ACS. Homework 4.

Горбачева Маргарита Валерьевна I БПИ-245

О самом ТЗ: написать программу, осуществляющую вычисление корня квадратного из положительного числа с заданной точностью, используя для этого соответствующую итерационную формулу. В своём решении я использовала формулу Ньютона - “Поиск решения осуществляется путём построения последовательных приближений и основан на принципах простой итерации” (https://ru.ruwiki.ru/wiki/Метод_Ньютона).

Выражается это следующей формулой:

$$\boxed{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.}$$

В реализации программы я воспользовалась формулой следующего вида:

$$\boxed{a_{n+1} = \frac{1}{2}\left(a_n + \frac{x}{a_n}\right)}$$

. Тут у нас *a* - число, которое мы извлекаем из корня. x_n - текущее приближение корня. X_{n+1} - след.приближение корня. Сама суть алгоритма заключается в том, что процесс повторяется, пока $|x_{n+1} - x_n| < \varepsilon$ (точность), то есть итерационно.

Работа алгоритма

```
sqrt_iterative:
    # Сохранение регистров в стеке (пролог функции)
    addi sp, sp, -20        # Выделяем 20 байт в стеке для сохранения 5 регистров (4 float + 1 integer)
    fsw fs0, 16(sp)         # Сохраняем регистр fs0 в стек по смещению 16 (сохраняем предыдущее значение)
    fsw fs1, 12(sp)         # Сохраняем регистр fs1 в стек по смещению 12
    fsw fs2, 8(sp)          # Сохраняем регистр fs2 в стек по смещению 8
    fsw fs3, 4(sp)          # Сохраняем регистр fs3 в стек по смещению 4
    sw ra, 0(sp)            # Сохраняем адрес возврата (ra) в стек по смещению 0

    # Сохранение аргументов, переданных в подпрограмму
    fmv.s fs0, fa0          # Копируем первый аргумент (число a) из fa0 в fs0 для постоянного хранения
    fmv.s fs1, fa1          # Копируем второй аргумент (точность ε) из fa1 в fs1 для постоянного хранения
```

Как видно из программы, мы работаем уже с вещественными числами и поэтому команды у нас отличаются от тех, с которыми мы работали раньше, то есть теперь мы пишем “fsw”, например, что означает Float Store Word.

```
    # x_{n+1} = 0.5 * (x_n + a / x_n)

    # a / x_n (деление исходного числа на текущее приближение)
    fdiv.s ft0, fs0, fs2     # ft0 = a / x_n (a - исходное число, x_n - текущее приближение)

    # x_n + a / x_n (сумма текущего приближения и отношения)
    fadd.s ft0, fs2, ft0     # ft0 = x_n + (a / x_n)

    # 0.5 * (x_n + a / x_n) (усреднение - получаем новое приближение)
    fdiv.s ft0, ft0, fs3     # ft0 = (x_n + a/x_n) / 2 = x_{n+1} (новое приближение)

    # Проверка точности: |x_{n+1} - x_n| < ε (достигли ли требуемой точности?)
    fsub.s ft1, ft0, fs2     # Вычисляем разность: ft1 = x_{n+1} - x_n (новое - старое)

    # Модуль разности (абсолютное значение)
    fabs.s ft1, ft1          # ft1 = |x_{n+1} - x_n| (берем модуль разности)

    # Если |разность| < точности, выходим из цикла
    flt.s t1, ft1, fs1       # Сравниваем: если |разность| < ε, то t1 = 1, иначе t1 = 0
    bnez t1, sqrt_end        # Если t1 ≠ 0 (достигли требуемой точности), переходим к концу

    # Обновление текущего приближения (если точность еще не достигнута)
    fmv.s fs2, ft0           # Обновляем текущее приближение: x_n = x_{n+1}
    j sqrt_loop              # Переходим к следующей итерации цикла

# Метка выхода из цикла (достигнута требуемая точность)
sqrt_end:
    # Возврат результата (последнее вычисленное приближение)
    fmv.s fa0, fs2           # Копируем результат из fs2 в fa0 (fa0 - регистр для возврата значения)

    # Восстановление регистров из стека (эпилог функции)
    flw fs0, 16(sp)          # Восстанавливаем регистр fs0 из стека
    flw fs1, 12(sp)          # Восстанавливаем регистр fs1 из стека
    flw fs2, 8(sp)           # Восстанавливаем регистр fs2 из стека
    flw fs3, 4(sp)           # Восстанавливаем регистр fs3 из стека
    lw ra, 0(sp)             # Восстанавливаем адрес возврата из стека
```

olumn: 14 Show Line Numbers

Я специально писала комментарии в коде сразу, чтобы в отчёте было наглядно видно, что происходит внутри самого алгоритма:

