**Lab01 : RISC-V Programming**

**109550135 范恩宇**

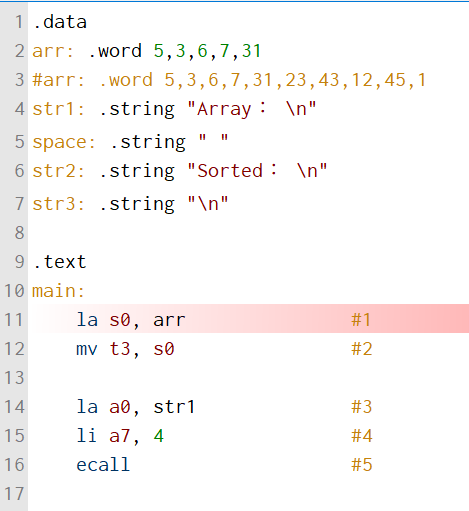
1. **Bubble Sort:**
   * 1. There are 23 instructions in “main:”.

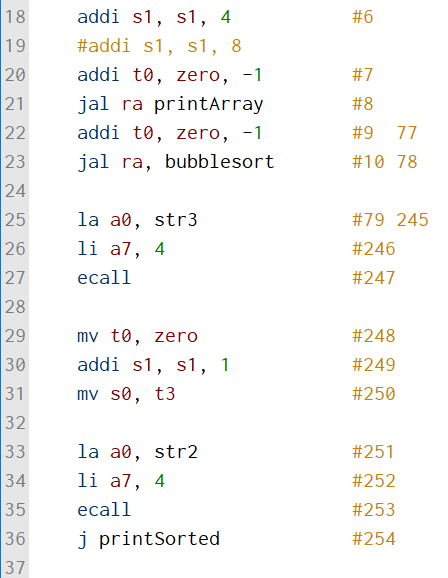
In “printArray:”entered by line 8 , it requires 5 loops since there are 5 elements in given array , each with 10 instructions , so there are 10\*5+5( remaining instructions )+6( part “Exit1:”)=61 instructions in this part .

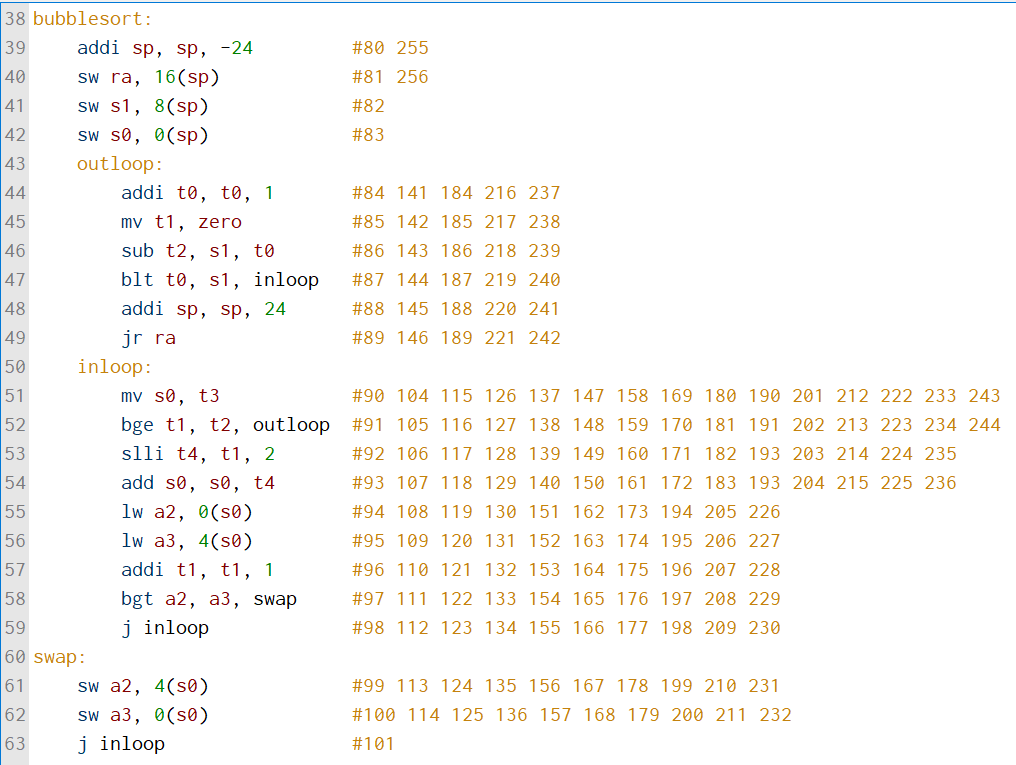
Because of having 5 elements , in “bubblesort:”

