

Отчёт по лабораторной работе № 2

Дисциплина: Низкоуровневое программирование

Тема: программирование RISC-V

Вариант: 5

Выполнил студент гр. 3530901/90002 _____ Е. В. Бурков
(подпись)

Принял преподаватель _____ Д. С. Степанов
(подпись)

“ _____ ” _____ 2021 г.

Формулировка задачи

1. Разработать программу на языке ассемблера RISC-V, реализующую определенную вариантом задания функциональность, отладить программу в симуляторе VSim/Jupiter. Массив 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

По варианту номер 5 необходимо реализовать сортировку обменом чисел in-place. Сортировка обменом является простым алгоритмом. Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются $N - 1$ раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции, как пузырёк в воде — отсюда и название алгоритма).

Первая часть задания

Согласно заданию, была написана программа, сортирующая массив в заданной ячейке памяти. Код представлен в приложении 1. Приведём описание работы кода.

Для данной части задания начальной меткой (Start label) выбрана *start*. Массив данных расположен в секции **data**, а числа массива представлены смежными 4-байтными словами (*word*). Так же в секции **read only data** расположен размер массива в *word*. В начале программы исполняются следующие строки:

```
la a1, length
lw a1, 0(a1)
slli a1, a1, 2
la a0, array
add a1, a1, a0
li a3, 1
```

В первой строке мы записываем в регистр *x11* (далее *a1*) адрес ячейки, где хранится длина массива в 4-байтном слове. После выполняется загрузка значения длины массива в регистр *a1*. Сдвигаем число на 2 разряда влево (умножение на 4) – это сделано из-за того, что числа хранятся в *word* и чтобы итерироваться по ним надо переходить не через 1 адрес, а через 4. Далее загружаем в регистр *x10* (далее *a0*) адрес начала массива и в следующей инструкции добавляем его к регистру *a1*. Тем самым мы получили адрес последнего элемента массива (будем использовать для условия выхода из цикла). Так же загружаем в регистр *x13* (далее *a3*) единицу, позже будет пояснено для чего это сделано.

Напишем внешний цикл для сортировки.

```
addi a1, a1, -4
bed a1, a0, finish
outter_loop:
    beqz a3, finish
    li a3, 0
    la a0, array
    <
    inner_loop:
    >
    addi a1, a1, -4
    bne a1, a0, outter_loop
```

В самом начале проверим, не равно ли длина массива единице, это поможет избежать в будущем нагромождения псевдоинструкции в цикле. В первой псевдоинструкции “цикла” (транслируется как проверка на равенство с регистром zero в котором хранится ноль) происходит проверка регистра *a3* на ноль. Если он содержит ноль, то выполнение программы завершается. Регистр *a3* мы будем использовать как метку – происходили ли обмены во время итерации? Если обменов не было, то завершаем сортировку.

Далее загружаем в *a3* ноль, в *a0* адрес начала массива ($j = 0$). Выполняем некоторые действия во внутреннем цикле и после уменьшаем адрес в регистре *a1* на 4 (уменьшение i на 1) и проверяем – если *a1* и *a0* не равны, то переходим в начало цикла, если равны, то выполняются инструкции ниже (выход).

Перейдём к содержанию внутреннего цикла:

```
inner_loop:
    lw t2, 0(a0)
    lw t3, 4(a0)
    bltu t2,t3,skip_swap
    li a3, 1
    sw t3, 0(a0)
    sw t2, 4(a0)
skip_swap:
    addi a0, a0, 4
    bltu a0, a1, inner_loop
```

