T.C.

MARMARA UNIVERSITY

FACULTY of ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE3038 - Computer Organization - Spring 2024

Project #1 Report

Group Members

150119639 - Erdem PEHLİVANLAR

150119553 - Onur ALKURT

150120530 - Enes Taha Karaduman

150119066 - Ertan Karaoğlu

In order to analyze the questions throughout the project and translate them into the MIPS procedure, we first analyzed the questions and then created C source code to meet our expectations in the questions. We will take this approach throughout the report. For each question, we will proceed comparatively by comparing what we did in the C code and the MIPS procedure.

Q1)

To start with the first question, we have an equation given to us in the question. This equation is; f(x) = a* f(x-1) + b* f(x-2) - 2. We need to get some inputs from the user, the first one is our coefficients. We get the coefficients a and b in this equation. Then it expects us to get the first two elements of the sequence. We can call them x0 and x1. Finally, the expected input value is the number of elements of the sequence the user wants to get, this value must be greater than 1. If it is not, we expect an error message, and it should ask the same question again. We can also call this value n. Below is a screenshot of the C code we have created according to these expectations.

```c
#include <stdio.h>
#include <stdlib.h>


int main() {
    int a[2], b[100];
    int n;
    printf("Please enter the coefficients: ");
    scanf("%d %d", &a[0], &a[1]);

    printf("Please enter first two numbers of the sequence: ");
    scanf("%d %d", &b[0], &b[1]);

    printf("Enter the number you want to calculate (it must be greater than 1): ");
    scanf("%d", &n);

    while (n <= 1) {
        printf("Invalid number, Enter a number that is greater than 1: ");
        scanf("%d", &n);
    }

    int i;
    for (i = 2; i < n; i++) {
        b[i] = a[0] * b[i - 1] + a[1] * b[i - 2] - 2;
    }

    printf("Output: %d-th element of the sequence is %d.", n, b[n - 1]);
    return 0;
}
```

If we need to explain the C code we wrote above, we see the lines in box 1 where we get the input values we need to get from the user. In box 2, we ask the user to enter a new n value if the n value we receive is greater than 1 or not, and this process continues until the n value is greater than 1. In box 3, we solve the equation for each element from 2 to n and add it as b to the string we defined in box 1. The reason we start from 2 here is that we have already received the first two string elements from the user. Finally, in the 4th box, we print the n-1st element of the string b, which we have just added in the 3rd box, to the screen. The reason we use n-1 here is that the index of the strings starts from 0, as in simple programming logic. The operation performed in the MIPS procedure lines that I have provided below

corresponds to the lines that we receive input values from the user, which we have also implemented in C code. And checking whether the n value we receive from the user is greater than 1 is also provided here. The operations performed in boxes 1 and 2, indicated by the numbering above, are here translated into the MIPS procedure.

```
    move $s0, $zero

li    $v0, 4
    la   $a0, message      # Please enter the coefficients:
    syscall

    li $v0, 5           # syscall code for reading integer
    syscall
    move $t0, $v0       # store the integer read into $t0
    sw $t0, a_ls($s0)

    addi $s0, $s0, 4
    li $v0, 5           # syscall code for reading integer
    syscall
    move $t0, $v0       # store the integer read into $t0
    sw $t0, a_ls($s0)

    move $s0, $zero
    li   $v0, 4
    la   $a0, message2      # Please enter the coefficients:
    syscall

    li $v0, 5           # syscall code for reading integer
    syscall
    move $t0, $v0       # store the integer read into $t0
    sw $t0, b_ls($s0)

    addi $s0, $s0, 4
    li $v0, 5           # syscall code for reading integer
    syscall
    move $t0, $v0       # store the integer read into $t0
    sw $t0, b_ls($s0)

    li   $v0, 4
    la   $a0, message3      # Please enter the coefficients:
    syscall

    li $v0, 5           # syscall code for reading integer
    syscall
    move $s0, $v0       # store the integer read into $t0

    addi $t0, $zero, 1
loop1:  bgt  $s0, $t0, exit1
    li   $v0, 4
    la   $a0, errMsg      # Please enter the coefficients:
    syscall
    li $v0, 5           # syscall code for reading integer
    syscall
    move $s0, $v0       # store the integer read into $t0
    j loop1


exit1:  la   $t1, a_ls        # $t1 = base addr. of a_ls
    la   $t2, b_ls        # $t2 = base addr. of b_ls
    addi $t0, $zero, 2    # $t0 = i = 2
```

