第四次理论作业

一、冒泡排序

首先写出C代码如下：

```
int A[10] = { ... };

printf("Please input 10 numbers:\n");

for (int i = 0; i <= 9; i++) scanf("%d", &A[i]);

for (int i = 0; i <= 8; i++) {
    for (int j = 0; j <= 8 - i; j++) {
        if (A[j] > A[j + 1]) {
            int temp = A[j];
            A[j] = A[j + 1];
            A[j + 1] = temp;
        }
    }
}

printf("Sorted 10 numbers is:\n");

for (int i = 0; i <= 9; i++) printf("%d", A[i]);
```

翻译如下：

```
.data
space: .asciiz " "
welcome: .asciiz "Please input 10 numbers:\n"
sorted: .asciiz "Sorted 10 numbers is:\n"

.text
.globl main

main:
    li $t0, 0
    li $t1, 10
    li $t2, 9
    li $t3, 0
    li $t4, 0

    li $v0, 4
    la $a0, welcome
    syscall

read_loop:
    beq $t0, $t1, sort
    la $a0, A
    mul $a1, $t0, 4
    add $a0, $a0, $a1
    li $v0, 5
    syscall
    sw $v0, 0($a0)
    addi $t0, $t0, 1
    j read_loop

sort:
    li $t0, 0
outer_loop:
    li $t3, 0
inner_loop:
    la $a0, A
    mul $t5, $t3, 4
    add $a0, $a0, $t5
    addi $a1, $a0, 4

    lw $t5, 0($a0)
    lw $t6, 0($a1)

    ble $t5, $t6, no_swap

    sw $t6, 0($a0)
    sw $t5, 0($a1)

no_swap:
    addi $t3, $t3, 1
```

```
    bne $t3, $t2, inner_loop

    addi $t0, $t0, 1
    bne $t0, $t1, outer_loop

    li $v0, 4
    la $a0, sorted
    syscall

    li $t0, 0
print_loop:
    beq $t0, $t1, done
    la $a0, A
    mul $a1, $t0, 4
    add $a0, $a0, $a1
    lw $a0, 0($a0)
    li $v0, 1
    syscall
    li $v0, 4
    la $a0, space
    syscall
    addi $t0, $t0, 1
    j print_loop

done:
    li $v0, 10
    syscall
```

代码截图如下：

```
1  .data
2  A:       .space 40
3  space: .asciiz " "
4  welcome: .asciiz "Please input 10 numbers:\n"
5  sorted: .asciiz "Sorted 10 numbers is:\n"
6
7  .text
8  .globl main
```

在这里， `.data` 段中为 `A` 预留了 `4*10=40` 字节的空间。并且设置了 `space` 以输出空格； `welcome` 以输出输入提示语； `sorted` 以输出排序完成提示语。

```
10  main:
11      li $t0, 0
12      li $t1, 10
13      li $t2, 9
14      li $t3, 0
15      li $t4, 0
16
17      li $v0, 4
18      la $a0, welcome
19      syscall
```

进入 main 后，我规定用 $t0 寄存外层变量 i ， $t3 寄存内层变量 j 。同时， $t1 寄存外层变量上限，即 10 ； $t2 寄存内层变量上限，即 9 。还有 $t4 用于寄存暂时变量 temp 。

之后调用字符串输出，输出输入提示语 welcome 。

```
21  read_loop:
22      beq $t0, $t1, sort
23      la $a0, A
24      mul $a1, $t0, 4
25      add $a0, $a0, $a1
26      li $v0, 5
27      syscall
28      sw $v0, 0($a0)
29      addi $t0, $t0, 1
30      j read_loop
```

之后进入读取十个数的循环，命名为 read_loop 。当 $t0 从 0 自增到 10 的时候(自增了 11 次，但剩余内容只执行了 10 次)即可跳到 sort 循环段。

