Lab01 : RISC-V Programming

109550135 范恩宇

I. Bubble Sort:

1. There are 23 instructions in “main:”.

   In “printArray:” entered by line 8 , it requires 5 loops since there are 5 elements in given array , each with 10 instructions , so there are 10*5+5(remaining instructions)+6(part “Exit1:”)=61 instructions in this part .

   Because of having 5 elements , in “bubblesort:” entered by line 23 , each instruction is executed 5 times in “outloop:”, 10 times in “inloop:”, and 10 times in “swap:”, including the extra instructions before loop and those for leaving the loop , there are 6+6*5+9*10+18+2*10+1=165 instructions in this part .

   Finally , in “printSorted:” entered by line 36 , it requires 5 loops since there are 5 elements in given array , each with 10 instructions , so there are 10*5+13(remaining instructions)+9(part “Exit2:”)=72 instructions in this part . So there are 23+61+165+72 = 321 instructions in total (the 24 th instruction was calculated for 2 times in the picture).

```
.data
arr: .word 5,3,6,7,31
#arr: .word 5,3,6,7,31,23,43,12,45,1
str1: .string "Array： \n"
space: .string " "
str2: .string "Sorted： \n"
str3: .string "\n"

.text
main:
    la s0, arr                #1
    mv t3, s0                 #2

    la a0, str1               #3
    li a7, 4                  #4
    ecall                     #5

```

```
    addi s1, s1, 4            #6
    #addi s1, s1, 8
    addi t0, zero, -1         #7
    jal ra printArray         #8
    addi t0, zero, -1         #9  77
    jal ra, bubblesort        #10 78

    la a0, str3               #79 245
    li a7, 4                  #246
    ecall                     #247

    mv t0, zero               #248
    addi s1, s1, 1            #249
    mv s0, t3                 #250

    la a0, str2               #251
    li a7, 4                  #252
    ecall                     #253
    j printSorted             #254

```

```
bubblesort:
    addi sp, sp, -24          #80 255
    sw ra, 16(sp)             #81 256
    sw s1, 8(sp)              #82
    sw s0, 0(sp)              #83
    outloop:
        addi t0, t0, 1        #84 141 184 216 237
        mv t1, zero           #85 142 185 217 238
        sub t2, s1, t0        #86 143 186 218 239
        blt t0, s1, inloop    #87 144 187 219 240
        addi sp, sp, 24       #88 145 188 220 241
        jr ra                 #89 146 189 221 242
    inloop:
        mv s0, t3             #90 104 115 126 137 147 158 169 180 190 201 212 222 233 243
        bge t1, t2, outloop   #91 105 116 127 138 148 159 170 181 191 202 213 223 234 244
        slli t4, t1, 2        #92 106 117 128 139 149 160 171 182 193 203 214 224 235
        add s0, s0, t4        #93 107 118 129 140 150 161 172 183 193 204 215 225 236
        lw a2, 0(s0)          #94 108 119 130 151 162 173 194 205 226
        lw a3, 4(s0)          #95 109 120 131 152 163 174 195 206 227
        addi t1, t1, 1        #96 110 121 132 153 164 175 196 207 228
        bgt a2, a3, swap      #97 111 122 133 154 165 176 197 208 229
        j inloop              #98 112 123 134 155 166 177 198 209 230
swap:
    sw a2, 4(s0)              #99 113 124 135 156 167 178 199 210 231
    sw a3, 0(s0)              #100 114 125 136 157 168 179 200 211 232
    j inloop                  #101
```

```

printArray:
    bge t0, s1, Exit1         #11 23 35 47 59 71 102
    lw a0, 0(s0)              #12 24 36 48 60 72 103
    li a7, 1                  #13 25 37 49 61 73
    ecall                     #14 26 38 50 62
    la a0, space              #15 27 39 51 63
    li a7, 4                  #16 28 40 52 64
    ecall                     #17 29 41 53 65
    addi t0, t0, 1            #18 30 42 54 66
    addi s0, s0, 4            #19 31 43 55 67
    j printArray              #20 32 44 56 68

printSorted:
    bge t0, s1, Exit2         #21 33 45 57 69 257 269 281 293 305 317
    lw a0, 0(s0)              #22 34 46 58 70 258 270 282 294 306 318
    li a7, 1                  #259 271 283 295 307 318
    ecall                     #260 272 284 296 308
    la a0, space              #261 273 285 297 309
    li a7, 4                  #262 274 286 298 310
    ecall                     #263 275 287 299 311
    addi t0, t0, 1            #264 276 288 300 312
    addi s0, s0, 4            #265 277 289 301 313
    j printSorted             #266 278 290 302 314
```

```

Exit1:
    ret                       #74 267 279 291 303 315

Exit2:
    li a7, 10                 #75 268 280 292 304 316 320
    ecall                     #76 321
```

2. Since bubble sort is implemented by loop , not recursion , there is 0 variable pushed into stack .

II. GCD :

1. There are 9 instructions in “main:” .

After enterinh “ gcd ” at line 13 , it requires 12 (in “gcd:”)+6(in “cal ” , entered by line 30) for each recursion of calculating gcd , while there are 2 turns of recursion .