Here are the MIPS procedure equivalents of the operations we performed in lines 2 and 3, which we numbered in C code. In the loop, which continues as long as the value of i is less than n, we can see the steps to perform operations for each index and these operations go to the list we have created. Finally, here is the part where the output message is printed on the screen.

```
loop2:  bge  $t0, $s0, exit2  # i < n
    sll  $t3, $t0, 2      # $t3 = 4i
    add  $t4, $t2, $t3    # $t4 = addr. of b[i]
    addi $t5, $t0, -1     # $t5 = i - 1
    sll  $t5, $t5, 2      # $t5 = 4*(i-1)
    add  $t5, $t2, $t5    # $t5 = addr. of b[i - 1]
    addi $t6, $zero, 4    # $t6 = 4
    add  $t6, $t1, $t6    # $t6 = addr. of a[1]
    addi $s1, $t0, -2     # $s1 = i - 2
    sll  $s1, $s1, 2      # $s1 = 4*(i-2)
    add  $s1, $t2, $s1    # $s1 = addr. of b[i - 2]
    la   $s2, a_ls        # $s2 = addr. of a[0]
    lw   $s3, 0($s2)      # $s3 = a[0]
    lw   $s4, 0($t5)      # $s4 = b[i - 1]
    lw   $s5, 0($t6)      # $s5 = a[1]
    lw   $s6, 0($s1)      # $s6 = b[i - 2]
    mul  $s3, $s3, $s4    # $s3 = a[0] * b[i - 1]
    mul  $s4, $s5, $s6    # $s4 = a[1] * b[i - 2]
    add  $s3, $s3, $s4    # $s3 = a[0] * b[i - 1] + a[1] * b[i - 2]
    addi $s3, $s3, -2     # $s3 = a[0] * b[i - 1] + a[1] * b[i - 2] - 2
    sw   $s3, 0($t4)      # b[i] = a[0] * b[i - 1] + a[1] * b[i - 2] - 2
    addi $t0, $t0, 1      # i++
    j    loop2
    exit2:  li   $v0, 4
        la   $a0, outputMsg      # Please enter the coefficients:
        syscall
        li $v0, 1          # syscall for print_int
        move $a0, $s0      # load integer from register $t0 (you can replace $t0 with $a0)
        syscall            # print the integer
        li   $v0, 4
        la   $a0, outputMsg2      # Please enter the coefficients:
        syscall

        addi $t0, $s0, -1  # $t0 = n - 1
        sll  $t0, $t0, 2   # $t0 = 4*(n - 1)
        add  $t0, $t2, $t0 # $t0 = addr. of b[n - 1]
        lw   $t3, 0($t0)   # $t3 = b[n - 1]

        li $v0, 1          # syscall for print_int
        move $a0, $t3      # load integer from register $t0 (you can replace $t0 with $a0)
        syscall
        li   $v0, 4
        la   $a0, dot      # Please enter the coefficients:
        syscall
        li $v0, 10
        syscall
```

Now, in the screenshots I have provided below, we can see the results we get when running question 1.

Example Run 1: In the first example, we gave the n value as true without getting an error and it printed our expected result on the screen.