首先加载 A 的首地址到 $a0 。之后通过此时的 i 值(在 $t0 中)计算出偏移量，即乘 4 (或左移两位)，再将 $a0 的值加上偏移量得到输入数的存放地址。读入数后存储，并使 i 自增 1 。

```
32  sort:
33      li $t0, 0
34  outer_loop:
35      li $t3, 0
36  inner_loop:
37      la $a0, A
38      mul $t5, $t3, 4
39      add $a0, $a0, $t5
40      addi $a1, $a0, 4
41
42      lw $t5, 0($a0)
43      lw $t6, 0($a1)
44
45      ble $t5, $t6, no_swap
46
47      sw $t6, 0($a0)
48      sw $t5, 0($a1)
```

开始排序。首先计算出相应相邻位置的地址，然后比较大小。之后判断是否跳转。如果符合条件，则进入下一条语句以交换。否则跳入 no_swap 分支以规避值交换。

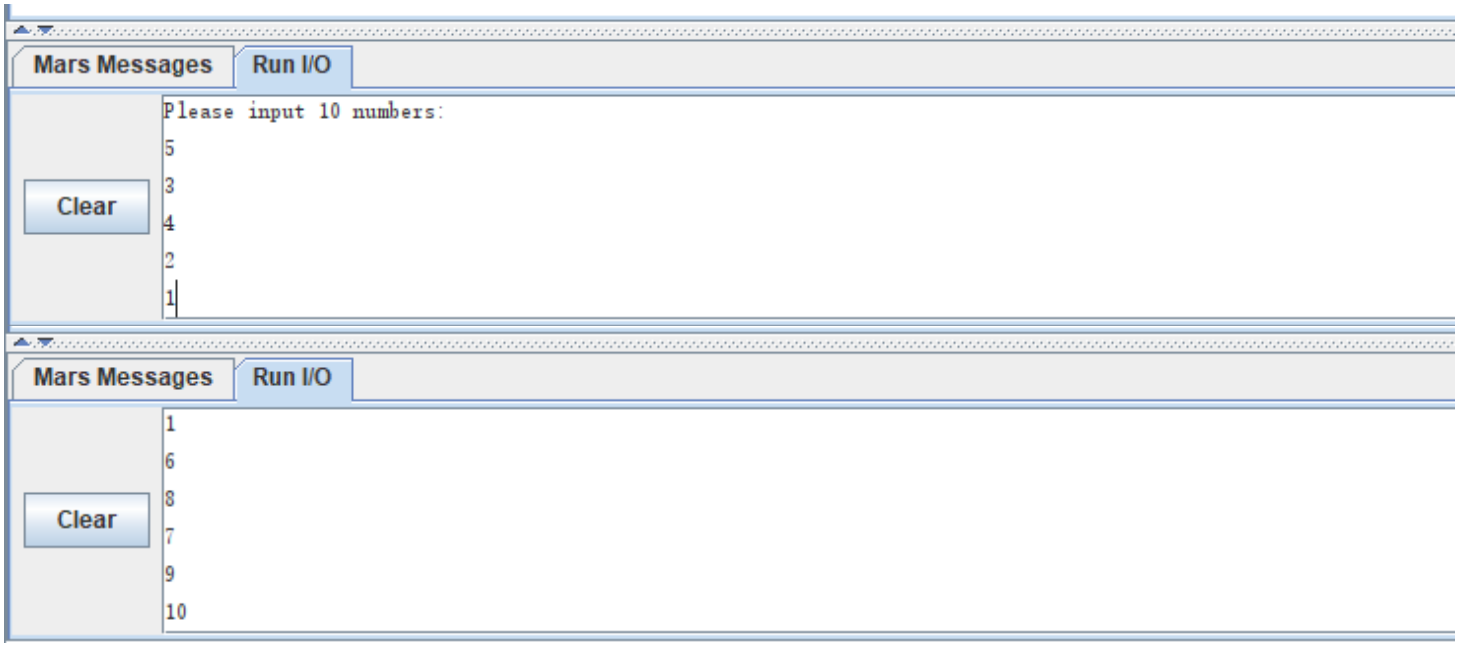
```
53  no_swap:
54      addi $t3, $t3, 1
55      bne $t3, $t2, inner_loop
56
57      addi $t0, $t0, 1
58      bne $t0, $t1, outer_loop
59
60      li $v0, 4
61      la $a0, sorted
62      syscall
```

