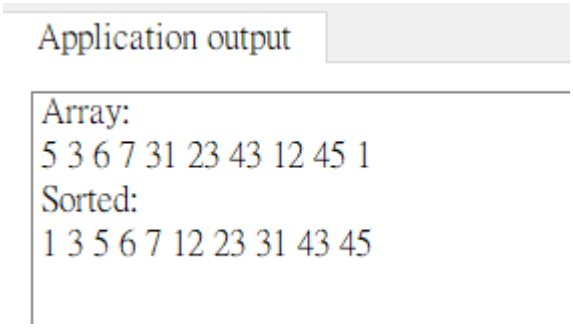
Computer Organization, Spring 2020

Lab 1: RISC-V Programming

學號: 0846013
姓名: 葉致廷

1. Bubble sort

Output of bubble_sort.s:



Explanation:

```
.data
```
data section
```
argument: .word 10
```
a static variable argument, which is a word and stores 10
```
newline: .string "\n"
```
a static variable newline, which is a string and change new line
```
space: .string " "
```
a static variable newline, which is a string and stores a space
```
str1: .string "Array: "
```
a static variable st1, which is a string and stores “Array: “
```
str2: .string "Sorted: "
```
a static variable st1, which is a string and stores “Sorted: “
```
data: .word   5 3 6 7 31 23 43 12 45 1
```
a static array with a length of 10, which stores value 5,3,6,7,31,23,43,12,45,1
```
.text
```
text section
```
main:
```
main procedure
```
la   a3, data
```
load the base address of data array into register a3
```
lw   a2, argument
```
load word, argument, into register a2
```
 jal  ra, printResult1
```
jump and link to printResult1 to print out “Array: “

```
jal  ra, printArray
```
jump and link to printArray procedure to print out the unsorted array
```
jal  ra, printResult2
```
jump and link to printResult2 to print out “Sorted: “
```
jal  ra, printArray
```
jump and link to printArray procedure to print out the sorted array
```
li        a0, 10
ecall
```
Exit program

```
bubblesort:
```
bubble sort procedure
```
addi  sp,sp, -40
```
Everytime bubblesort procedure is called, create a stack frame of 5 items
```
sw ra, 32(sp)
```
store ra into stack
```
 sw s6, 24(sp)
```
store s6 into stack
```
sw s5, 16(sp)
```
store s5 into stack
```
sw s4, 8(sp)
```
store s4 into stack
```
sw s3, 0(sp)
```
store s3 into stack
```
 mv s5, a3
```
copy base address into s5
```
 mv s6, a2
```
copy argument into s6
```
# Outer loop
li s3, 0
```
load immediate 0 into s3, which also means initialize i to 0
```
for1ist: bge s3, a2, exit1
```
go to exit1 if i >= n
```
 # Inner loop
addi s4,  s3, -1
```
Otherwise, start of the inner for loop, j = i - 1
```
for2tst:    blt s4, zero, exit2
```
if j < 0 go to exit2
```
slli t0, s4, 2
```
Otherwise,reg t0 = j*4
```
add t0, s5, t0
```
reg t0 = v(base address) + j*4
```
lw t1, 0(t0)
```

load v[j]: reg x6 = v[j]

```
lw t2, 4(t0)
```

load v[j+1]: reg x7 = v[j+1]

```
ble t1, t2, exit2
```

go to exit2 if t1 <= t2

```
# Pass parameters and call
mv a0, s5
```

move v(base address) to a0

```
 mv a1, s4
```

move j(index) to a0

```
 jal ra, swap
```

call swap procedure

```
addi s4, s4, -1
```

j--

```
j for2tst
```

go to the next innner loop iteration

```
exit2:
```

exit2 block(After the inner loop is over)

```
addi s3, s3, 1
```

i++

```
j for1ist
```

jump to the next iteration of the outer loop

```
exit1:
```

exit1 block(After the outer loop is over)

```
lw s3, 0(sp)
```

Restore s3 from stack

```
lw s4, 8(sp)
```

Restore s4 from stack

```
 lw s5, 16(sp)
```

Restore s5 from stack

```
lw s6, 24(sp)
```

Restore s6 from stack

```
lw ra, 32(sp)
```

Restore return address from stack

```
addi sp, sp, 40
```

pop 5 items from stack when the current procedure is over

```
jalr     x0, x1, 0
```

