", 0x6}, // sum 2 register. result saved on rgk. rga + rgb = rgk:
add rga rgb
    {"sub", 0x7}, // subtract 2 register. result saved on rgk. rga - rgb =
rgk:  sub rga rgb
    {"rom", 0x8}, // set rom channel:
rom 0x4
    {"inp", 0x9}, // save input value to selected resiter:
inp 0xF rga // input value saved on rga
    {"fin", 0xA}, // set ready out flag:
fin
    {"err", 0xB}, // set error code:
err 0x1
    {"shr", 0xC}, // right bitwise shift for selected register:
shr rga
    {"shl", 0xD}, // left bitwise shift for selected register:
shl rga
    {"brk", 0xE}, // breakpoint. breakpoint input should be 1 to continue
brk

    {"jmp", 0x10}, // always
    {"jne", 0x11}, // if not equal
    {"jme", 0x12}, // if equal
    {"jml", 0x13}, // if less
    {"jmo", 0x14}, // overflow exception
};
const std::unordered_map<std::string, unsigned short> wire = {
    {"rga", 0x0},
    {"rgb", 0x1},
    {"rgc", 0x2},
    {"rgd", 0x3},
    {"rge", 0x4},
    {"rgf", 0x5},
    // {"rgg", 0x6},

    {"rgk", 0xD},
    {"rmv", 0xE},
    {"inp", 0xF}
};
std::unordered_map<std::string, std::size_t> jmp_address;

std::size_t command_line = 0;
std::size_t all_line = 0;
bit_cmd line = {};

void parse_file(std::string input) {
    // delete comment
    std::size_t comment_pos = input.find("//");
    std::string l = (comment_pos != std::string::npos) ? input.substr(0,
comment_pos) : input;

    if (l.find(":") != std::string::npos) {
        std::istringstream stream(l);
        std::string word;
        stream >> word;
        std::string jump_line = word.substr(0, word.find(":"));
        jmp_address.insert({ jump_line, command_line });
        return;
    }

    std::istringstream stream(l);
```

Продолжение Листинга А.1.

```
    std::string word;
    stream >> word;
    if (cmd.end() != cmd.find(word))
        command_line++;
    return;
}
bool readline(std::string input) {
    // reset line
    line = {};
    all_line++;

    // delete comment
    std::size_t comment_pos = input.find("//");
    std::string l = (comment_pos != std::string::npos) ? input.substr(0,
comment_pos) : input;
    // jump block
    if (l.find(":") != std::string::npos) {
        return false;
    }

    // parse line without comment
    std::istringstream stream(l);
    std::vector<std::string> result;
    std::string word;
    while (stream >> word) result.push_back(word);

    if (!result.size()) return false;
    if (result.size() > 3)
        throw std::invalid_argument("too much argument on line " + input);

    while (result.size() != 3) result.push_back("0x0");

    // assembly bitset
    std::unordered_map<std::string, unsigned short>::const_iterator iter_cmd =
cmd.find(result[0]);
    if (cmd.end() == iter_cmd)
        throw std::invalid_argument("undefined operator: " + result[0] + " on
line " + std::to_string(all_line));
    unsigned short cmd_num = iter_cmd->second;

    if (0x0 <= cmd_num && 0xF >= cmd_num) {
        std::unordered_map<std::string, unsigned short>::const_iterator iter1
= wire.find(result[1]);
        std::unordered_map<std::string, unsigned short>::const_iterator iter2
= wire.find(result[2]);
        unsigned short wire1_num;
        unsigned short wire2_num;
        if (wire.end() != iter1) wire1_num = (*iter1).second;
        else wire1_num = std::stoi(result[1], nullptr, 16);
        if (wire.end() != iter2) wire2_num = (*iter2).second;
        else wire2_num = std::stoi(result[2], nullptr, 16);

        // set cmd
        line[line.size() - 1]  = cmd_num   & 0x08ui16;
        line[line.size() - 2]  = cmd_num   & 0x04ui16;
        line[line.size() - 3]  = cmd_num   & 0x02ui16;
        line[line.size() - 4]  = cmd_num   & 0x01ui16;

        // set wire1_ctrl
        line[line.size() - 5]  = wire1_num & 0x08ui16;
        line[line.size() - 6]  = wire1_num & 0x04ui16;
        line[line.size() - 7]  = wire1_num & 0x02ui16;
```

Продолжение Листинга А.1.

```
        line[line.size() - 8]  = wire1_num & 0x01ui16;

        // set wire2_ctrl
        line[line.size() - 9]  = wire2_num & 0x08ui16;
        line[line.size() - 10] = wire2_num & 0x04ui16;
        line[line.size() - 11] = wire2_num & 0x02ui16;
        line[line.size() - 12] = wire2_num & 0x01ui16;

        if (0x9 == cmd_num) {
            // waiting for a sign RI
            line[line.size() - 13] = 0;
            line[line.size() - 14] = 0;
            line[line.size() - 15] = 1;

            // set E
            line[line.size() - 16] = 1;
        }
        if (0xA == cmd_num) {
            // set E
            line[line.size() - 16] = 1;
        }
        if (0xE == cmd_num) {
            // waiting for a sign BREAKPOINT
            line[line.size() - 13] = 1;
            line[line.size() - 14] = 1;
            line[line.size() - 15] = 0;