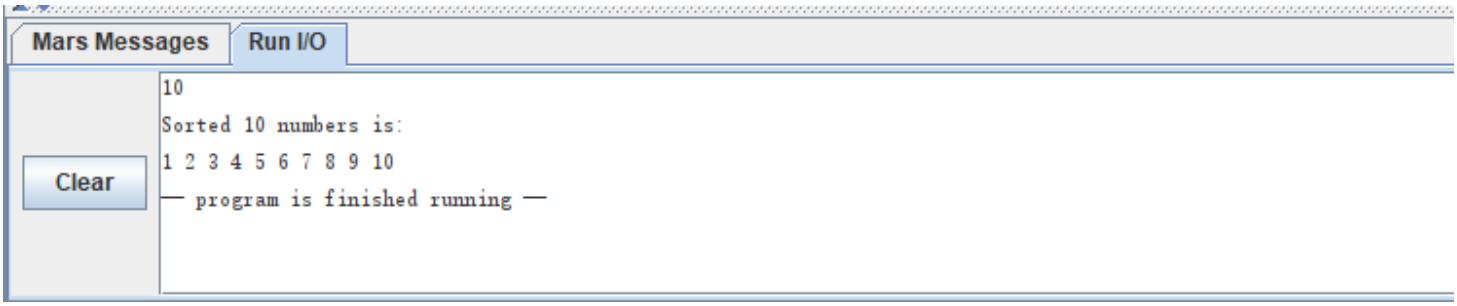
如果不符合条件，就无需交换数值，直接跳到此段 no_swap 。但 no_swap 无论是否符合条件都会生效。这一段实现 i ， j 的自增，并判断应该跳到外循环还是内循环，还是已经排序完成到输出阶段。

```
64      li $t0, 0
65  print_loop:
66      beq $t0, $t1, done
67      la $a0, A
68      mul $a1, $t0, 4
69      add $a0, $a0, $a1
70      lw $a0, 0($a0)
71      li $v0, 1
72      syscall
73      li $v0, 4
74      la $a0, space
75      syscall
76      addi $t0, $t0, 1
77      j print_loop
78
79  done:
80      li $v0, 10
81      syscall
```

最后是输出循环。输出十次之后即可停止，跳转到 done 标签，返回到操作系统。

运行结果如下：





在编写的过程中，我感受到汇编的思维过程与高级语言稍有差别。例如，从编写的便捷性来说，为了减少分支跳转，考虑到重复的过程，在汇编中事实上是判断为假才跳转。

二、阶乘

首先写出C代码如下：

```c
int main()
{
    int start = 0, result = 0;
    scanf("%d", &start);
    result = Fac(start);
    printf("%d", result);
}

int Fac(int n)
{
    if (n >= 2) return (n * Fac(n - 1));
    if (n == 1) return 1;
}
```

翻译如下：

```
.data
Input: .asciiz "Input a number: "
Result: .asciiz "Result: "

.text
main:
    li $v0, 4
    la $a0, Input
    syscall

    li $v0, 5
    syscall

    move $t0, $v0
    move $t1, $t0
    li $a1, 1

Fac:
    addi $sp, $sp, -8
    sw $ra, 0($sp)
    sw $t1, 4($sp)
    beq $t1, 1, re0
    addi $t1, $t1, -1
    jal Fac

    lw $a0, 4($sp)
    mult $a1, $a0
    mflo $a1
    beq $t1, $t0, done

re0:
    addi $t1, $t1, 1
    lw $ra, 0($sp)
    addi $sp, $sp, 8
    jr $ra

done:
    la $a0, Result
    li $v0, 4
    syscall

    move $a0, $a1
    li $v0, 1
    syscall

    li $v0, 10
    syscall
```

代码截图如下：

```
1  .data
2  Input: .asciiz "Input a number: "
3  Result: .asciiz "Result: "
```