Return to calling routine

```
swap:
```

swap procedure

```
slli    t1, a1, 2
```

reg t1 = j*4

```
add     t1, a0, t1
```
reg t1 = v + (j*4)
```
lw      t0, 0(t1)
```
load the value of v[j] into t0, reg t0 = v[j]
```
lw      t2, 4(t1)
```
load the value of v[j+1] into t2, reg t2 = v[j+1]
```
sw      t2, 0(t1)
```
save the value in v[j] into t2, v[k] = reg t2
```
sw      t0, 4(t1)
```
save the value in v[j+1] into t2, v[j+1] = reg t0
```
jalr    x0, x1, 0
```
Return to calling routine

```
la       a1, str1
```
load the address of str1 in data section into a1
```
li       a0, 4
ecall
```
Prints the null-terminated string located at address in `a1`
```
la       a1, newline
```
load the address of newline in data section into a1
```
li       a0, 4
ecall
```
Prints the null-terminated string located at address in `a1`
```
jalr    x0, x1, 0
```
Return to the caller procedure
```
la       a1, str2
```
load the address of str2 in data section into a1
```
li       a0, 4
ecall
```
Prints the null-terminated string located at address in `a1`
```
la       a1, newline
```
load the address of newline in data section into a1
```
li       a0, 4
ecall
```
Prints the null-terminated string located at address in `a1`
```
jalr    x0, x1, 0
```
Return to the caller procedure

```
la       a1, newline
```
load the address of newline in data section into a1
```
li       a0, 4
ecall
```

Prints the null-terminated string located at address in `a1`

```
jalr    x0, x1, 0
```

Return to the caller procedure

```
li      t0, 0
```

load immediate 0 into t0

i = 0

```
loop:
```

the loop block

```
slli    t1, t0, 2
```

t1 = i*4

```
add     t2, a3, t1
```

t2 = base address of array data + i*4

```
lw      t3, 0(t2)
```

load the value at t2 in data section into t3

```
mv      a1, t3
```

move the value in t3 to a1

```
li      a0, 1
ecall
```

Prints the value located in `a1` as a signed integer

```
la      a1,  space
```

load the address of space in data section into a1

```
li      a0,     4
ecall
```

Prints the null-terminated string located at address in `a1`

```
addi    t0, t0, 1
```

increment t0

i++

```
blt     t0, a2, loop
```

check stopping condition

if t0 is lesser than a2(argument) then go to the next iteration

If not · do the rest

```
la       a1, newline
```

load the address of newline in data section into a1

```
li       a0, 4
ecall
```

Prints the null-terminated string located at address in `a1`

```
jalr    x0, x1, 0
```

Return to the caller procedure

- data section:

```
1   .data
2   argument: .word 10
3   newline: .string "\n"
4   space: .string " "
5   str1: .string "Array: "
6   str2: .string "Sorted: "
7   data: .word   5 3 6 7 31 23 43 12 45 1
```

這邊是記憶體中的static data，值得解釋的是我特別把換行跟空格當作字串儲存以便之後輸出可以符合格式。另外第7行code是integer array, data 的表示。

- text section:

包含了以下的block，以下一一對重要的部分解釋

```
 9   .text
10 > main:        ···
26 > bubblesort: ···
38 > for1ist:    bge x19, a2, exit1  # go to exit1 if i >= n···
43 > for2tst:    blt x20, zero, exit2 # if x20 < zero exit2···
55 > exit2:      addi x19, x19, 1     # i++···
58 > exit1:      lw x19, 0(sp)        # Restore x19 from stack···
66 > swap: ···
76 > printResult1: # print str1···
87 > printResult2: # print str2          ···
99 > printResult3: # printf("\n") ···
105 > printArray: ···
108 > loop:    ···
```

main:

```
 9   .text
10   main:
11               la   a3, data
12               lw   a2, argument
13               # print str1
14               jal  ra, printResult1
15               # print array
16               jal  ra, printArray
17               # bubblesort
18               jal  ra, bubblesort
19               # print str2
20               jal  ra, printResult2
21               # print array
22               jal  ra, printArray
23               # Exit program
24               li       a0, 10
25               ecall
```

register a3 儲存了 data array 的 base address 所以這邊用的operation是la (load address)
register a2 則儲存了 argument，利用的是load word operation
接下來，第14~22行 是利用jal 去其他的function 再利用當前jal 記下的ra 跳回