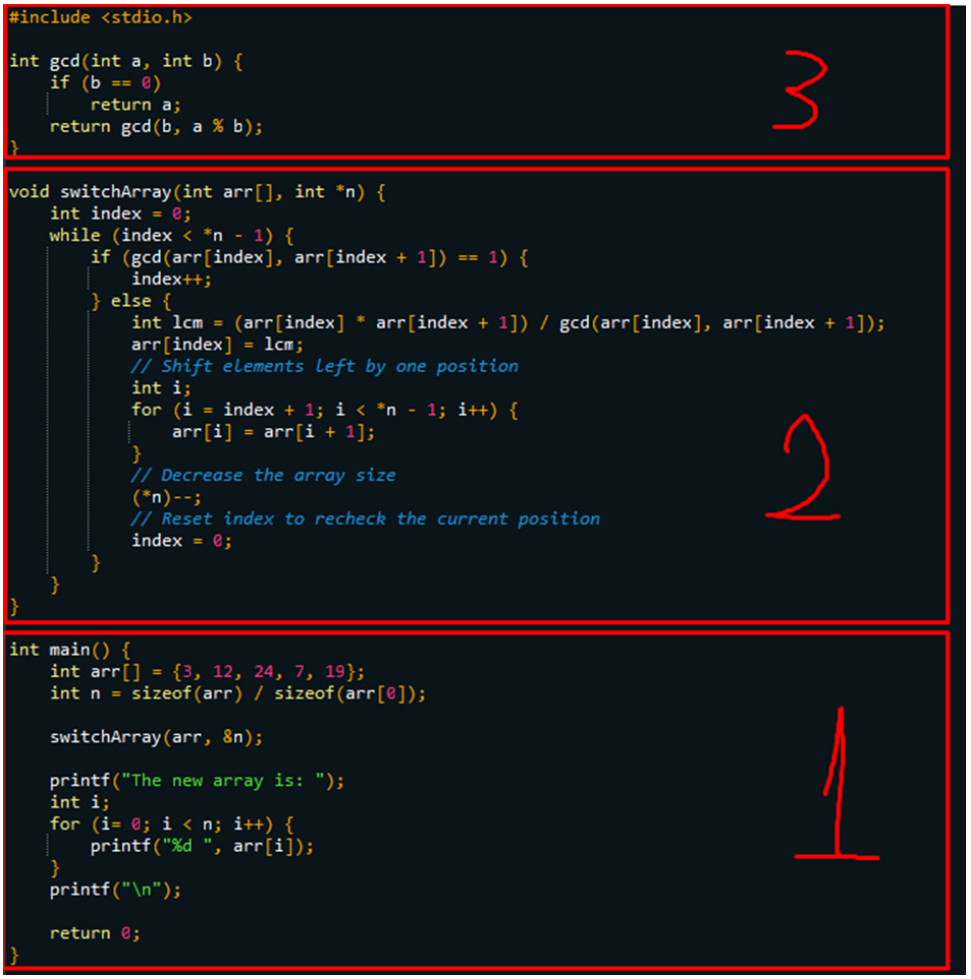
```
Please enter the coefficients: 2
1
Please enter first two numbers of the sequence: 1
2
Enter the number you want to calculate (it must be greater than 1): 4
Output: 4th element of the sequence is 6.
-- program is finished running --
```

Example Run 2: Now let's create a new example by entering the values of another array in the examples given in the project file, but let's also check whether the value of n is greater than 1 in this example.

```
Please enter the coefficients: 6
1
Please enter first two numbers of the sequence: 0
1
Enter the number you want to calculate (it must be greater than 1): 1
Invalid number, Enter a number that is greater than 1: 0
Invalid number, Enter a number that is greater than 1: 4
Output: 4th element of the sequence is 23.
-- program is finished running --
```

Q2)

In the second question, we need to check whether each binary neighboring element in the list we receive as input from the user is prime. If the adjacent pair we are considering is not prime, we need to remove these two numbers from the list and instead place the least common multiple of the two numbers in the list. After making such a change, we need to start the list again and check it in two neighboring groups from the beginning. In this way, we will print our new updated array in the last output we created.



```c
#include <stdio.h>

int gcd(int a, int b) {
    if (b == 0)
        return a;
    return gcd(b, a % b);
}
```

3

```c
void switchArray(int arr[], int *n) {
    int index = 0;
    while (index < *n - 1) {
        if (gcd(arr[index], arr[index + 1]) == 1) {
            index++;
        } else {
            int lcm = (arr[index] * arr[index + 1]) / gcd(arr[index], arr[index + 1]);
            arr[index] = lcm;
            // Shift elements left by one position
            int i;
            for (i = index + 1; i < *n - 1; i++) {
                arr[i] = arr[i + 1];
            }
            // Decrease the array size
            (*n)--;
            // Reset index to recheck the current position
            index = 0;
        }
    }
}
```

2

```c
int main() {
    int arr[] = {3, 12, 24, 7, 19};
    int n = sizeof(arr) / sizeof(arr[0]);

    switchArray(arr, &n);

    printf("The new array is: ");
    int i;
    for (i= 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

1

Here, in box 1, we get the input list we need from the user, and then calculate the number of elements in this list. We send the list entered by the user and the number of elements to the switchArray method. We print the value returned by the switchArray method to the screen with a for loop. This value that we printed with the for loop is our Output value, that is, the updated value of our list.

In the second box, the switchArray method, we check the binary neighbors one by one. If we decide that these are prime in this method, we add their least common product to the list. If we update our list in this way, we need to start the for loop and check the neighbor pairs one by one from the beginning. While doing these operations here, we also need the greatest common divisor method, which is marked in box number 3.

Here we display all of the transactions in box 1 translated into the MIPS procedure.