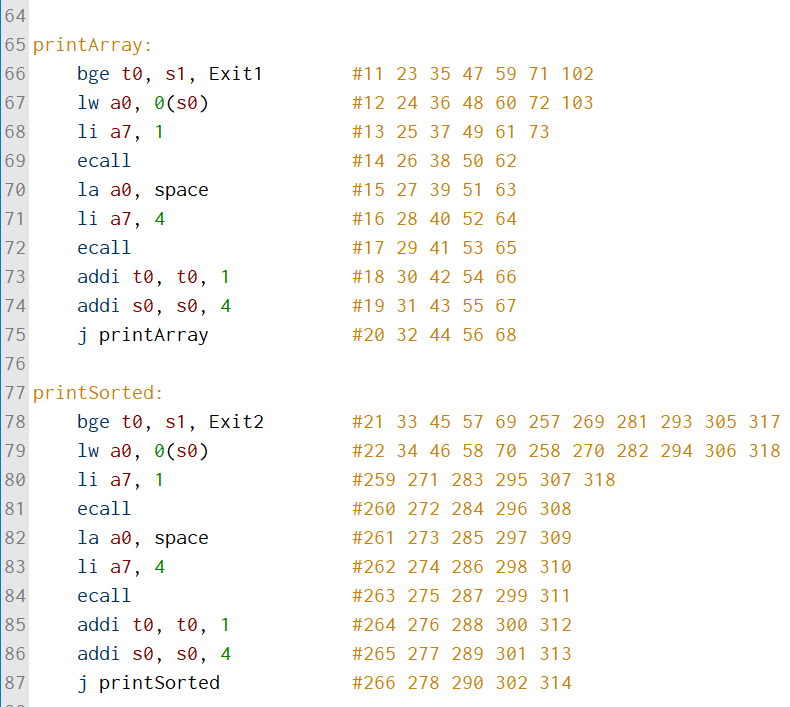
entered by line 23 , each instruction is executed 5 times in “outloop:”, 10 times in “inloop:”, and 10 times in “swap:”, including the extra instructions before loop and those for leaving the loop , there are 6+6\*5+9\*10+18+2\*10+1=165 instructions in this part .

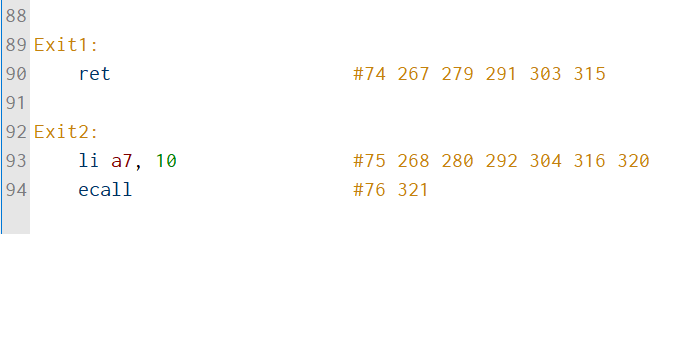
Finally , in “printSorted:”entered by line 36 , it requires 5 loops since there are 5 elements in given array , each with 10 instructions , so there are 10\*5+13( remaining instructions )+9( part“Exit2:”)=72 instructions in this part . So there are 23+61+165+72 = 321 instructions in total ( the 24 th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









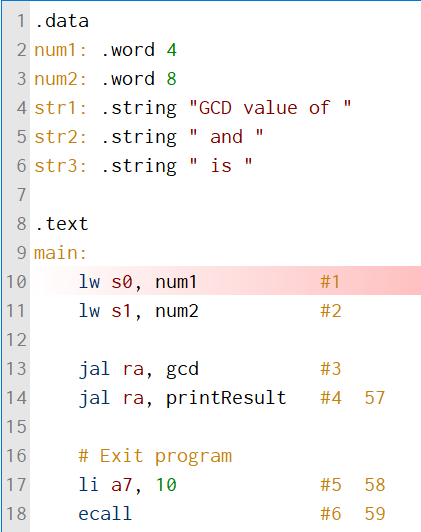


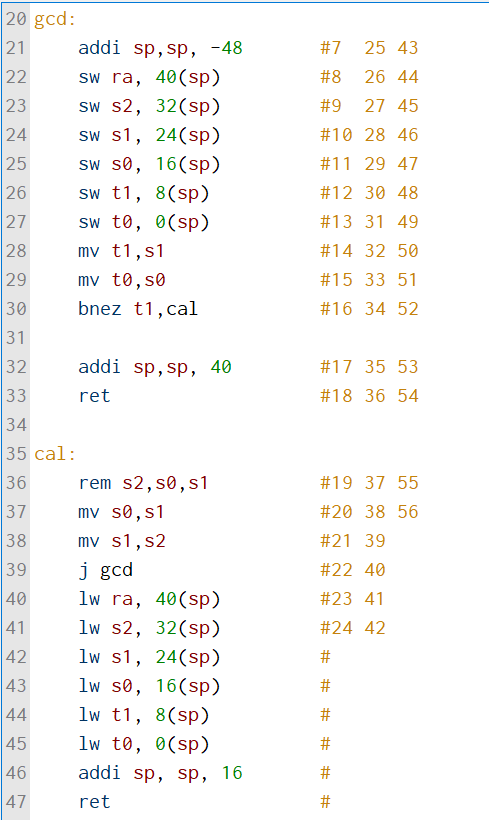
* + 1. Since bubble sort is implemented by loop , not recursion , there is 0 variable pushed into stack .