最後則是結束main function 的 environment calls 的表達方式，而 a0 = 10 是exit 的意思。

bubblesort:

```
27   bubblesort:
28               addi  sp,sp, -40
29               sw ra, 32(sp)
30               sw s6, 24(sp) # n
31               sw s5, 16(sp) # v
32               sw s4, 8(sp)  # j
33               sw s3, 0(sp)  # i
34               # Procedure body
35               mv s5, a3    # copy base address into x21
36               mv s6, a2    # copy argument into x22
```

這裡算是整個bubble sort的前置作業，先把stack frame開出來存入五個參數的初始值，雖然這裡bubble sort只有一層也不會與swap function用到的register相互汙染，以結果來說只有ra是真的需要存的，但我想把會用register存起來是好習慣，所以還是照著課本那般的做了。其中，reg s5 暫存了base address，reg s6 暫存了 argument，接著就要進入雙層的 for loop了。

首先進入Outer for loop
進入前先初始化s3， li s3, 0 是 load immediate 0 進s3，也就是說s3被初始為0了。

```
38               # Outer loop
39               li s3, 0     # i = 0, initial condition
```

for1st:

```
41   for1ist:    bge s3, a2, exit1  # go to exit1 if i >= n
```

for loop 有三個條件 initial condition 前面s3已初始化為0，stopping condition 就是第41行的意思，如果s3 be greater than or equal to a2 (argument) 的話就跳離最外層迴圈，去exit1，而在每次iteration之中跟結束做了哪些事呢? 我們繼續往第二層for loop走。

接著準備進入 Inner for loop， 進入前先把 initial conditon設好，j = i -1，s4 = s3 + (-1)

```
42               # Inner loop
43               addi s4,  s3, -1  # j = i-1, initial condition
```

for2tst:

```
44   for2tst:    blt s4, zero, exit2 # if x20 < zero exit2
45               slli t0, s4, 2      # reg x5 = j*4
46               add t0, s5, t0      # reg x5 = v + (j*4)
47               lw t1, 0(t0)         # load v[j]: reg x6 = v[j]
48               lw t2, 4(t0)         # load v[j+1]: reg x7 = v[j+1]
49               ble t1, t2, exit2    # go to exit2 if x6 <= x7
50               # Pass parameters and call
51               mv a0, s5           # first parameter is v
52               mv a1, s4           # second parameter is j
53               jal ra, swap         # call swap
54               addi s4, s4, -1    # j--
55               j for2tst            # go to the next iteration
```

stopping condition為兩個，一是 s4 be lesser than zero，當條件成立，去exit2，結束Inner loop。二是當data[j] < data[j+1] 時，這就是第45~49行在做的事，第44行 將s4*4 暫存進t0中，再將 base address + s4*4 的結果暫存進t0中，其代表的是v[j] 在 memory中的位置。

lw t1, 0(t0) 將 v[j] 的content， load 進reg t1 中。lw t2, 4(t0) 則將 v[j+1] 的content， load 進reg t2 中，然後第49行便是在比較二者的值，如果v[j] 比較小，則去exit2。
而在每次的iteration中(不符合stopping condition)，則將s5的值傳入a0，s4的值傳入a1，當作swap的function argument，然後jal 去swap。當swap結束跳回bubble sort中 則將s4減一，存回s4，進入下一輪iteration中。

```
57    exit2:      addi s3, s3, 1     # i++
58                j for1ist            # jump to the next iteration
```

而在每次Inner loop terminate 後把s3減一，存回s3，並進入Outer loop 下一輪的iteration中。

```
60    exit1:      lw s3, 0(sp)         # Restore x19 from stack
61                lw s4, 8(sp)
62                lw s5, 16(sp)
63                lw s6, 24(sp)
64                lw ra, 32(sp)
65                addi sp, sp, 40
66                jalr     x0, x1, 0   # Return to calling routine
```

當Outer for loop terminates 跳到exit1來 要做的事是 restore所有在stack中的值讓這些register回到初始狀態，並pop掉這些item，釋放stack中的空間。並return 到bubble sort的caller也就是main procedure中。