            // set E
            line[line.size() - 16] = 1;
        }

        // set current command
        line[line.size() - 17] = command_line & 0x800ui16;
        line[line.size() - 18] = command_line & 0x400ui16;
        line[line.size() - 19] = command_line & 0x200ui16;
        line[line.size() - 20] = command_line & 0x100ui16;
        line[line.size() - 21] = command_line & 0x080ui16;
        line[line.size() - 22] = command_line & 0x040ui16;
        line[line.size() - 23] = command_line & 0x020ui16;
        line[line.size() - 24] = command_line & 0x010ui16;
        line[line.size() - 25] = command_line & 0x008ui16;
        line[line.size() - 26] = command_line & 0x004ui16;
        line[line.size() - 27] = command_line & 0x002ui16;
        line[line.size() - 28] = command_line & 0x001ui16;

        command_line++;
    }
    if (0x10 <= cmd_num && 0x14 >= cmd_num) {
        // set cmd
        line[line.size() - 1] = 1;
        line[line.size() - 2] = 1;
        line[line.size() - 3] = 1;
        line[line.size() - 4] = 1;

        // set wire1_ctrl
        line[line.size() - 5] = 0;
        line[line.size() - 6] = 0;
        line[line.size() - 7] = 0;
        line[line.size() - 8] = 0;

        // set wire2_ctrl
        line[line.size() - 9]  = 0;
```

Продолжение Листинга А.1.

```
        line[line.size() - 10] = 0;
        line[line.size() - 11] = 0;
        line[line.size() - 12] = 0;

        if (0x10 == cmd_num) {
            // void sign
            line[line.size() - 13] = 0;
            line[line.size() - 14] = 0;
            line[line.size() - 15] = 0;

            // set E
            line[line.size() - 16] = 1;
        }
        if (0x11 == cmd_num) {
            // PNEQUAL sign
            line[line.size() - 13] = 0;
            line[line.size() - 14] = 1;
            line[line.size() - 15] = 0;
        }
        if (0x12 == cmd_num) {
            // PEQUAL sign
            line[line.size() - 13] = 0;
            line[line.size() - 14] = 1;
            line[line.size() - 15] = 1;
        }
        if (0x13 == cmd_num) {
            // PLESS sign
            line[line.size() - 13] = 1;
            line[line.size() - 14] = 0;
            line[line.size() - 15] = 0;
        }
        if (0x14 == cmd_num) {
            // OVERFLOW sign
            line[line.size() - 13] = 1;
            line[line.size() - 14] = 0;
            line[line.size() - 15] = 1;
        }

        std::unordered_map<std::string, std::size_t>::iterator iter_cline =
jmp_address.find(result[1]);
        if (jmp_address.end() == iter_cline)
            throw std::invalid_argument("undefiend jump point: " + result[1]);
        std::size_t cline = iter_cline->second;

        // set current command
        line[line.size() - 17] = cline & 0x800ui16;
        line[line.size() - 18] = cline & 0x400ui16;
        line[line.size() - 19] = cline & 0x200ui16;
        line[line.size() - 20] = cline & 0x100ui16;
        line[line.size() - 21] = cline & 0x080ui16;
        line[line.size() - 22] = cline & 0x040ui16;
        line[line.size() - 23] = cline & 0x020ui16;
        line[line.size() - 24] = cline & 0x010ui16;
        line[line.size() - 25] = cline & 0x008ui16;
        line[line.size() - 26] = cline & 0x004ui16;
        line[line.size() - 27] = cline & 0x002ui16;
        line[line.size() - 28] = cline & 0x001ui16;

        command_line++;
    }

    return true;
```

Продолжение Листинга А.1.

```
}
const std::string base_logisim_rom = "v2.0 raw\n";

int main()
{
    bool base_path = true;
    std::string path_asm = "C:\\files\\avt\\merge.asm";
    std::string path_logisim = "C:\\files\\avt\\merge";
    std::string path_source;
    std::string path_end;

    try {
        if (base_path) {
            std::cout << "rewrite file " + path_logisim + "\nuse " + path_asm
+ " ?\n[y/n] >>";
            std::string user_choose;
            std::cin >> user_choose;
            if ("y" == user_choose) {
                path_source = path_asm;
                path_end = path_logisim;
                goto read_files;
            }
        }

        std::cout << "enter path source >> ";
        std::cin >> path_source;
        std::cout << "enter endpoint >> ";
        std::cin >> path_end;

    read_files:
        std::ifstream asmfile(path_source);
        if (!asmfile.is_open()) {
            std::cerr << "error: could not open the file " << path_source <<
std::endl;
            return 1;
        }
        std::ofstream logisimfile(path_end);
        if (!logisimfile.is_open()) {
            std::cerr << "error: could not open the file " << path_end <<
std::endl;
            return 1;
        }
        logisimfile << base_logisim_rom;

        std::string assembly_line;
        while (std::getline(asmfile, assembly_line)) {
            parse_file(assembly_line);
        }
        asmfile.clear();
        asmfile.seekg(0, std::ios::beg);
        command_line = 0;
        std::size_t cl = 0;
        while (std::getline(asmfile, assembly_line)) {
            if (readline(assembly_line)) {
                cl++;
                if (cl != command_line) {
                    std::cout << "???????????? on line " << all_line <<
std::endl;
                }
                std::cout << " --> " << to_hex_string(line) << std::endl;
                logisimfile << to_hex_string(line);
                logisimfile << " ";
```

Продолжение Листинга А.1.

```
            }
        }

        std::cout << "total lines: " << command_line << std::endl;
        std::cout << "total cl: " << cl << std::endl;
        std::cout << "end process...";
    }
    catch (std::exception e) {
        std::cerr << "error: " << e.what() << " on line " << all_line <<
std::endl;
    }

    return 0;
}
```