

Lab 3 汇编程序设计 report

PB21111681 朱炜荣

实验目的与内容

1. 了解并理解 RISC-V 常用的几个 32 位整数指令功能
2. 掌握 RISC-V 简单汇编程序设计，以及下载测试数据(COE 文件) 的生成方法
3. 熟悉 RISC-V 汇编程序仿真运行环境和调试基本方法

逻辑设计

1. 实验一的逻辑为：

利用寄存器数值中强制为 0 的 0 号寄存器，先测试 beq 指令，通过 beq 的正常跳转与否，来判断接下来的指令。如果 beq 正常跳转，则 t5 寄存器的值为 0，否则 t5 寄存器的值为 1（通过验证后续 addi 指令是否正确可以判断 beq 的跳转）。如果跳转成功，则可以利用 beq 来判断 jal 指令的正确性，即成功则继续顺序执行，失败则会利用 beq 的跳转至失败状态处。然后我们测试 lw 指令，利用 lw 读取数值并与 0 比较来判断 lw 是否成功。同样失败的话利用 jal 跳转。将 lw、jal、beq 指令验证完毕之后，其余的各种指令便通过设计容易判断是否正确。具体细节可以查看 test.asm 源代码。

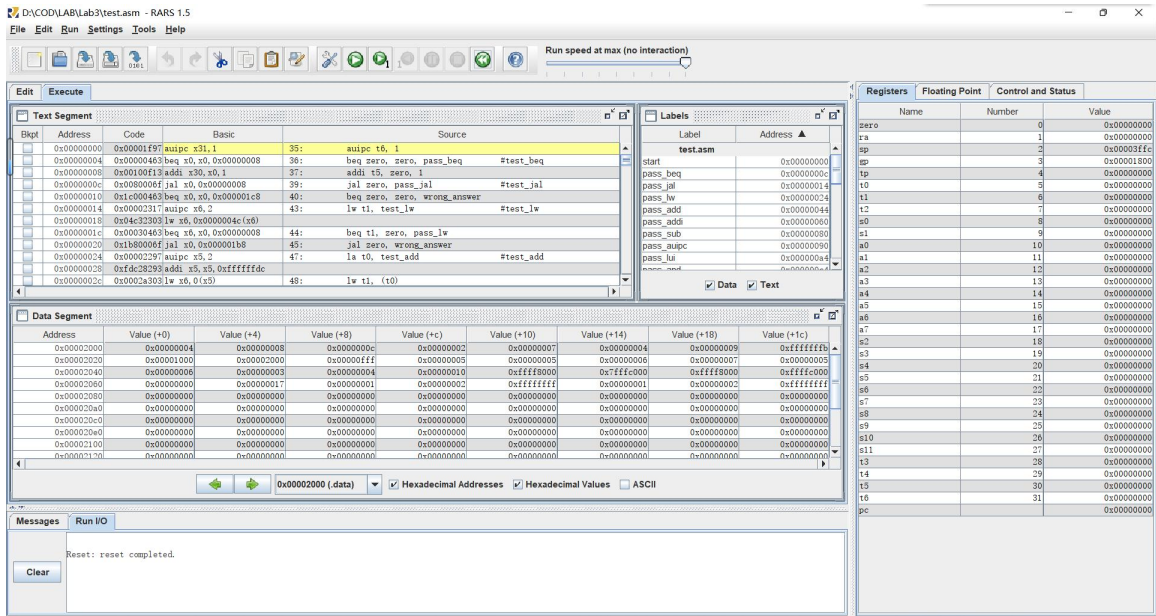
2. 实验二的逻辑为：

先读取要排序的数组长度。然后利用数组长度进行简单的冒泡排序，分别读取对应地址的两个数进行比较，如果更靠前的数更小，则将两个数交换位置，即分别将对方的数值输入到内存中。第一轮循环后将最大的数放到数组的第一位，然后移动“第一位”到第二位再进行循环，直到“第一位”到达最后一位。