緊接著，來介紹swap procedure

```
68    swap:
69                slli    t1, a1, 2    # reg x6 = k*4
70                add     t1, a0, t1   # reg x6 = v + (k*4)
71                lw      t0, 0(t1)    # reg x5 = v[k]
72                lw      t2, 4(t1)    # reg x7 = v[k+1]
73                sw      t2, 0(t1)    # v[k] = reg x7
74                sw      t0, 4(t1)    # v[k+1] = reg x5
75                jalr    x0, x1, 0   # Return to calling routine
```

前面在第52行，將v[j] 中的j 暫存入a1中，而第69行便將a1*4 暫存入t1中，接著把a0，也就是array data 的base address + j*4 存入t1，也就是說t1現在暫存著v[j] 在memory中的address。
第71行把 v[j]中的值load進reg t0中，第72行把 v[j+1]中的值load進reg t2中，並把t2中的值寫入原本v[j]中，t1寫入v[j+1]中，完成了swap，值得說的是這裡為address+4，而不是+8，因為array是存word。寫入完後便跳回caller: bubble sort procedure。

最後來介紹每個print 的procedure

```
78  printResult1: # print str1
79              la      a1, str1
80              li      a0, 4
81              ecall
82              # print new line
83              la      a1, newline
84              li      a0, 4
85              ecall
86
87              jalr    x0, x1, 0   # Return to calling routine
```

```
la      a1, str1
```

load the address of str1 in data section into a1

```
li      a0, 4
ecall
```

Prints the null-terminated string located at address in `a1`

```
la      a1, newline
```

load the address of newline in data section into a1

```
li      a0, 4
ecall
```

Prints the null-terminated string located at address in `a1`

```
jalr    x0, x1, 0
```

Return to the caller procedure

```
89  printResult2: # print str2
90              la      a1, str2
91              li      a0, 4
92              ecall
93              # print new line
94              la      a1, newline
95              li      a0, 4
96              ecall
97
98              jalr    x0, x1, 0   # Return to calling routine
```

```
la      a1, str2
```

load the address of str2 in data section into a1

```
li      a0, 4
ecall
```

Prints the null-terminated string located at address in `a1`

```
la      a1, newline
```

load the address of newline in data section into a1

```
li        a0, 4
ecall
```

Prints the null-terminated string located at address in `a1`

```
jalr    x0, x1, 0
```

Return to the caller procedure

```
101    printResult3: # printf("\n")
102                  la      a1, newline
103                  li      a0, 4
104                  ecall
105                  jalr     x0, x1, 0   # Return to calling routine
```

```
la        a1, newline
```

load the address of newline in data section into a1

```
li        a0, 4
ecall
```

Prints the null-terminated string located at address in `a1`

```
jalr    x0, x1, 0
```

Return to the caller procedure

```
107    printArray:
108
109            li      t0, 0 # i = 0 (register x5 = 0)
110    loop:
111            slli    t1, t0, 2 # x6 = i * 4
112            add     t2, a3, t1 # x7 = address of array[i]
113            lw      t3, 0(t2)
114            # Print data[i] value
115            mv      a1, t3
116            li      a0, 1
117            ecall
118            # Print space
119            la      a1,  space
120            li      a0,     4
121            ecall
122            # Increment
123            addi    t0, t0, 1 # i++
124            blt     t0, a2, loop
125            # Pinrt newline
126            la      a1,  newline
127            li      a0,     4
128            ecall
129            # Return
130            jalr    x0 x1 0
```

```
li      t0, 0
```