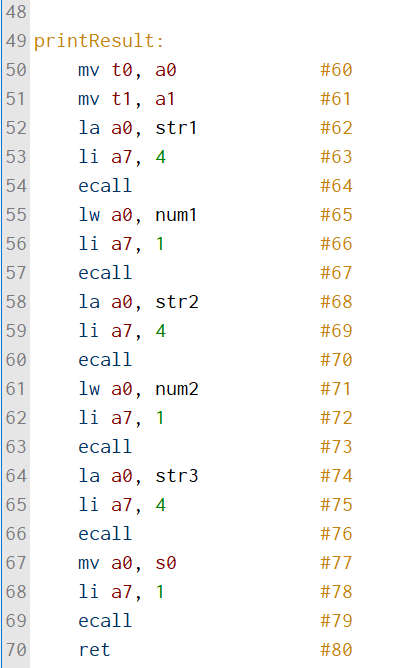
1. **GCD :**
   * 1. There are 9 instructions in “main:”.

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 .





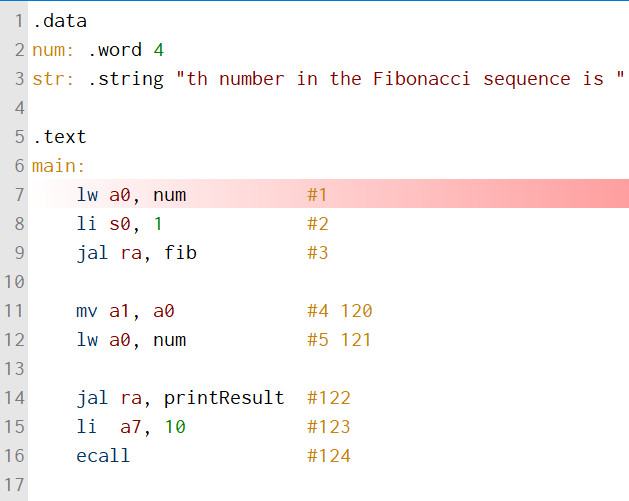


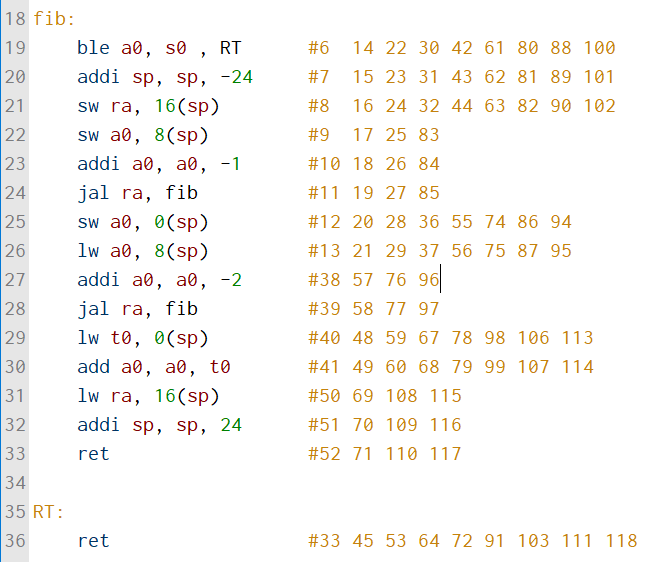
* + 1. There will be 3\*6=18 variables pushed into the stack at the same time when the code is executed .

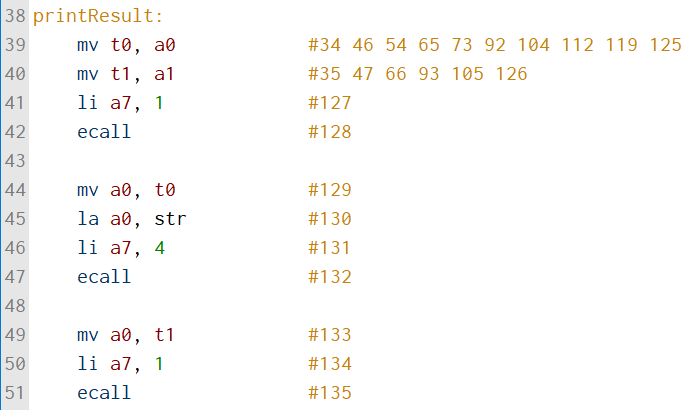
1. **Fibonacci :**
   * 1. There are 10 instructions in “main:”.

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 .







* + 1. There will be 9+4+8=21 variables pushed into the stack at the same time when the code is executed .

1. **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 .