To stop the recursion and exit it , it needs 12+2 instructions . Finally , it requires 21 instructions to print the result , so there are

9+(12+6)*2+(12+2)+21=80 instructions in total .

```
.data
num1:  .word 4
num2:  .word 8
str1:  .string "GCD value of "
str2:  .string " and "
str3:  .string " is "

.text
main:
    lw s0, num1             #1
    lw s1, num2             #2

    jal ra, gcd             #3
    jal ra, printResult     #4   57

    # Exit program
    li a7, 10               #5   58
    ecall                   #6   59
```

```
gcd:
    addi sp,sp, -48         #7   25 43
    sw ra, 40(sp)           #8   26 44
    sw s2, 32(sp)           #9   27 45
    sw s1, 24(sp)           #10 28 46
    sw s0, 16(sp)           #11 29 47
    sw t1, 8(sp)            #12 30 48
    sw t0, 0(sp)            #13 31 49
    mv t1,s1                #14 32 50
    mv t0,s0                #15 33 51
    bnez t1,cal             #16 34 52

    addi sp,sp, 40          #17 35 53
    ret                     #18 36 54

cal:
    rem s2,s0,s1            #19 37 55
    mv s0,s1                #20 38 56
    mv s1,s2                #21 39
    j gcd                   #22 40
    lw ra, 40(sp)           #23 41
    lw s2, 32(sp)           #24 42
    lw s1, 24(sp)           #
    lw s0, 16(sp)           #
    lw t1, 8(sp)            #
    lw t0, 0(sp)            #
    addi sp, sp, 16         #
    ret                     #
```

```
printResult:
    mv t0, a0            #60
    mv t1, a1            #61
    la a0, str1          #62
    li a7, 4             #63
    ecall                #64
    lw a0, num1          #65
    li a7, 1             #66
    ecall                #67
    la a0, str2          #68
    li a7, 4             #69
    ecall                #70
    lw a0, num2          #71
    li a7, 1             #72
    ecall                #73
    la a0, str3          #74
    li a7, 4             #75
    ecall                #76
    mv a0, s0            #77
    li a7, 1             #78
    ecall                #79
    ret                  #80
```

2. There will be 3*6=18 variables pushed into the stack at the same time when the code is executed .

III. Fibonacci :

1. There are 10 instructions in "main:" .

   In part "fib:" , given n is 4 , this leads to the following results . First , instructions that judge if n <= 1 , push stacks and store return address are used for 4*2+1=9 times because we need fib(n-1) and fib(n-2) for each fib(n) . Second , instructions for calculation of fib(n-1) and fib(n-2) (both reload "fib:") , loading return address , and poping stacks are used for 4 times . Third , instructions for loading argument n , storing/loading fib(n-1) , and fib(n-1) + fib(n-2) are used for 8 times . Last but not least , if argument n <= 1 , it goes to "RT:" for 9 times in total .

There are 9*3+4*8+8*4+9=100 instructions in "fib:" part .

Finally , there are 25 instructions in part "printResult" , so there are 10+100+25=135 instructions in total .

```
.data
num: .word 4
str: .string "th number in the Fibonacci sequence is "

.text
main:
    lw a0, num              #1
    li s0, 1                #2
    jal ra, fib             #3

    mv a1, a0               #4 120
    lw a0, num              #5 121

    jal ra, printResult     #122
    li  a7, 10              #123
    ecall                   #124

```

```
fib:
    ble a0, s0 , RT        #6  14 22 30 42 61 80 88 100
    addi sp, sp, -24       #7  15 23 31 43 62 81 89 101
    sw ra, 16(sp)          #8  16 24 32 44 63 82 90 102
    sw a0, 8(sp)           #9  17 25 83
    addi a0, a0, -1        #10 18 26 84
    jal ra, fib            #11 19 27 85
    sw a0, 0(sp)           #12 20 28 36 55 74 86 94
    lw a0, 8(sp)           #13 21 29 37 56 75 87 95
    addi a0, a0, -2        #38 57 76 96
    jal ra, fib            #39 58 77 97
    lw t0, 0(sp)           #40 48 59 67 78 98 106 113
    add a0, a0, t0         #41 49 60 68 79 99 107 114
    lw ra, 16(sp)          #50 69 108 115
    addi sp, sp, 24        #51 70 109 116
    ret                    #52 71 110 117

RT:
    ret                    #33 45 53 64 72 91 103 111 118
```

```
printResult:
    mv t0, a0              #34 46 54 65 73 92 104 112 119 125
    mv t1, a1              #35 47 66 93 105 126
    li a7, 1               #127
    ecall                  #128

    mv a0, t0              #129
    la a0, str             #130
    li a7, 4               #131
    ecall                  #132

    mv a0, t1              #133
    li a7, 1               #134
    ecall                  #135
```

2. There will be 9+4+8=21 variables pushed into the stack at the same time when the code is executed .

IV. Experience :

Just like most of other classmates , it’ s my first time learning assembly codes , and it didn’ t go really well in the beginning . Comparing to other languages which I have learned , like C++ and Python , assembly code seems to run with a different logic . Although the original C code is simple , it really took me quite some time to figure out how the instructions switch between each part in assembly code . Not only understanding the structure , choosing between some certain commands is also not an easy work , even if I had already checked the command manual of RISC-V , for example “j” and “jal” . To solve this kind of problems , I tried those commands that I considered similar , and then checked how they run in the program . Through the above operations , assembly code is now more familiar to me , fortunately . For this lab , I think that “figuring out how to store the data” is truly an important task when generating assembly code . Once accomplishing this task and studying how the commands work more , this lab doesn’ t seem to be as difficult as the first time I encountered it .