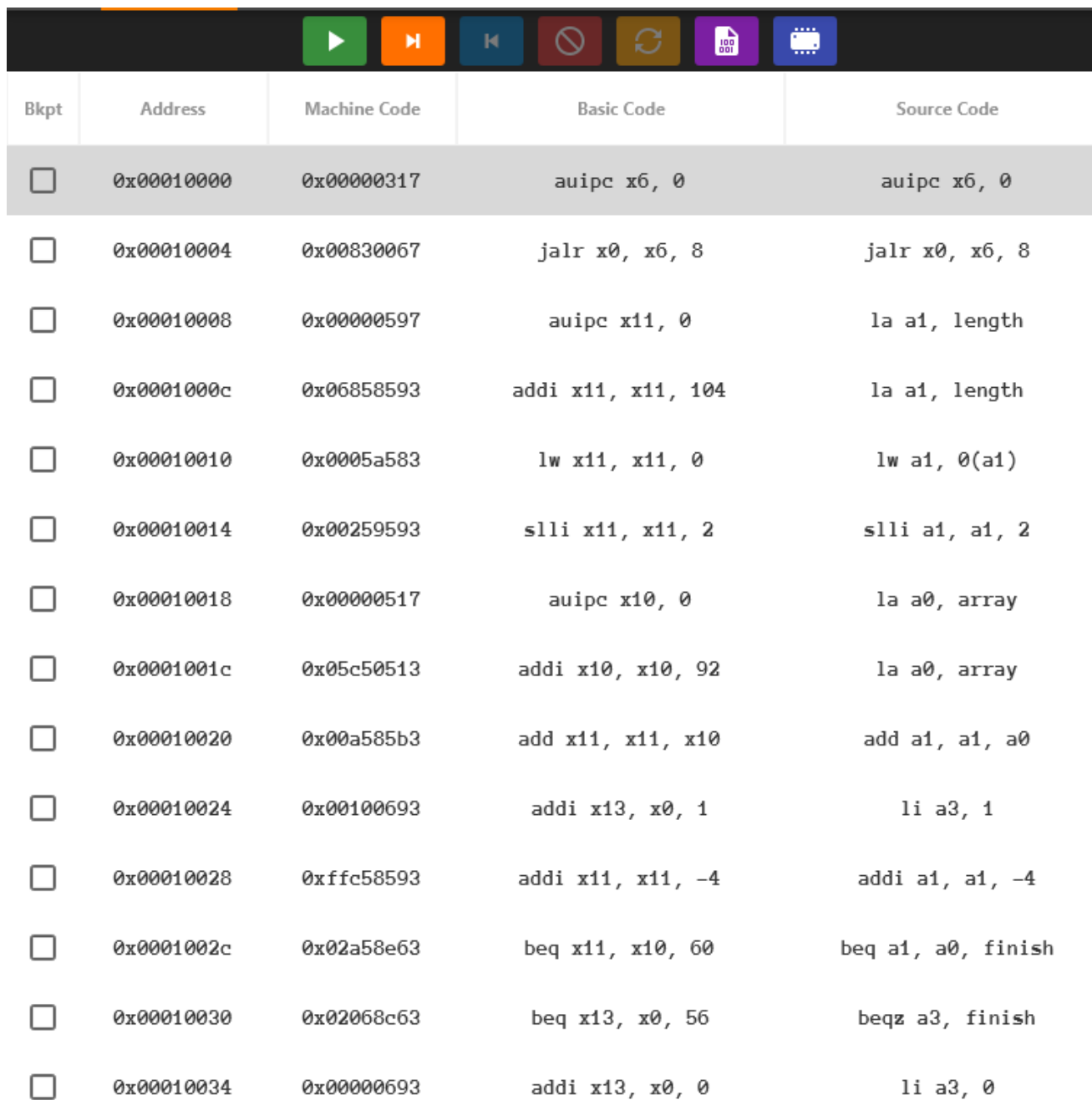
В первых двух инструкциях загружаем значения $a[j]$ в регистр $t2$, и $a[j+1]$ в регистр $t3$. Далее сравним числа, и если значение $t2$ больше значения $t3$, то производим обмен (через инструкции store word). Не забываем установить в $a3$ не ноль. После увеличиваем указатель j на 1 и если указатель меньше условия выхода, то продолжаем работу цикла.

```
finish:
    li a0, 10
    ecall
```

В конце программы, на метке “*finish*” выходим из программы при помощи ecall.

Тестирование программы

Отладка и тестирование написанной программы выполнялись при помощи симулятора *Jupiter* с графической оболочкой. Запустим нашу программу и посмотрим, как транслировались наши инструкции.



The screenshot shows the Jupiter simulator interface. At the top is a toolbar with icons for play, step over, step back, stop, refresh, memory view, and CPU view. Below the toolbar is a table with five columns: Bkpt, Address, Machine Code, Basic Code, and Source Code. The table contains 14 rows of instruction data.