load immediate 0 into t0

i = 0

```
loop:
```

the loop block

```
slli    t1, t0, 2
```

t1 = i*4

```
add     t2, a3, t1
```

t2 = base address of array data + i*4

```
lw      t3, 0(t2)
```

load the value at t2 in data section into t3

```
mv      a1, t3
```

move the value in t3 to a1

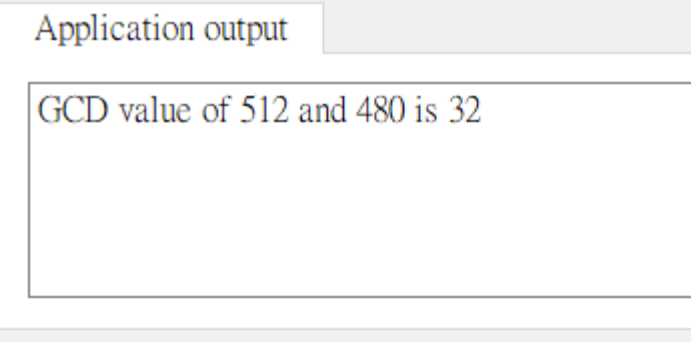
```
li      a0, 1
ecall
```

Prints the value located in `a1` as a signed integer

```
la      a1,  space
```
load the address of space in data section into a1
```
li      a0,     4
ecall
```
Prints the null-terminated string located at address in `a1`
```
addi    t0, t0, 1
```
increment t0
i++
```
blt     t0, a2, loop
```
check stopping condition
if t0 is lesser than a2(argument) then go to the next iteration
If not · do the rest
```
la       a1, newline
```
load the address of newline in data section into a1
```
li       a0, 4
ecall
```
Prints the null-terminated string located at address in `a1`
```
jalr    x0, x1, 0
```
Return to the caller procedure

2. Greatest Common Divisor gcd.s

Output of gcd.s



Explanation:

```
.data
```
data section
```
N1: .word 512
```
a static variable N1, which is a word and stores 512
```
N2: .word 480
```
a static variable N2, which is a word and stores 480

```
str1: .string "GCD value of "
```
a static variable str1, which is a string and stores “GCD value of “
```
str2: .string " and "
```
a static variable str2, which is a string and stores “ and “
```
str3: .string " is "
```
a static variable str3, which is a string and stores “ is “
```
.text
```
text section
```
main:
```
main procedure
```
lw        a0, N1
```
load word N1 into a0
```
lw        a1, N2
```
load word N2 into a1
```
mv        x21, a0
```
move the value of 512 to x21
```
mv        x22, a1
```
move the value of 480 to x21
```
jal       ra, gcd
```
jump and link to gcd procedure

After we return from gcd procedure, we’ve lost the value of N1 and N2 in a0, a1 so we have to load them back.

```
lw        a2, N1
```
load N1 into a2
```
lw        a3, N2
```
load N2 into a3
```
jal       ra, printResult
```
jal and link to printResult procedure to print out the result
```
li        a0, 10
    ecall
```
Exit program
```
gcd:
```
gcd procedure
```
addi      sp,sp, -16
```
every time gcd is called, create a stack frame for 2 items
```
sw        ra, 8(sp)
```
save the return address in stack
```
bne       x22, zero, L1
```
if x22 is not equal to zero, go to L1 block.
```
mv        a0, x21
```
if x22 is equal to zero, then move the content in x21 to a0

```
mv        a1, x22
```
move the content in x22 to a1
```
jalr      x0, x1, 0
```
return to the caller
```
L1:
```
L1 block
```
mv        t1, x22
```
make a copy of the value in x22 in t1 # t1 = n
```
rem       x22, x21, x22
```
x22 = x21 % x22 # r = m % n
```
mv        x21, t1
```
switch the initial value of x22 into x21
```
jal       ra, gcd
```
jump and link to another gcd procedure
```
lw        ra, 8(sp)
```
every time a child gcd procedure we have to restore ra to be able to jump back to the caller correctly.
```
 addi      sp, sp, 16
```
pop the stack space create by child gcd procedure
```
jalr      x0, x1, 0
```
return to the caller

```
printResult:
```
printResult procedure

```
mv  t0, a0
```
move a0 to t0
```
mv  t1, a1
```
move a1 to t1
```
la        a1, str1
```
load the address of str1 in data section into a1
```
li        a0, 4
ecall
```
Prints the null-terminated string located at address in `a1`
```
mv      a1, a2
move the value in a2 to a1
li      a0, 1
ecall
```
Prints the value located in `a1` as a signed integer
```
la        a1, str2
```
load the address of str2 in data section into a1
```
li        a0, 4
ecall
```

Prints the null-terminated string located at address in `a1`

```
mv      a1, a3
```

move the value in a3 to a1

```
li      a0, 1
ecall
```

Prints the value located in `a1` as a signed integer

```
la       a1, str3
```

load the address of str2 in data section into a1

```
li       a0, 4
ecall
```

Prints the null-terminated string located at address in `a1`

```
mv      a1, t0
```

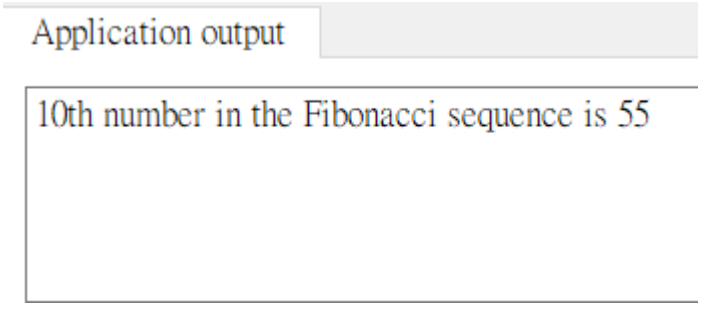
move the value in t0 to a1

```
li      a0, 1
ecall
```

Prints the value located in `a1` as a signed integer

3. Fibonacci Sequence fibonacci.s

Output of fibonacci.s



Explanation:

```
.data
```

data section

```
argument: .word 10
```

a static variable argument, which is a word and stores 10

```
str1: .string "th number in the Fibonacci sequence is "
```

a static variable str1, which is a string and stores “th number in the Fibonacci sequence is “

```
.text
```

text section

```
main:
```

main procedure

```
lw       a0, argument
```

load word argument into register a0

```
jal       ra, fibonacci
```

jump and link to fibonacci procedure

After jumping back from the callee

```
lw        a0, argument
```

load argument back to a0

```
jal       ra, printResult
```

jump and link to printResult procedure

After jumping back from printResult procedure

```
li        a0, 10
ecall
```

exit program

```
fibonacci:
```

fibonacci procedure

```
addi      sp, sp, -16
```

Every time fibonacci procedure is called, push 2 items into stack.

```
sw        ra, 8(sp)
```

save the return address in stack

```
sw        a0, 0(sp)
```

save the content in a0 into stack

```
bne       a0, zero, ElseIf
```

if the content in a0 is not equal to zero, go to ElseIf block

```
addi      x22, x22, 0
```

Otherwise, add zero to our result in register x22

```
lw        ra, 8(sp)
```

Restore return address from stack

```
addi      sp, sp, 16
```

pop 2 items

```
ret
```

Return to the caller

```
ElseIf:
```

ElseIf block

```
mv        a2, zero
addi      a2, a2, 1
```

let a2 = 1

```
bne       a0, a2, Else
```

if the content in a0 is not equal to one, go to Else block

```
addi      x22, x22, 0
```

Otherwise, add one to our result in register x22

```
lw        ra, 8(sp)
```

Restore return address from stack

```
addi      sp, sp, 16
```

pop 2 items

```
ret
```

Return to the caller

```
Else:
```

Else block

```
addi      a0, a0, -1
```

a0 = a0 - 1, which means n --

```
jal       ra  fibonacci
```

jump and link to another fibonacci procedure(child procedure)

```
lw        a0, 0(sp)
```

Since we store our result in x22 (used as a global variable)
we can safely store back the value in 0(sp) back to a0. a0 = n again

```
addi      a0, a0, -2
```

a0 = a0 - 2, which means n = n -2

```
jal       ra  fibonacci
```

jump and link to another fibonacci procedure(child procedure)(the sibling of the child procedure above)

```
lw        ra, 8(sp)
```

Restore return address from stack

```
addi      sp, sp, 16
```

pop 2 items

```
ret
```

Return to the caller

```
printResult:
```

printResult procedure

```
mv        t0, x22
```

move the result, stored in x22, to t0

```
mv        t1, a1
```

move the argument, stored in a1, into t1

```
mv      a1, t1
```

```
move the value in t1 to a1
```

```
li      a0, 1
ecall
```

Prints the value located in `a1` as a signed integer

```
la        a1, str1
```

load the address of str1 in data section into a1

```
li        a0, 4
ecall
```

Prints the null-terminated string located at address in `a1`

```
mv      a1, t0
```
move the value in a3 to a1
```
li      a0, 1
ecall
```
Prints the value located in `a1, which is the result,` as a signed integer
```
ret
```
Return to the caller