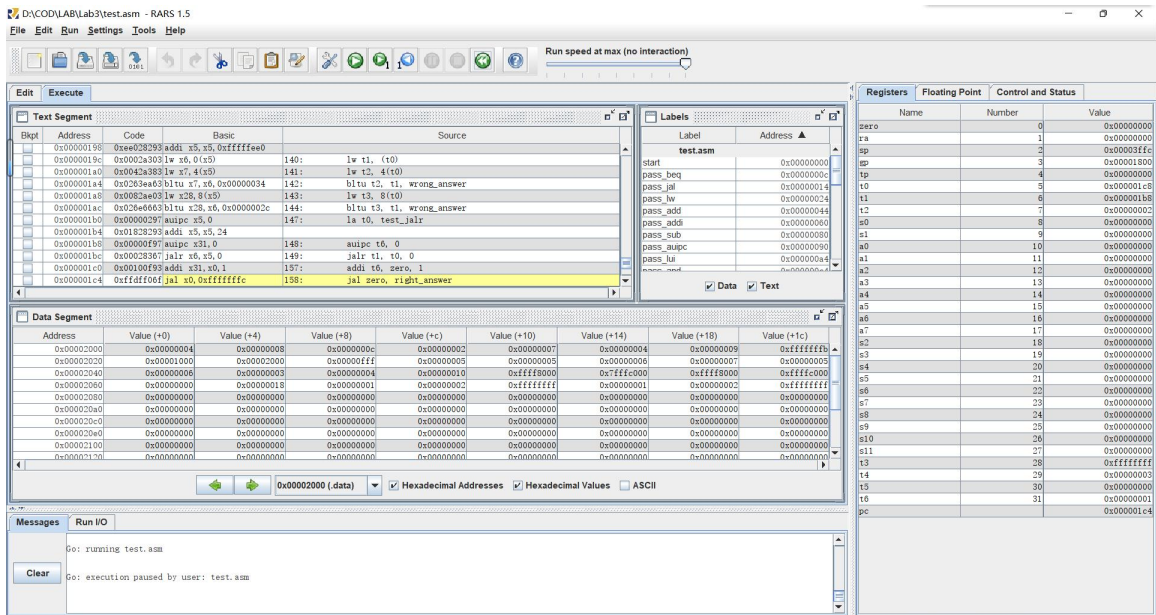
测试结果与分析

1. 实验一：

在运行之前整体的情况为：



在运行之后得到的结果为：



其中 t5 寄存器代表了 beq 指令是否正常，t5 为 0 则正常，t5 为 1 则异常。

t6 寄存器代表了当前状态是什么状态，t6 为 1 则为正常状态，t6 为 0 则为异常状态。

2. 实验二：

排序前的内存数值为：（其中数值的第一位为要排序的数组长度）

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00002000	0x00000005	0x00000003	0x00000004	0x0000000a	0x00000005	0x00000003	0x00000235	0x00000000
0x00002020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00002040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00002060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00002080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000020a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000020c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000020e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00002100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00002120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

进行排序之后得到的结果为：

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00002000	0x00000005	0x0000000a	0x00000005	0x00000004	0x00000003	0x00000003	0x00000235	0x00000000
0x00002020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00002040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00002060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00002080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000020a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000020c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000020e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00002100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00002120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

总结

本次实验总体而言难度较为正常，同时以实验督促学习，在实验中我进一步学习了有关 RISC-V 的有关语法和相关指令知识。同时借助 RARS 软件学会了如何将编写的汇编程序转变成机器码，也即生成 COE 文件的过程。同时学习如何在 RARS 中调试 RISC-V 汇编程序。本次实验的最大难度在于，如何设计合适的测试指令顺序使得，在后序测试需要的指令在该指令测试之前就要有所测试。最开始我想先测试 jal 指令，但在助教的提醒之下发现程序内存在逻辑错误，程序过程有点想自举，就不在此展示了。然后是写了多次的冒泡排序，我们先在 C 语言里学习、然后实现。再然后是 python、然后是 LC-3、接着是上次的 Verilog，这次是 RISC-v，不知道是不是冒泡排序在排序中的地位与 hello, world 在编程中的地位一样，写了很多次。不过好在都有惊无险地写出来了，也还算得上是一个基本的计科人吧。

附加代码

test.asm

```

1. .data
2. test_add:
3. .word 4, 8, 12

```

```
4.  test_addi:
5.    .word 2, 7
6.  test_sub:
7.    .word 4, 9, -5
8.  test_auipc:
9.    .word 4096
10. test_lui:
11.   .word 8192
12. test_and:
13.   .word 4095, 5
14. test_or:
15.   .word 5, 6, 7
16. test_xor:
17.   .word 5, 6, 3
18. test_slli:
19.   .word 4, 16
20. test_srli:
21.   .word -32768, 2147467264
22. test_srai:
23.   .word -32768, -16384
24. test_lw:
25.   .word 0
26. test_sw:
27.   .word 23
28. test_blt:
29.   .word 1, 2, -1
30. test_bltu:
31.   .word 1, 2, -1
32.
33. .text
34. start:
35.   auipc t6, 1
36.   beq zero, zero, pass_beq #test_beq
37.   addi t5, zero, 1
38. pass_beq:
39.   jal zero, pass_jal #test_jal
40.   beq zero, zero, wrong_answer
41. pass_jal:
42.
43.   lw t1, test_lw    #test_lw
44.   beq t1, zero, pass_lw
45.   jal zero, wrong_answer
46. pass_lw:
47.   la t0, test_add    #test_add
48.   lw t1, (t0)
49.   lw t2, 4(t0)
50.   lw t3, 8(t0)
51.   add t4, t1, t2
```

```

52.    beq t3, t4, pass_add
53.    jal zero, wrong_answer
54.    pass_add:
55.    la t0, test_addi    #test_addi
56.    lw t1, (t0)
57.    lw t2, 4(t0)
58.    addi t1, t1, 5
59.    beq t1, t2, pass_addi
60.    jal zero, wrong_answer
61.    pass_addi:
62.    la t0, test_sub     #test_sub
63.    lw t1, (t0)
64.    lw t2, 4(t0)
65.    lw t3, 8(t0)
66.    sub t4, t1, t2
67.    beq t3, t4, pass_sub
68.    jal zero, wrong_answer
69.    pass_sub:
70.    lw t1, test_auipc   #test_auipc
71.    beq t6, t1, pass_auipc
72.    jal zero, wrong_answer
73.    pass_auipc:
74.    lw t1, test_lui     #test_lui
75.    lui t2, 2
76.    beq t1, t2, pass_lui
77.    jal zero, wrong_answer
78.    pass_lui:
79.    la t0, test_and     #test_and
80.    lw t1, (t0)
81.    lw t2, 4(t0)
82.    and t3, t1, t2
83.    addi t4, zero, 5
84.    beq t3, t4, pass_and
85.    jal zero, wrong_answer
86.    pass_and:
87.    la t0, test_or      #test_or
88.    lw t1, (t0)
89.    lw t2, 4(t0)
90.    lw t3, 8(t0)
91.    or t4, t1, t2
92.    beq t3, t4, pass_or
93.    jal zero, wrong_answer
94.    pass_or:
95.    la t0, test_xor     #test_xor
96.    lw t1, (t0)
97.    lw t2, 4(t0)
98.    lw t3, 8(t0)
99.    xor t4, t1, t2