提示语存储在 data 段。

```
5  .text
6  main:
7          li $v0, 4
8          la $a0, Input
9          syscall
10
11         li $v0, 5
12         syscall
13
14         move $t0, $v0
15         move $t1, $t0
16         li $a1, 1
```

进入 main 函数之后，主要任务有三个：

其一，提示输入一个数；

其二，从键盘读取一个数；

其三，设置初始值。

在这里，规定 $t0 做存储 const 型数作用，而 $t1 在程序生命周期内做变量直到递减为 1 ， $a1 做中间/最后结果存储。

```
18  Fac:
19          addi $sp, $sp, -8
20          sw $ra, 0($sp)
21          sw $t1, 4($sp)
22          beq $t1, 1, re0
23          addi $t1, $t1, -1
24          jal Fac
25
26          lw $a0, 4($sp)
27          mult $a1, $a0
28          mflo $a1
29          beq $t1, $t0, done
```

进入 `Fac` 段。首先压栈存值。直到 `$t1` 减为 `1` 跳到 `re0` 段，之后再一层层出栈(但这是 `re0` 的任务)。每次出栈将暂时结果乘以当前出栈的数值，最后判断 `$t0==$t1` 得到最终结果,跳到 `done` 执行最后指令段。

```
31  re0:
32          addi $t1, $t1, 1
33          lw $ra, 0($sp)
34          addi $sp, $sp, 8
35          jr $ra
```

执行出栈。

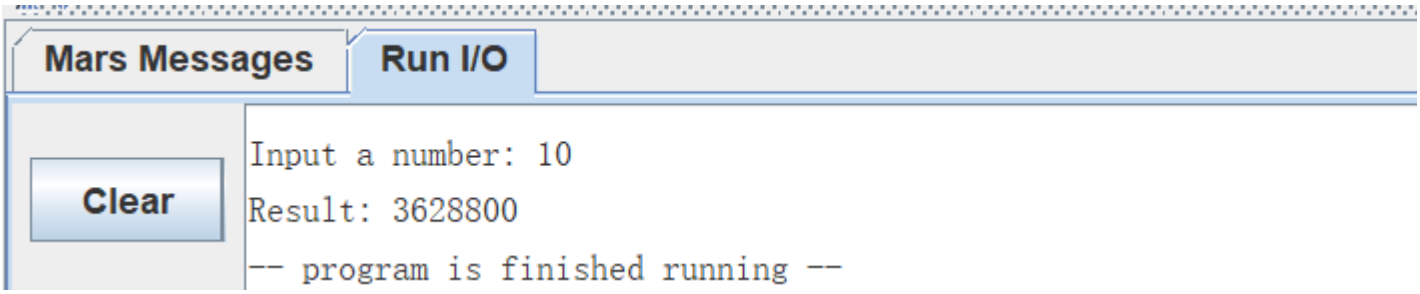
```
done:
        la $a0, Result
        li $v0, 4
        syscall

        move $a0, $a1
        li $v0, 1
        syscall

        li $v0, 10
        syscall
```