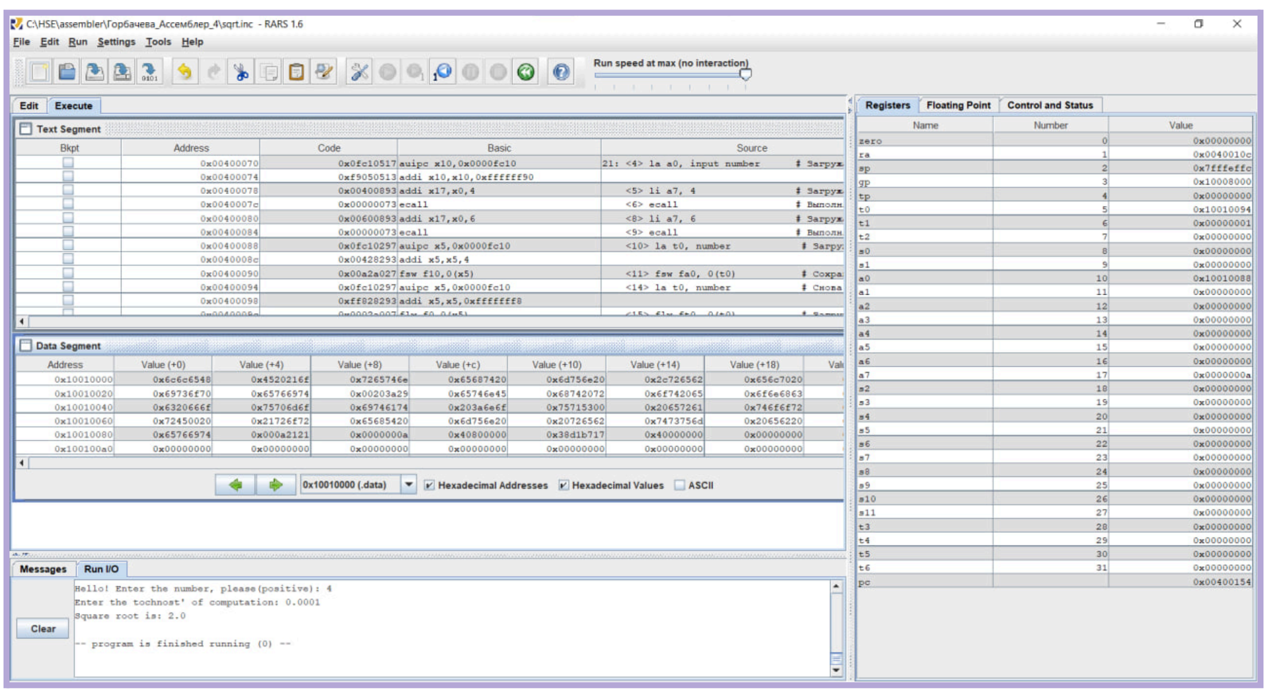
1) берём исходное число, из которого нужно извлечь корень, берём требуемую точность вычислений, я в программе гоняла разные, можно взять 0.001, например, дальше выбираем начальное "предположение" значения числа, обычно это половина числа, как видно из формулы.

2) Дальше: делим исходное число на текущее предположение (9/ (9/2) ~ 3), складываем наше текущее предположение с получившимся частным (3 + 3 = 6), и делим эту сумму пополам -теперь – это наше новое предположение.

