Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Отчёт по лабораторной работе № 3

Дисциплина: Низкоуровневое программирование

Тема: Программирование RISC-V

Вариант: 14

Выполнил студент гр. 3530901/90002 _____С.А. Федоров
(подпись)

Принял преподаватель _____ Д.С. Степанов
(подпись)

“_____” _____ 2021 г.

Санкт-Петербург
2021

Цели работы:

1. Разработать программу на языке ассемблера RISC-V, реализующую определенную вариантом задания функциональность, отладить программу в симуляторе VSim или Jupiter. Массив (массивы) данных и другие параметры (количество итераций, счетчик результата и пр.) располагаются в памяти по фиксированным адресам.
2. Выделить определенную вариантом задания функциональность в подпрограмму, организованную в соответствии с ABI, разработать использующую ее тестовую программу. Адрес обрабатываемого массива данных и другие значения передавать через параметры подпрограммы в соответствии с ABI. Тестовая программа должна состоять из инициализирующего кода, кода завершения, подпрограммы main и тестируемой подпрограммы.

Начальные данные для 14 варианта

Определение наиболее часто встречающегося в массиве значения.

1. Постановка задачи и алгоритм решения

Требуется смоделировать программу для RISC-V, которая определит наиболее часто встречающееся значение в массиве (по заданию надо найти одно значение, поэтому если таких значений будет несколько, то возьмем наибольшее по величине).

Для реализации воспользуемся сортировкой методом пузырька, а затем в отсортированным массиве пройдемся по всем элементам и найдем наиболее часто встречающееся значение.

Для тестирования будет использован симулятор V-Sim-2.0.2.

2. Подробный алгоритм для решения поставленной задачи

1) Проверка индекса внешнего цикла сортировки, если a1 будет больше или равен a3(размер массива), значит цикл завершился – завершилась сортировка.
2) Проверка индекса внутреннего цикла, если a2 будет больше или равен a7 (a7 = N - i), то надо осуществить переход на новый круг итераций (увеличить значение индекса внешнего цикла и обновить значения a2 и a7)
3) Если индекс внутреннего цикла оказался меньше a7 = N-i, тогда осуществим проверку текущих элементов массива(x[i] и x[i-1]):
    - Если x[i-1] - x[i] окажется меньше нуля, значит элементы уже отсортированы и их не надо менять местами, то есть увеличиваем значение a2 (переменная внутреннего цикла)
    - Если окажется больше или равен нулю, значит надо осуществить перестановку элементов местами и также осуществить увеличение переменной a2
4) Начинаем итерироваться по отсортированному массиву:
    - Для начала получим индекс ячейки, где будет храниться result. Эта ячейка находится сразу после элементов массива
    - Если разность x[i-1] и x[i] оказывается меньше нуля, значит значения

отличаются и надо осуществить проверку на обновление результата (сравнить counter и resCounter – счетчики для количества текущих значений и значений результата) и осуществить обновление результата, если потребуется (counter>= resCounter). Также надо изменить значение counter на 1 и увеличить переменную a2 для итерации по массиву.

- Если разность больше или равна нулю, тогда к counter прибавляется 1 и обновляется a2

5) Осуществить проверку краевого случая (если counter> = resCounter. То следует обновить результат)

3. Документация к RISC-V

В данном пункте разберемся с теми командами и типами данных, которые будут использоваться в написании кода программы.

- **.text** – указание ассемблеру размещать последующие инструкции в секции кода
- **Start** – метка начала работы программы
- Директива **.global** указывает, что данный символ является экспортируемыми
- **la –** это инструкция для присвоения переменной адреса другой переменной
- **lw –** это инструкция для присвоения переменной значения другой переменной, путем обращения к адресу этой переменной
- **li –** это инструкция для присвоения переменной какого-то значения
- **sw –** это инструкция для того, чтобы записать значение переменной по какому-либо адресу
- **add –** это инструкция для сложения двух переменных
- **addi –** это инструкция для сложения переменной и числа
- **slli –** это инструкция для осуществления сдвига переменной влево на N
- **bgeu –** это инструкция для сравнения двух переменных if (x >= y) goto loop
- **bltu –** это инструкция для сравнения двух переменных if (x < y) goto loop
- **jal, zero loop –** это инструкция для безусловного перескока на метку loop
- **.rodata** – указание на сегмент флеш-памяти, в котором данные будут неизменны
- **.data** – указание на сегмент флеш-памяти, в котором данные могут быть

изменены

- **.word** – метка, указывающая на то, что используются 32 битные слова (4 байта)
- **t0-t6 –** это временные регистры
- **a0-a7 –** это аргументы функции, где **a0** и **a1** – это возвращаемые значения функции (используются для конструкций завершения программы)

4. Реализация программы

Программа для RISC-V, которая реализует поиск наиболее часто встречающегося значения в массиве.

```
.text
start:
.globl start
 la a3, array_length #} 
 lw a3, 0(a3) #} a3 = <длина массива>
 la a4, array # a4 = <адрес 0-го элемента массива>
 li a2, 1 # a2 = 1 # переменная для внутреннего цикла
 li a1, 0 # a1 = 1 # переменная для внешнего цикла

 la a7, array_length # переменная для хранения значения текущего N-i
 lw a7, 0(a7) #} a7 = <длина массива>
 addi a7, a7, 1 # a7 = a7 + 1 (длина массива + 1)

output_loop: # внешний цикл от 1 до N
 addi a1, a1, 1 # a1 += 1
 addi a7, a7, -1 # изменим значение N-i
 li a2, 1 # обновим a2 для нового внутреннего цикла

 bgeu a1, a3, loop_findElement # if( a1 >= a3 ) goto loop_findElement (внешний цикл закончился -> массив отсортирован)
 inner_loop: # внутренний цикл от 1 до N-i, где i - значение внешнего цикла

  bgeu a2, a7, output_loop # if( a2 >= a7 ) goto output_loop (внутренний цикл закончился -> идем на внешний цикл)

  # Получим значения x[i], где i = a5 и x[i-1], где i-1 = a6
  # каждый элемент в отдельном байте, а чтобы получить индекс след. элемента, надо увеличить текущее значение на 4
  slli a5, a2, 2 # a5 = a2 << 2 = a2 * 4
  add a5, a4, a5 # a5 = a4 + a5 = a4 + a2 * 4
  addi a6, a5, -4 # a6 = a5 + (-4) = a5 - 4
  lw t1, 0(a6) # t1 = array[i-1]
  lw t0, 0(a5) # t0 = array[i]

  # Делаем проверку для того, чтобы поменяться местами x[i] и x[i-1] элементы
  bltu t1, t0, noChange_loop # if (t1 < t0) goto noChange_loop
  sw t1, 0(a5) # array[i] = t1
  sw t0, 0(a6) # array[i-1] = t0

  noChange_loop: # эдементы местами менять не надо
   addi a2, a2, 1 # a2 += 1

  jal zero, inner_loop # goto inner_loop

loop_findElement: # массив отсортирован и теперь в нем можно найти наиболее часто встречающееся значение
```

Рис.1. Код программы first.

```
loop_findElement: # массив отсортирован и теперь в нем можно найти наиболее часто встречающееся значение
 # Сначала сделаем t2 - индекс ячейки, где находится result
 la a2, array_length # Присвоим a2 значение длины массива
 lw a2, 0(a2) #} a3 = <длина массива>
 slli t2, a2, 2 # t2 = a2 << 2 = a2 * 4 (теперь сдвинемся на следующую ячейку, где лежит result)
 add t2, a4, t2 # t2 = a4 + t2 = a4 + a2 * 4 (t2 - это индекс result)

 li a2, 1 # обновим a2
 lw t0, 0(a4) # t0 = array[0]
 sw t0, 0(t2) # result = t0 (t0 == array[0])

# Введем несколько перменных для поиска result
li a1, 1 # Создадим counter (счетчик текущего количества повторений элемента)
li a7, 1 # Создадим Rescounter (счетчик количества повторений элемента, находящегося в result)

# Теперь напишем цикл, в котором пробежимся по массиву и найдем наиболее часто встречающееся значение
loop_searchCycle:
 bgeu a2, a3, loop_endCycle # if( a2 >= a3 ) goto loop_endCycle (обошли весь массив -> завершаем работу)

 # Получим значения x[i], где i = a5 и x[i-1], где i-1 = a6
 slli a5, a2, 2 # a5 = a2 << 2 = a2 * 4
 add a5, a4, a5 # a5 = a4 + a5 = a4 + a2 * 4
 addi a6, a5, -4 # a6 = a5 + (-4) = a5 - 4
 lw t1, 0(a6) # t1 = array[i-1]
 lw t0, 0(a5) # t0 = array[i]

 # Делаем проверку на то, начались ли элементы массива с другим значением
 bltu t1, t0, replaceRes_loop # if (t1 < t0) goto replaceRes_loop
  addi a1, a1, 1 # a1 += 1 (увеличиваем значение counter, тк значение не изменилось)
  addi a2, a2, 1 # a2 += 1
  jal zero, loop_searchCycle # goto loop_searchCycle

 replaceRes_loop: # Надо сделать проверку (conter > resCounter или нет)
  bltu a1, a7, noChangeRes_loop # if (a1 < a7, что означает counter < resCounter) goto noChangeRes_loop
   # Попадаем сюда, если a1 >= a7, что означает counter >= resCounter
   sw t1, 0(t2) # result = t1 (t1 == array[i-1])
   mv a7, a1 # resCounter = counter

  noChangeRes_loop:
   li a1, 1 # counter = 1
   addi a2, a2, 1 # a2 += 1
   jal zero, loop_searchCycle # goto loop_searchCycle

loop_endCycle:
# Сделаем проверку краевого случая, когда наиболее часто встр. значение стоит в конце массива
 bltu a1, a7, loop_exit # if (a1 < a7, что означает counter < resCounter) goto loop_exit
  # Попадаем сюда, если a1 >= a7, что означает counter >= resCounter
  sw t0, 0(t2) # result = t0 (t0 == array[i])

loop_exit:
finish:
 li a0, 10 # x10 = 10
 li a1, 0 # x11 = 0
 ecall # ecall при значении x10 = 10 => останов симулятора
.rodata # помещение в секцию неизменяемых данных
array_length:
 .word 9
.data # помещение в секцию изменяемых данных
array:
 .word 4, 2, 3, 2, 1, 5, 2, 1, 1
result:
 .word 0 # запишем значение result в следующую ячейку после элементов массива (можно присовить любое значение)
```

Рис.2. Код программы first.

Теперь для воспользуемся командной строкой для запуска нашей программы.

```
>>> memory 0x10000008 3
            Value (+0) Value (+4) Value (+8) Value (+c)
[0x10000008] 0x00000004 0x00000002 0x00000003 0x00000002
[0x10000018] 0x00000001 0x00000005 0x00000002 0x00000001
[0x10000028] 0x00000001 0x00000000 0x00000000 0x00000000
```

Рис.3. Массив до начала работы программы.

Для начала проверим правильность работы сортировки пузырьком. Для этого поставим точку останова по адресу метки loop_findElement.

```
>>> breakpoint 0x10070
>>> c
>>> memory 0x10000008 3
            Value (+0) Value (+4) Value (+8) Value (+c)
[0x10000008] 0x00000001 0x00000001 0x00000001 0x00000002
[0x10000018] 0x00000002 0x00000002 0x00000003 0x00000004
[0x10000028] 0x00000005 0x00000000 0x00000000 0x00000000
```

Рис.4. Массив после сортировки.

Теперь можно поставить точку останова на метке finish, когда программа уже определит наиболее часто встречающееся значение массива и запишет его в следующую ячейку памяти, после всех элементов массива.

```
>>> breakpoint 0x100e0
>>> c
>>> memory 0x10000008 3
            Value (+0) Value (+4) Value (+8) Value (+c)
[0x10000008] 0x00000001 0x00000001 0x00000001 0x00000002
[0x10000018] 0x00000002 0x00000002 0x00000003 0x00000004
[0x10000028] 0x00000005 0x00000002 0x00000000 0x00000000
```

Рис.5. Результат работы программы.

Исходя из рис.3 – рис.5 можно сделать вывод, что программы работает правильно, так как в изначальном массиве данных все значения встречаются по одному разу, кроме значений 1 и 2, которые встречаются по три раза, но так как значение 2> 1, то оно и вывелось в результат.

Программа универсальна, так как работает для массива любого размера, но при добавлении или же удалении элементов из массива потребуется изменить значение размерности массива (array_length).

Результат записывается сразу после всех элементов массива, а в ходе работы основного цикла (строки 42-83) он перезаписывается, если находится значение, которое встречается чаще, чем то, что уже находится в result.

Сортировка методом пузырька находится с 14 по 42 строку. С 14- 17 происходит переход на новый круг внешнего цикла, а с 19 по 40 строки находится внутренний цикл, где в случае выполнения условия, элементы меняются местами и перезаписываются в памяти.

С 85-89 строку происходит проверка краевого случая, когда наиболее часто встречающееся значение является самым большим по значению элементом в исходном массиве.

5. Реализация подпрограммы

В данном пункте требуется написать тестирующую программу и подпрограмму main, в которой будет вызываться наша программа из предыдущего пункта.

```
# setup.s
.text
start:
.globl start
 call main # вызываем подпрограмму main
finish:
 li a0, 10 # x10 = 10
 ecall # ecall при значении x10 = 10 => останов симулятора
```

Рис.6. Тестирующая программа.

Псевдоинструкция call соответствует следующей паре инструкций:

```
auipc ra, %pcrel_hi(main)
jalr ra, ra, %pcrel_lo(main)
```

Исполненные одна за другой, эти инструкции обеспечивают безусловный переход (jump) на метку main с сохранением адреса следующей за jalr инструкции в регистре ra (синоним x1).

Когда выполнения подпрограммы main завершится мы перейдем на метку finish, где работы тестирующей программы завершится.

```
1  # main.s
2  .text
3  main:
4  .globl main
5   la a3, array_length #}
6   lw a3, 0(a3) #} a3 = <длина массива>
7   la a4, array # a4 = <адрес 0-го элемента массива>
8
9   la a7, array_length # переменная для хранения значения текущего N-i
10  lw a7, 0(a7) #} a7 = <длина массива>
11
12  addi sp, sp, -16 # выделение памяти в стеке
13  sw ra, 12(sp) # записываем ra (адрес возврата)
14
15  call second # вызываем нашу подпрограмму
16
17  lw ra, 12(sp) # восстанавливаем ra
18  addi sp, sp, 16 # освобождение памяти в стеке
19
20  li a0, 0
21  ret
22
23 .rodata # помещение в секцию неизменяемых данных
24 array_length:
25  .word 9
26 .data # помещение в секцию изменяемых данных
27 array:
28  .word 4, 2, 3, 2, 1, 5, 2, 1, 1
29 result:
30  .word 0 # запишем значение result
```

Рис.7. Программа main.

В подпрограмме main задаются все регистры **a3**, **a4** и **a7** за счет написания псевдоинструкций (строки 5-10). Также задаются данные массива и результата (строки 23 - 30).

В 21 строке задается обертка **ret** – возврат из подпрограммы по адресу из **ra**

В 15 строке происходит вызов подпрограммы second (программы из предыдущего пункта). Здесь, из-за того, что мы уже находимся в подпрограмме, вызов еще одной подпрограммы меняется. Поэтому надо выделить память в стеке и записать адрес возврата **ra.** После того, как second выполнится, требуется восстановить **ra** и освободить выделенную ранее память в стеке.

Пояснение из учебного пособия*:*

В случае 32-разрядной версии RISC-V для сохранения значения ra в стеке требуется только 4 байта, однако ABI RISC-V требует выравнивания указателя стека на границу 128 разрядов (16 байт), следовательно, величина изменения указателя стека должна быть кратна 16. Кроме того, в RISC-V (как и в

большинстве архитектур) стек растет вниз (grows downwards), то есть выделению памяти в стеке (stack allocation) соответствует уменьшение значения указателя стека. Отметим, что начальное значение sp устанавливается симулятором.

В ABI RISC-V регистр sp является сохраняемым, то есть при возврате из подпрограммы он должен иметь исходное значение. Поскольку для выделения памяти в стеке значение sp уменьшается (в данном случае на 16), перед возвратом из подпрограммы достаточно увеличить sp на ту же величину.

```
# second.s
.text
second:
.globl second
 # la a3, array_length #}
 # lw a3, 0(a3) #} a3 = <длина массива>
 # la a4, array # a4 = <адрес 0-го элемента массива>
 li a2, 1 # a2 = 1 # переменная для внутреннего цикла
 li a1, 0 # a1 = 1 # переменная для внешнего цикла

 # la a7, array_length # переменная для хранения значения текущего N-i
 # lw a7, 0(a7) #} a7 = <длина массива>
 addi a7, a7, 1 # a7 = a7 + 1 (длина массива + 1)

output_loop: # внешний цикл от 1 до N
 addi a1, a1, 1 # a1 += 1
 addi a7, a7, -1 # изменим значение N-i
 li a2, 1 # обновим a2 для нового внутреннего цикла

 bgeu a1, a3, loop_findElement # if( a1 >= a3 ) goto loop_findElement (внешний цикл закончился -> массив отсортирован)
 inner_loop: # внутренний цикл от 1 до N-i, где i - значение внешнего цикла

  bgeu a2, a7, output_loop # if( a2 >= a7 ) goto output_loop (внутренний цикл закончился -> идем на внешний цикл)

  # Получим значения x[i], где i = a5 и x[i-1], где i-1 = a6
  # каждый элемент в отдельном байте, а чтобы получить индекс след. элемента, надо увеличить текущее значение на 4
  slli a5, a2, 2 # a5 = a2 << 2 = a2 * 4
  add a5, a4, a5 # a5 = a4 + a5 = a4 + a2 * 4
  addi a6, a5, -4 # a6 = a5 + (-4) = a5 - 4
  lw t1, 0(a6) # t1 = array[i-1]
  lw t0, 0(a5) # t0 = array[i]

  # Делаем проверку для того, чтобы поменяться местами x[i] и x[i-1] элементы
  bltu t1, t0, noChange_loop # if (t1 < t0) goto noChange_loop
  sw t1, 0(a5) # array[i] = t1
  sw t0, 0(a6) # array[i-1] = t0

  noChange_loop: # эдементы местами менять не надо
   addi a2, a2, 1 # a2 += 1

  jal zero, inner_loop # goto inner_loop

loop_findElement: # массив отсортирован и теперь в нем можно найти наиболее часто встречающееся значение
```

Рис.8. Код подпрограммы second.

```
loop_findElement: # массив отсортирован и теперь в нем можно найти наиболее часто встречающееся значение
 # Сначала сделаем t2 - индекс ячейки, где находится result
 # la a2, array_length # Присвоим a2 значение длины массива
 mv a2, a3 #} a2 = <длина массива>
 slli t2, a2, 2 # t2 = a2 << 2 = a2 * 4 (теперь сдвинемся на следующую ячейку, где лежит result)
 add t2, a4, t2 # t2 = a4 + t2 = a4 + a2 * 4 (t2 - это индекс result)

 li a2, 1 # обновим a2
 lw t0, 0(a4) # t0 = array[0]
 sw t0, 0(t2) # result = t0 (t0 == array[0])

# Введем несколько перменных для поиска result
li a1, 1 # Создадим counter (счетчик текущего количества повторений элемента)
li a7, 1 # Создадим Rescounter (счетчик количества повторений элемента, находящегося в result)

# Теперь напишем цикл, в котором пробежимся по массиву и найдем наиболее часто встречающееся значение
loop_searchCycle:
 bgeu a2, a3, loop_endCycle # if( a2 >= a3 ) goto loop_endCycle (обошли весь массив -> завершаем работу)

 # Получим значения x[i], где i = a5 и x[i-1], где i-1 = a6
 slli a5, a2, 2 # a5 = a2 << 2 = a2 * 4
 add a5, a4, a5 # a5 = a4 + a5 = a4 + a2 * 4
 addi a6, a5, -4 # a6 = a5 + (-4) = a5 - 4
 lw t1, 0(a6) # t1 = array[i-1]
 lw t0, 0(a5) # t0 = array[i]

 # Делаем проверку на то, начались ли элементы массива с другим значением
 bltu t1, t0, replaceRes_loop # if (t1 < t0) goto replaceRes_loop
  addi a1, a1, 1 # a1 += 1 (увеличиваем значение counter, тк значение не изменилось)
  addi a2, a2, 1 # a2 += 1
  jal zero, loop_searchCycle # goto loop_searchCycle

 replaceRes_loop: # Надо сделать проверку (conter > resCounter или нет)
  bltu a1, a7, noChangeRes_loop # if (a1 < a7, что означает counter < resCounter) goto noChangeRes_loop
   # Попадаем сюда, если a1 >= a7, что означает counter >= resCounter
   sw t1, 0(t2) # result = t1 (t1 == array[i-1])
   mv a7, a1 # resCounter = counter

  noChangeRes_loop:
   li a1, 1 # counter = 1
   addi a2, a2, 1 # a2 += 1
   jal zero, loop_searchCycle # goto loop_searchCycle

loop_endCycle:
# Сделаем проверку краевого случая, когда наиболее часто встр. значение стоит в конце массива
 bltu a1, a7, loop_exit # if (a1 < a7, что означает counter < resCounter) goto loop_exit
  # Попадаем сюда, если a1 >= a7, что означает counter >= resCounter
  sw t0, 0(t2) # result = t0 (t0 == array[i])

loop_exit:
 ret
```

Рис.9. Код подпрограммы second.

Рис.8 и рис.9 – это программы из предыдущего пункта, только модифицированная в подпрограмму, в которой удалены строчки с объявлением регистров за счет псевдоинструкций. Также после метки loop_exit убраны строчки с заданием массива и добавлена обертка ret, которая нужна для выхода из подпрограммы.

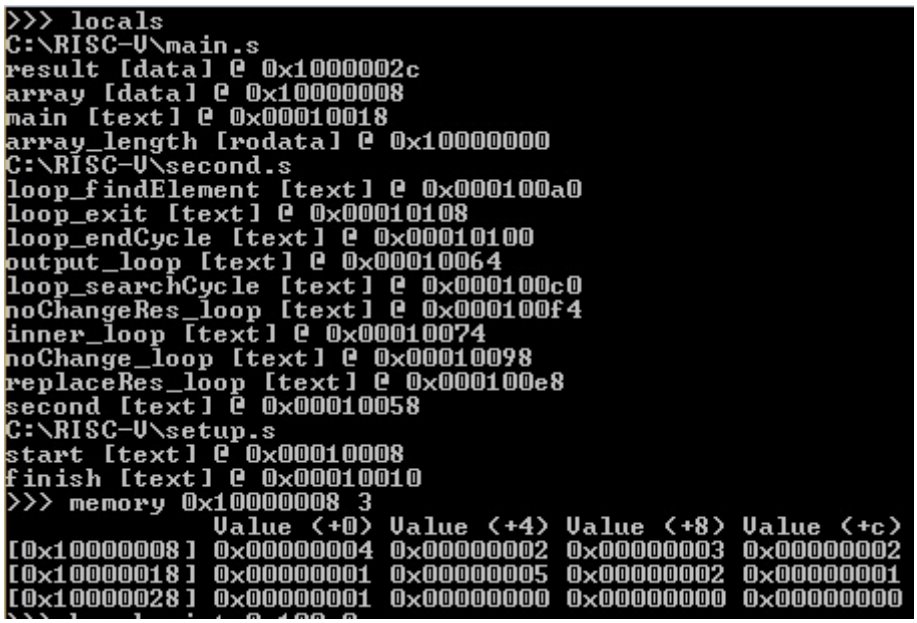

Рис.10. Массив значений до начала работы программы и значения адресов.

Для начала проверим правильность работы сортировки пузырьком. Для этого поставим точку останова по адресу метки loop_findElement.

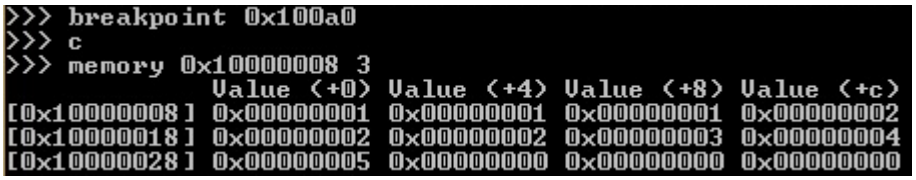

Рис.11. Массив после сортировки.

Теперь можно поставить точку останова на метке finish, когда программа уже определит наиболее часто встречающееся значение массива и запишет его в следующую ячейку памяти, после всех элементов массива.

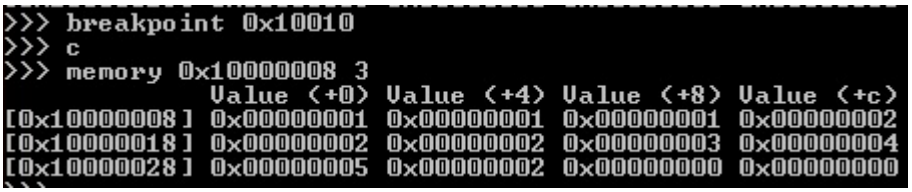

Рис.12. Результат работы программы.

Исходя из рис.11 – рис.12 можно сделать вывод, что подпрограммы вызывается и работает правильно, так как массив сортируется по возрастанию и потом в нем определяется наибольшее чаще всего встречающееся значение (result записывается в следующую ячейку после всех элементов массива).

6. Вывод

В данной лабораторной работе я познакомился с принципом работы RISC-V с написанием программы и реализацией вызова подпрограммы из программы, в частности была написана программы для поиска наиболее часто встречающегося в массиве значения.

Список использованных источников

http://kspt.icc.spbstu.ru/media/files/2020/lowlevelprog/riscv_prgc.pdf

https://m.habr.com/ru/post/533272/

http://kspt.icc.spbstu.ru/media/files/2020/lowlevelprog/riscv_subprgc.pdf

http://kspt.icc.spbstu.ru/media/files/2020/lowlevelprog/lab3.pdf