Bkpt	Address	Machine Code	Basic Code	Source Code
<input type="checkbox"/>	0x00010000	0x00000317	auipc x6, 0	auipc x6, 0
<input type="checkbox"/>	0x00010004	0x00830067	jalr x0, x6, 8	jalr x0, x6, 8
<input type="checkbox"/>	0x00010008	0x00000597	auipc x11, 0	la a1, length
<input type="checkbox"/>	0x0001000c	0x06858593	addi x11, x11, 104	la a1, length
<input type="checkbox"/>	0x00010010	0x0005a583	lw x11, x11, 0	lw a1, 0(a1)
<input type="checkbox"/>	0x00010014	0x00259593	slli x11, x11, 2	slli a1, a1, 2
<input type="checkbox"/>	0x00010018	0x00000517	auipc x10, 0	la a0, array
<input type="checkbox"/>	0x0001001c	0x05c50513	addi x10, x10, 92	la a0, array
<input type="checkbox"/>	0x00010020	0x00a585b3	add x11, x11, x10	add a1, a1, a0
<input type="checkbox"/>	0x00010024	0x00100693	addi x13, x0, 1	li a3, 1
<input type="checkbox"/>	0x00010028	0xffc58593	addi x11, x11, -4	addi a1, a1, -4
<input type="checkbox"/>	0x0001002c	0x02a58e63	beq x11, x10, 60	beq a1, a0, finish
<input type="checkbox"/>	0x00010030	0x02068c63	beq x13, x0, 56	beqz a3, finish
<input type="checkbox"/>	0x00010034	0x00000693	addi x13, x0, 0	li a3, 0

Рис. 1 Симулятор Jupiter

Можно заметить, что исполнение начинается не с адреса *0x00010008* (где должны храниться инструкции в секции *.text*) а с *0x00010000* с двумя инструкциями, которые отвечают за переход к метке, указанной как “*Start label*”

Перейдём к разделу Memory – data и пронаблюдаем наш массив.

0x00010098	0	0	0	4
0x00010094	0	0	0	42
0x00010090	0	0	0	23
0x0001008c	0	0	0	12
0x00010088	0	0	0	0
0x00010084	0	0	0	56
0x00010080	0	0	0	42
0x0001007c	0	0	0	51
0x00010078	0	0	0	1
0x00010074	0	0	0	5
0x00010070	0	0	0	10

Рис. 2 Отображение содержимого памяти в разделе data

Видно, что длина массива записалась в *0x00010070*, а в следующих адресах записаны наши четырёхбайтные слова в числе в младшем байте. Выполним инструкции в начале программы и посмотрим какой адрес запишется в регистр *a1* (конец массива).

a1	x11	0x00010098
----	-----	------------

Рис. 3 Регистр a1

В данный регистр действительно записывается адрес последнего элемента массива, значит его составление выполнено верно. Совершим один обход по внешнему циклу и посмотрим на то, как изменится массив.

0x00010098	0	0	0	56
0x00010094	0	0	0	4
0x00010090	0	0	0	42
0x0001008c	0	0	0	23
0x00010088	0	0	0	12
0x00010084	0	0	0	0
0x00010080	0	0	0	51
0x0001007c	0	0	0	42
0x00010078	0	0	0	5
0x00010074	0	0	0	1

Рис. 4 Содержание массива после одной итерации по внешнему циклу

Нетрудно заметить, что самое большое число перешло в конец массива, что соответствует сортировке пузырьком. Продолжим выполнение программы до конца.

0x00010098	0	0	0	56
0x00010094	0	0	0	51
0x00010090	0	0	0	42
0x0001008c	0	0	0	42
0x00010088	0	0	0	23
0x00010084	0	0	0	12
0x00010080	0	0	0	5
0x0001007c	0	0	0	4
0x00010078	0	0	0	1
0x00010074	0	0	0	0
0x00010070	0	0	0	10

Рис. 5 Отсортированный массив

Сортировка выполнена и был совершён выход из программы.

Если массив уже отсортирован, то выполнится выход из программы. Сортировка с указанием длины массива 1 завершиться без передвижения элементов в памяти.

Выделение сортировки в подпрограмму

Согласно заданию, была написана программа. Листинг находится в приложении 2. Была составлена основная программа *start*, которая вызывает тестовую программу *main*, а та в свою очередь вызывает *sort*. Так как при исполнении псевдоинструкции **call** в регистр *ra* записывается адрес возврата для инструкции **ret**, а мы вызываем **call** 2 раза и возврат в *start* не выполняется, нам нужно использовать стек. Алгоритм следующий:

- Выделить память в стеке
- Записать адрес возврата в стек
- Использовать **call** (регистр *ra* примет другое значение)
- Взять из стека адрес возврата
- Освободить память в стеке
- Использовать восстановленный адрес возврата

Так и было сделано в данном проекте.