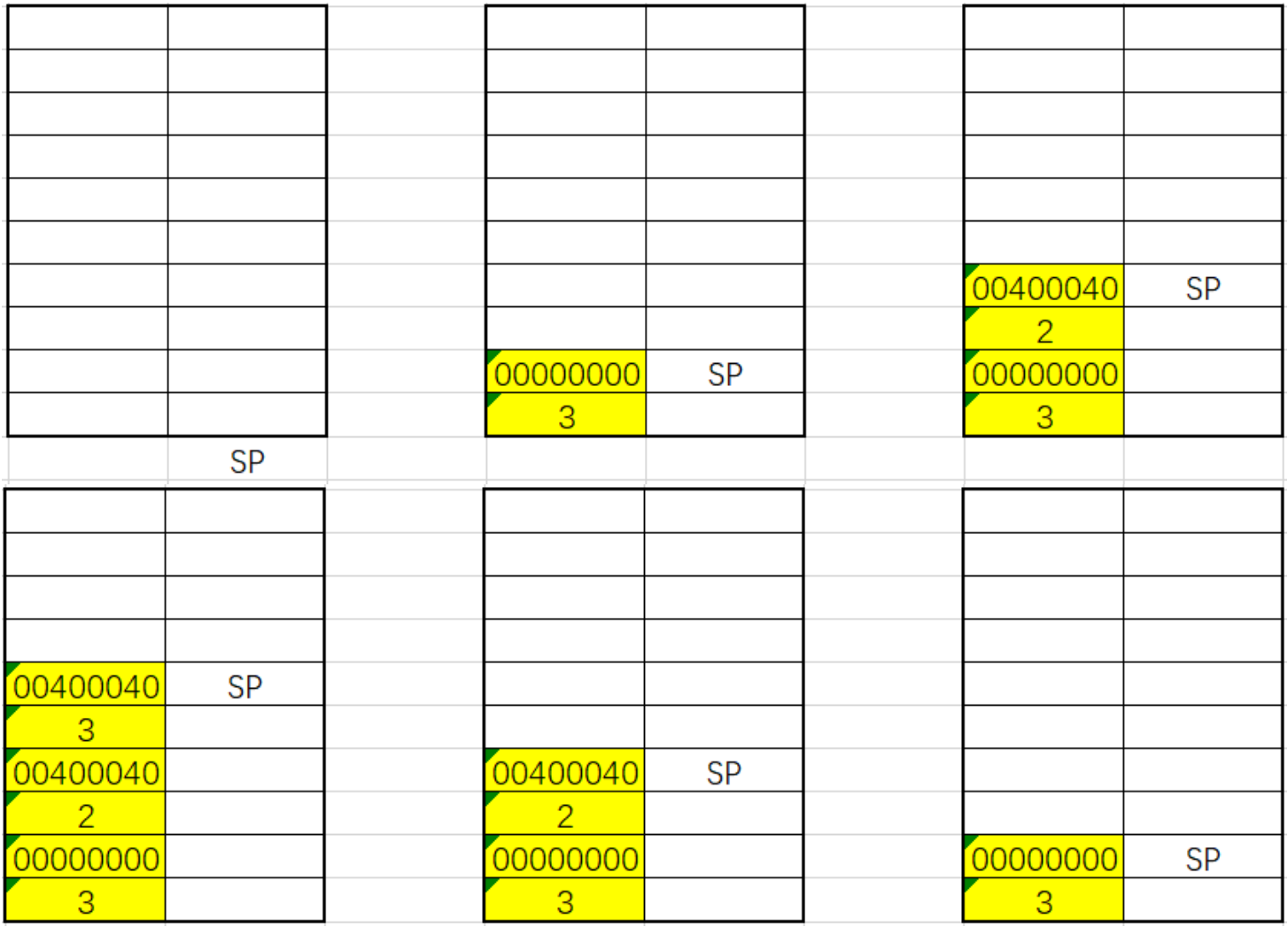
输出 `Result` 字符串，输出最终结果，返回操作系统。

程序运行截图如下：



可以感受到，汇编的递归函数对思维有较高要求。高级语言隐藏了压栈出栈的过程，使得程序员能把更多心思花在数据本身上而不是程序如何运行上。

简便起见，假设递归深度为 `2` ，即要计算的是 `3!` 。堆栈变化简要图：



首先连入三次栈，每次都保存当时 `$ra` ， `$t1` 。但由于 `$a1` 初始已经设置为 `1` ， 故第一次入栈的 `1` 的值可以不出栈，即只需出栈两次。因此只需依次计算：

`$a1*2` , `($a1*2)*3`

即可。

在这里，直到 `$t1` 变为 `1` ， 每次遇到 `jal Fac` 都将跳转到 `Fac` 标签，只压栈。

`$t1` 变为 `1` 之后，跳入 `re0` 段，开始出栈。每次遇到 `jr $ra` 都将回到 `jal Fac` 的下一条语句 `lw $a0, 4($sp)` ， 执行计算与取值。

直到 `$t1` 恢复为 `3` ， 跳转到 `done` 执行程序最后阶段。