```

```

100. beq t3, t4, pass_xor
101. jal zero, wrong_answer
102. pass_xor:
103. la t0, test_slli #test_slli
104. lw t1, (t0)
105. lw t2, 4(t0)
106. slli t1, t1, 2
107. beq t1, t2, pass_slli
108. jal zero, wrong_answer
109. pass_slli:
110. la t0, test_srli #test_srli
111. lw t1, (t0)
112. lw t2, 4(t0)
113. srli t1, t1, 1
114. beq t1, t2, pass_srli
115. jal zero, wrong_answer
116. pass_srli:
117. la t0, test_srai #test_srai
118. lw t1, (t0)
119. lw t2, 4(t0)
120. srai t1, t1, 1
121. beq t1, t2, pass_srai
122. jal zero, wrong_answer
123. pass_srai:
124. la t0, test_sw #test_sw
125. lw t1, (t0)
126. addi t2, t1, 1
127. sw t2, (t0)
128. lw t3, (t0)
129. beq t1, t3, wrong_answer
130.
131. la t0, test_blt #test_blt
132. lw t1, (t0)
133. lw t2, 4(t0)
134. blt t2, t1, wrong_answer
135. lw t3, 8(t0)
136. blt t3, t1, pass_blt
137. jal zero, wrong_answer
138. pass_blt:
139. la t0, test_bltu #test_bltu
140. lw t1, (t0)
141. lw t2, 4(t0)
142. bltu t2, t1, wrong_answer
143. lw t3, 8(t0)
144. bltu t3, t1, wrong_answer
145.
146.
147. la t0, test_jalr

```

```

148. auipc t6, 0
149. jalr t1, t0, 0
150.
151. right_answer:
152. # la a0, str2
153. #      li a7, 4
154. #      ecall
155. # li a7, 10
156. #      ecall
157. addi t6, zero, 1
158.      jal zero, right_answer
159.
160.
161. test_jalr:
162. addi t1, t1, -8
163. beq t1, t6, pass_jalr
164. jal zero, wrong_answer
165. pass_jalr:
166. jal zero, right_answer
167.
168. wrong_answer:
169. # la a0, str1
170. #      li a7, 4
171. #      ecall
172. # li a7, 10
173. #      ecall
174.      and t6, t6, zero
175.      jal zero, wrong_answer
176.      beq zero, zero, wrong_answer
177.

```

sort.asm

```

1. .text
2. start:
3. la a2, array
4. la a3, n
5. lw t1, (a3)
6. addi t1, t1, -1 # t1 = n - 1
7. loop_i:
8. blt t1, zero, end # t1 >= 0?
9. and t2, t2, zero # t2 = 0
10. and t3, t3, zero # t3 = 0
11. loop_j:
12. beq t2, t1, exit_j # t2 == t1
13. slli t3, t1, 2 # t3 = t1 * 4

```

```
14.  add t3, a2, t3
15.  lw t5, (t3)
16.  slli t4, t2, 2  # t4 = t2 * 4
17.  add t4, a2, t4
18.  lw t6, (t4)
19.  blt t5, t6, smaller
20.  sw t6, (t3)
21.  sw t5, (t4)
22.  smaller:
23.  addi t2, t2, 1
24.  jal zero, loop_j
25.  exit_j:
26.  addi t1, t1, -1
27.  jal zero, loop_i
28.  end:
29.  jal zero, end
30.
31.
32.  .data
33.  n:
34.  .word 5
35.  array:
36.  .word 3,4,10,5,3,565
37.
```