Адрес массива и массив передаются через регистры *a0*, *a1*. Сами значения записаны в тестовой программе.

При тестировании программы был выполнен повторный вызов сортировки и никаких ошибок не выявлено.

Программа организована в соответствии с **ABI**.

Вывод

В ходе выполнения данной лабораторной работы был получен опыт программирования на *RISC-V*. Была написана программа сортировки чисел обменом *in-place*. Так же данная программа была выделена как подпрограмма и вызвана несколько раз в тестовой программе. Вызов подпрограммы выполняется согласно соглашениям *ABI*. Весь исходный код предоставлен в репозитории на GitHub.

Приложение 1

P.S. Вставляю картинки, потому что текстом слишком убого выглядит.

```
1 .text
2
3 .global start
4
5 start:
6
7 la a1, length # a1 = adress of length
8 lw a1, 0(a1) # a1 = length value
9 slli a1, a1, 2 # a1 = length value << 2 (multiply by 4 for addressing of words)
10 la a0, array # a0 = array address
11 add a1, a1, a0 # a1 = end of array address
12 li a3, 1
13
14 addi a1, a1, -4 # I-- for exit
15 beq a1, a0, finish # if lenght = 1
16
17 # now we can use a0
18
19 #a0 = address
20 #a1 = end
21 outter_loop:
22     beqz a3, finish # if = zero
23     li a3, 0 # F, 1 if unsorted, 0 if sorted
24     la a0, array
25
26     inner_loop:
27         lw t2, 0(a0) # value of a[j]
28         lw t3, 4(a0) # value of a[j+1]
29
30         bltu t2, t3, skip_swap # if a[j] < a[j+1] -> skip
31         li a3, 1 # F = 1 (array is unsorted)
32         sw t3, 0(a0) # \ S A
33         sw t2, 4(a0) # / W P
34
35         skip_swap:
36             addi a0, a0, 4 # like j++
37             bltu a0, a1, inner_loop # if j < i -> continue
38     addi a1, a1, -4 # I-- for exit
39     # if start adress array's part equals last adress -> we cant take j+1 -> end of sorting
40     bne a1, a0, outter_loop
41
42 finish:
43     li a0, 10 # calling x10 = 10 for exit
44     ecall # exit
45
46 .rodata
47     length:
48         .word 10
49
50 .data
51 array:
52     .word 5, 1, 51, 42, 56, 0, 12, 23, 42, 4
53 |
```

Приложение 2

Main.s:

```
1 #main.s
2 .text
3
4 .global main
5
6 main:
7     la a0, array # array pointer
8     lw a1, array_length # length
9
10    addi sp, sp, -16
11    sw ra, 12(sp)
12
13    call sort
14
15    la a0, array_2
16    lw a1, array_length_2
17
18    call sort
19
20    lw ra, 12(sp)
21    addi sp, sp, 16
22
23    ret
24
25 .rodata
26 array_length:
27     .word 10
28 array_length_2:
29     .word 5
30
31 .data
32 array:
33     .word 12,32,42,24,5,52,42,2,6,2
34 array_2:
35     .word 23,42,4,2,0
36
```

Start.s:

```
1 #start.s
2 .text
3
4 start:
5 .global start
6
7     call main
8 finish:
9     li a0, 10
10    ecall
11
```

Sort.s:

```
1 #sort.s
2 .text
3
4 sort:
5 .globl sort
6 # a1 - length
7 # a0 - array pointer
8
9 mv a3, a1 # a3 - length
10 slli a3, a3, 2 # shift for multiply
11 add a3, a3, a0 # end of array+4
12 li a5, 1 # for exit
13 addi a3, a3, -4 # end of array
14 beq a3, a0, finish # if length = 1
15
16
17 outter_loop:
18     beqz a5, finish # if sorted
19     li a5, 0 # for exit
20     mv a4, a0 # restore pointer j = 0
21
22     inner_loop:
23         lw t2, 0(a4) # t2 = a[j]
24         lw t3, 4(a4) # t3 = a[j+1]
25
26         bltu t2, t3, skip_swap
27         li a5, 1 # for exit
28         sw t3, 0(a4) # swap
29         sw t2, 4(a4) # swap
30
31     skip_swap:
32         addi a4, a4, 4 # j++
33         bltu a4, a3, inner_loop # j < end
34
35         addi a3, a3, -4 # end--
36         bne a3, a0, outter_loop # end = start
37
38 finish:
39     ret
```