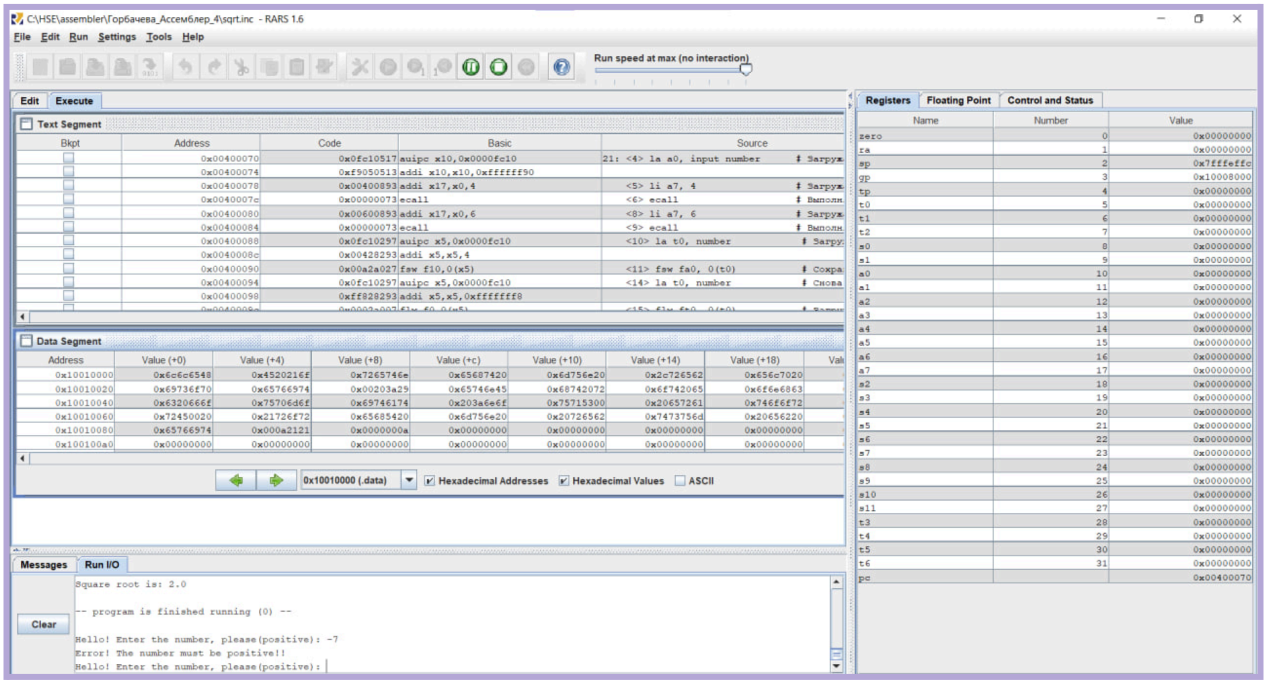
3) Потом вычисляем разницу между новым и старым предположением (3-3=0). Как видно, у нас все хорошо, однако мы пример хороший взяли, но может получиться неоднозначный результат, тогда мы берём модуль

разницы, и сравниваем с точностью: если больше точности - итеративно улучшаем предположение, если меньше - то все хорошо, нас устраивает, мы завершаем программу вычислений.
На ассемблере же мы сохраняем исходные числа и точности в переменные, и вот такой формулой `fdiv.s fs2, fs0, fs3` вычисляем начальное приближение = fs2 = a / 2.0 (x_0 = a/2). Потом идём в итерационный цикл (приводила скрин с комментариями чуть выше), а потом возвращаем результат.

Результаты запуска программы:
Сначала стоит отметить, что в формулировке дз было сказано, что необходимо сформировать несколько отдельных файлов, у меня это main.asm, macro.inc, sqrt.inc.
.inc - подключаемый неполноценный файл, как я поняла из документации, можно было оставить расширение .asm неглавным файлам(в которых реализован сам алгоритм), однако, чтобы показать, что эти файлы - несамостоятельные программы,я поменяла им расширение, так же необходимо в главный файл включить строки .include "macros.inc" и include "sqrt.inc", как и в C++, без данной команды проект не ассемблируется.



Корректный результат работы программы. А теперь необходимо проверить программу на некорректно введённые данные:



Программа вывела сообщение об ошибке и попросила ввести корректные данные, после чего снова оперативно отработала и вернула корректный результат: