

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 на RISC-V

Вариант: 15

Выполнил студент гр. 3530901/90002 _____ З.А. Фрид
(подпись)

Принял преподаватель _____ Д.С. Степанов
(подпись)

“ ____ ” _____ 2021 г.

Санкт-Петербург
2021

Постановка задачи

1. Разработать программу на языке ассемблера RISC-V, реализующую определенную вариантом задания функциональность, отладить программу в симуляторе VSim/Jupiter. Массив (массивы) данных и другие параметры (преобразуемое число, длина массива, параметр статистики 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.

Вариант задания

Согласно варианту 15, необходимо реализовать слияние двух отсортированных массивов.

Алгоритм

1. Сравнение элемента $A(i)$ и $B(j)$;
2. Если $A(i) > B(j)$, то в результирующий массив печатается $B(j)$, а для следующего прохода в массиве B берется следующий элемент, $A(i)$ остается прежним;
3. Если $B(j) \geq A(i)$, то в итоговый массив печатается $A(i)$, а для следующего прохода в массиве A берется следующий элемент, $B(j)$ остается прежним;
4. Цикл 1-3 повторяется до тех пор, пока в массивах A и/или B будут оставаться необработанные элементы.

Реализация программы

Есть два входных массива (arrayA и arrayB) и один выходной (arrayR).

Вначале мы берем адреса первых элементов. В цикле сравниваем текущие элементы массивов arrayA и arrayB. Если элемент массива arrayA больше элемента массива arrayB, то в итоговый массив arrayR записывается элемент массива arrayB и увеличиваем указатели массива arrayB и итогового массива arrayR. И наоборот. Повторяем цикл до тех пор, пока во входных массивах не останется необработанных элементов.

Текст программы

```
1 .text
2 start:
3 .globl start
4 la a0, arrayR          # a0 = address R[0]
5 lw a1, arrayA_length   # a1 = len(A)
6 lw a2, arrayB_length   # a2 = len(B)
7 la a3, arrayA          # a3 = address a[0]
8 la a4, arrayB          # a4 = address b[0]
9 li a5, 0               # counter for A = i
10 li a6, 0              # counter for B = j
11 jal zero, check       # goto check; jump to check
12 loop:
13 lw t0, 0(a3)          # t0 = a3 = a[i]
14 lw t1, 0(a4)          # t1 = a4 = b[j]
15 bgeu a5, a1, printB   # if(i >= len(A)); all arrayA elements passed
16 bgeu a6, a2, printA   # if(j >= len(B)); all arrayB elements passed
17 bgtu t0, t1, printB   # if(t0 > t1) => (a[i] > b[j]) goto printB
18 printA:
19 sw t0, 0(a0)          # r[k] = a[i]; writing a[i] to the resulting arrayR
20 addi a5, a5, 1        # i++; increase counter for A
21 addi a3, a3, 4        # a3 = a3 + 4 = address a[i] + 4 = address a[i+1]
22 jal zero, increaseR   # goto increaseR; jump to increaseR
23 printB:
24 sw t1, 0(a0)          # r[k] = b[j]; writing b[j] to the resulting arrayR
25 addi a6, a6, 1        # j++; increase counter for B
26 addi a4, a4, 4        # a4 = a4 + 4 = address b[j] + 4 = address b[j+1]
27 increaseR:
28 addi a0, a0, 4        # a0 = a0 + 4 = address r[k] + 4 = address r[k+1]
29 check:
30 bltu a5, a1, loop     # if there are unprocessed elements in arrayA then goto loop
31 bltu a6, a2, loop     # if there are unprocessed elements in arrayB then goto loop
32 loop_exit:
33 finish:
34 li a0, 10             # shutting down the program
35 ecall
36
37 .rodata               # read-only data; immutable data
38 arrayA_length:
39 .word 8
40 arrayA:
41 .word 1, 3, 4, 5, 5, 6, 9, 10
```

```

42 arrayB_length:
43 .word 4
44 arrayB:
45 .word 2, 3, 5, 7
46
47 .data          # data; mutable data
48 arrayR:
49 .word 255

```

Руководство

.text - указание ассемблеру размещать последующие инструкции в секции кода.

Метка **start:** - точка начала выполнения программы.

В строках 4-10 написаны псевдоинструкции установки значений регистров **a0-a6**. Псевдоинструкция транслируется ассемблером в последовательность инструкций (**instruction**) системы команд RISC-V, обеспечивающую выполнение требуемого действия. В **a0** хранится адрес **k**-го элемента массива **arrayR** (в самом начале туда записывается адрес первого элемента). В **a1** и **a2** хранятся длины массивов **arrayA** и **arrayB** соответственно. В **a3** и **a4** хранятся адреса **i**-го и **j**-го элементов массивов **arrayA** и **arrayB** соответственно (в самом начале туда записываются адреса первых элементов).

Основная часть программы написана на строках 11-35. **jal zero, check** – оператор, осуществляющий безусловный переход к метке **check**. В метке **check** происходит проверка на наличие необработанных элементов массива с помощью оператора **bltu r1, r2, addr**, который осуществляет переход на метку **addr**, если **r1 < r2**. Таким образом мы проверяем не превысило ли число обработанных элементов массива длину массива.

Из метки **check** при выполнении условий осуществляется переход к метке **loop**. Перейдя в метку **loop**, с помощью команд **lw t0, 0(a3)** и **lw t1, 0(a4)** мы записываем значения из ячеек по адресам **a3** и **a4** в регистры **t0** и **t1** соответственно. Оператор **bgeu r1, r2, addr** осуществляет переход на метку **addr**, если **r1 >= r2**. Таким образом, если, например, были пройдены все элементы массива **arrayA**, но не все элементы массива **arrayB**, то переходим к печати элемента массива **arrayB**, то есть к метке **printB**. И наоборот, переход к

метке **printA**, если пройдены все элементы массива **arrayB**. Оператор **bgtu r1, r2, addr** осуществляет переход на метку **addr**, если $r1 < r2$. В нашем случае, если элемент массива **arrayA** больше элемента массива **arrayB**, то переходим к печати элемента массива **arrayB**, то есть к метке **printB**. Иначе переходим на следующую строку к метке **printA**.

В метке **printB** печатаем элемент массива **arrayB** в итоговый массив **arrayR** с помощью команды **sw t1, 0(a0)**, которая записывает в ячейку по адресу **a0** значение ячейки **t1**. Затем увеличиваем счетчик для массива **arrayB** и увеличиваем указатель на текущий элемент массива **arrayB** на 4, так как элементы массива занимают 4 байта.

Аналогично написана метка **printA**. Еще добавлен переход к метке **increaseR (jal zero, increaseR)**, так как из метки **printB** переход к метке **increaseR** произойдет самостоятельно. В метке **increaseR** увеличиваем указатель на текущий элемент массива **arrayR** на 4, так как элементы массива занимают 4 байта. Затем идет вышеупомянутая метка **check**. Если условия в ней не выполнены, то есть все элементы входных массивов обработаны, то переходим к меткам **loop_exit** и **finish**, где происходит завершение работы программы.

Сами данные определяются в конце программы (37 – 49 строки). **.word** означает, что мы используем 32-битные слова (4 байта). То есть они занимают 4 восьмибитных секции. **.rodata** – неизменяемые данные, **.data** – изменяемые данные.

Пример выполнения программы:

Запустим нашу программу через консоль. С помощью команды **locals** посмотрим перечень символов и их значений:

```
>>> locals
C:\Учеба\НизУрПрога\3\3 лаба (RISC-V)\arraysMerge.s
printA [text] @ 0x00010050
printB [text] @ 0x00010060
loop_exit [text] @ 0x00010078
start [text] @ 0x00010008
arrayB [rodata] @ 0x10000028
arrayA [rodata] @ 0x10000004
check [text] @ 0x00010070
arrayB_length [rodata] @ 0x10000024
increaseR [text] @ 0x0001006c
arrayA_length [rodata] @ 0x10000000
loop [text] @ 0x0001003c
arrayR [data] @ 0x1000003c
finish [text] @ 0x00010078
```

С помощью команды **memory** проверим состояние массивов **arrayA** и **arrayB**:

```
>>> memory 0x10000004 2
      Value (+0) Value (+4) Value (+8) Value (+c)
[0x10000004] 0x00000001 0x00000003 0x00000004 0x00000005
[0x10000014] 0x00000005 0x00000006 0x00000009 0x0000000a
>>> memory 0x10000028 1
      Value (+0) Value (+4) Value (+8) Value (+c)
[0x10000028] 0x00000002 0x00000003 0x00000005 0x00000007
```

На рисунке видно, что данные в ячейках соответствуют введенным.

Аналогично сделаем для массива **arrayR**:

```
>>> memory 0x1000003c 3
      Value (+0) Value (+4) Value (+8) Value (+c)
[0x1000003c] 0x000000ff 0x00000000 0x00000000 0x00000000
[0x1000004c] 0x00000000 0x00000000 0x00000000 0x00000000
[0x1000005c] 0x00000000 0x00000000 0x00000000 0x00000000
```

Установим breakpoint на **finish** и запустим программу:

```
>>> breakpoint 0x00010078
>>> list
Breakpoints:

    0x00010078
>>> c
```

Проверим содержимое массива **arrayR** на конец выполнения программы:

```
>>> memory 0x1000003c 3
      Value (+0) Value (+4) Value (+8) Value (+c)
[0x1000003c] 0x00000001 0x00000002 0x00000003 0x00000003
[0x1000004c] 0x00000004 0x00000005 0x00000005 0x00000005
[0x1000005c] 0x00000006 0x00000007 0x00000009 0x0000000a
```

Таким образом, мы видим, что программа работает корректно и образуется массив 1, 2, 3, 3, 4, 5, 5, 5, 6, 7, 9, 10 из массивов 1, 3, 4, 5, 5, 6, 9, 10 и 2, 3, 5, 7.

Реализация подпрограммы

Код программы и руководство

Тестирующая программа:

```
1 # setup.s
2 .text
3 start:
4 .globl start
5 call main
6 finish:
7 li a0, 10
8 ecall
```

В программе вызывается подпрограмма `main` с помощью команды **call**.

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 выполнение подпрограммы завершится, исполнение кода тестирующей программы перейдет к метке `finish`, в которой работа программы завершается.

Подпрограмма main:

```
1 # main.s
2 .text
3 main:
4 .globl main
5 la a0, arrayR           # a0 = address R[0]
6 lw a1, arrayA_length    # a1 = len(A)
7 lw a2, arrayB_length    # a2 = len(B)
8 la a3, arrayA            # a3 = address a[0]
9 la a4, arrayB            # a4 = address b[0]
10
11 addi sp, sp, -16        # allocating memory on the stack
12 sw ra, 12(sp)           # saving ra
13
14 call merge
15
16 lw ra, 12(sp)           # recovery ra
17 addi sp, sp, 16         # freeing memory on the stack
18
19 li a0, 0
20 ret
21
```

```

22 .rodata                                # read-only data; immutable data
23 arrayA_length:
24 .word 8
25 arrayA:
26 .word 1, 3, 4, 5, 5, 6, 9, 10
27 arrayB_length:
28 .word 4
29 arrayB:
30 .word 2, 3, 5, 7
31
32 .data                                # data; mutable data
33 arrayR:
34 .word 255

```

Здесь задаются массивы и их длины, в строках 5-9 написаны псевдоинструкции установки значений регистров **a0-a4**. После этого происходит вызов подпрограммы **merge**. Однако ее вызов отличается от вызова **main**. Это связано с тем, что мы уже находимся в подпрограмме, а регистр с кодом возврата (**ra**) один, поэтому перед тем, как мы вызовем еще одну подпрограмму, нам нужно где-то сохранить этот код.

Пояснение:

*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.*

ret - возврат из подпрограммы.

Подпрограмма merge:

```
1 # merge.s
2 .text
3 merge:
4 .globl merge
5 # in a0 - address R[k]
6 # in a1 - len(A)
7 # in a2 - len(B)
8 # in a3 - address a[i]
9 # in a4 - address b[j]
10 li a5, 0 # counter for A = i
11 li a6, 0 # counter for B = j
12 jal zero, check # goto check; jump to check
13 loop:
14 lw t0, 0(a3) # t0 = a3 = a[i]
15 lw t1, 0(a4) # t1 = a4 = b[j]
16 bgeu a5, a1, printB # if(i >= len(A)); all arrayA elements passed
17 bgeu a6, a2, printA # if(j >= len(B)); all arrayB elements passed
18 bgtu t0, t1, printB # if(t0 > t1) => (a[i] > b[j]) goto printB
19 printA:
20 sw t0, 0(a0) # r[k] = a[i]; writing a[i] to the resulting arrayR
21 addi a5, a5, 1 # i++; increase counter for A
22 addi a3, a3, 4 # a3 = a3 + 4 = address a[i] + 4 = address a[i+1]
23 jal zero, increaseR # goto increaseR; jump to increaseR
24 printB:
25 sw t1, 0(a0) # r[k] = b[j]; writing b[j] to the resulting arrayR
26 addi a6, a6, 1 # j++; increase counter for B
27 addi a4, a4, 4 # a4 = a4 + 4 = address b[j] + 4 = address b[j+1]
28 increaseR:
29 addi a0, a0, 4 # a0 = a0 + 4 = address r[k] + 4 = address r[k+1]
30 check:
31 bltu a5, a1, loop # if there are unprocessed elements in arrayA then goto loop
32 bltu a6, a2, loop # if there are unprocessed elements in arrayB then goto loop
33 loop_exit:
34 ret
```

Это программа из первого пункта данной работы, оформленная в подпрограмму. Единственное, что значения в регистрах устанавливаются до её вызова, а в конце не завершение, а выход из подпрограммы с помощью **ret**.

Пример выполнения программы:

Запустим нашу программу через консоль записав названия всех трех файлов. С помощью команды **locals** посмотрим перечень символов и их значений:

```
>>> locals
C:\Учеба\НизУрПрога\3\3 лаба (RISC-V)\setup.s
start [text] @ 0x00010008
finish [text] @ 0x00010010
C:\Учеба\НизУрПрога\3\3 лаба (RISC-V)\merge.s
increaseR [text] @ 0x0001009c
```

```

printA [text] @ 0x00010080
printB [text] @ 0x00010090
loop [text] @ 0x0001006c
merge [text] @ 0x00010060
loop_exit [text] @ 0x000100a8
check [text] @ 0x000100a0
C:\Учеба\НизУрПрога\3\3 лаба (RISC-V)\main.s
arrayA_length [rodata] @ 0x10000000
arrayB [rodata] @ 0x10000028
arrayR [data] @ 0x1000003c
arrayA [rodata] @ 0x10000004
main [text] @ 0x00010018
arrayB_length [rodata] @ 0x10000024

```

В перечне значений мы видим символы всех трех вызванных файлов.

С помощью команды `memory` проверим состояние массивов `arrayA` и `arrayB`:

```

>>> memory 0x10000004 2
      Value (+0) Value (+4) Value (+8) Value (+c)
[0x10000004] 0x00000001 0x00000003 0x00000004 0x00000005
[0x10000014] 0x00000005 0x00000006 0x00000009 0x0000000a
>>> memory 0x10000028 1
      Value (+0) Value (+4) Value (+8) Value (+c)
[0x10000028] 0x00000002 0x00000003 0x00000005 0x00000007

```

На рисунке видно, что данные в ячейках соответствуют введенным.

Аналогично сделаем для массива `arrayR`:

```

>>> memory 0x1000003c 3
      Value (+0) Value (+4) Value (+8) Value (+c)
[0x1000003c] 0x000000ff 0x00000000 0x00000000 0x00000000
[0x1000004c] 0x00000000 0x00000000 0x00000000 0x00000000
[0x1000005c] 0x00000000 0x00000000 0x00000000 0x00000000

```

Установим breakpoint на `finish` и запустим программу:

```

>>> breakpoint 0x00010010
>>> list
Breakpoints:

    0x00010010
>>> c

```

Проверим содержимое массива `arrayR` на конец выполнения программы:

```

>>> memory 0x1000003c 3
      Value (+0) Value (+4) Value (+8) Value (+c)
[0x1000003c] 0x00000001 0x00000002 0x00000003 0x00000003
[0x1000004c] 0x00000004 0x00000005 0x00000005 0x00000005
[0x1000005c] 0x00000006 0x00000007 0x00000009 0x0000000a

```

Таким образом, мы видим, что программа работает корректно и образуется массив 1, 2, 3, 3, 4, 5, 5, 5, 6, 7, 9, 10 из массивов 1, 3, 4, 5, 5, 6, 9, 10 и 2, 3, 5, 7.

Вывод

В ходе выполнения лабораторной работы была реализована программа на RISC-V, реализующая слияние двух отсортированных массивов, и работающая корректно. Также эта программа была представлена, как подпрограмма, из-за чего её можно использовать несколько раз в других программах.

Список использованных источников

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

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

<https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md>

<https://habr.com/ru/post/533272/>