```
main:
    li   $v0, 4
        la   $a0, message      # Please enter the coefficients:
        syscall


        li $v0, 5          # syscall code for reading integer
        syscall
        la $t4, n
    sw $v0, 0($t4)     # $t0 = n

    li   $v0, 4
        la   $a0, message2     # Please enter the coefficients:
        syscall

    lw   $t0, 0($t4)
    move $t1, $zero    # $t1 = i = 0
loop1:  bge  $t1, $t0, exit1
    li $v0, 5          # syscall code for reading integer
        syscall
    move $t2, $v0        # $t2 = input
    sll  $t3, $t1, 2     # $t3 = 4i
    sw   $t2, array($t3) # array[i] = input
    addi $t1, $t1, 1     # i++
    j    loop1
exit1:
    la $t5, array        # $t5 = base addr. of array
    move $a0, $t5        # $a0 = array
    move $a1, $t4        # $a1 = &n
    jal  switchArray



        li   $v0, 4
        la   $a0, outputMsg     # Please enter the coefficients:
        syscall

        move $t0, $zero         # i = 0
        la   $t1, n             # $t1 = base address of n
        la   $t4, array          # $t4 = base addr. of array
        lw   $t2, 0($t1)        # t2 = n
    loop4:  bge  $t0, $t2, exit4    # i < n
        sll  $t3, $t0, 2        # $t3 = 4i
        add  $t5, $t4, $t3      # $t5 = addr. of array[i]
        lw   $t6, 0($t5)        # $t6 = array[i]
    move $a0, $t6
    li $v0, 1
    syscall
    li   $v0, 4
        la   $a0, space     # Please enter the coefficients:
        syscall
        addi $t0, $t0, 1
        j    loop4

    exit4:  li $v0, 10
        syscall
```

Here, the MIPS procedure of the switch Array section specified in the 2nd box is given below.

```
switchArray:
    addi $sp, $sp, -24
    sw   $ra, 20($sp)
    sw   $t0, 16($sp)      # $t0 = index
    sw   $t1, 12($sp)      # $t1 = lcm
    sw   $t2, 8($sp)       # $t2 = i
    sw   $a0, 4($sp)
    sw   $a1, 0($sp)

    la   $s1, array       # $s1 = base addr. of array
    move $t0, $zero       # index = 0
loop2:  lw   $s0, 0($a1)      # $s0 = *n
    addi $s0, $s0, -1     # $s0 = *n - 1
    bge  $t0, $s0, ret3   # index < *n - 1
    sll  $t3, $t0, 2      # $t3 = 4*index
    add  $t3, $t3, $s1    # $t3 = 4*index + array = array[index]
    addi $t4, $t0, 1      # $t4 = index + 1
    sll  $t4, $t4, 2      # $t4 = 4*(index+1)
    add  $t4, $t4, $s1    # $t4 = 4*(index+1) + array = array[index + 1]
    lw   $t5, 0($t3)      # $t5 = array[index]
    lw   $t6, 0($t4)      # $t6 = array[index + 1]
    move $a0, $t5         # $a0 = array[index]
    move $a1, $t6          # $a1 = array[index + 1]
    jal  gcd
    lw   $a0, 4($sp)
    lw   $a1, 0($sp)
    move $t3, $v0         # $t3 = result of gcd(arr[index], arr[index + 1]
    addi $t4, $zero, 1    # $t4 = 1
    bne  $t3, $t4, cond2
    addi $t0, $t0, 1      # index++
    j    loop2
cond2:  mul  $t4, $t5, $t6    # $t4 = arr[index] * arr[index + 1]
    move $a0, $t5         # $a0 = array[index]
    move $a1, $t6          # $a1 = array[index + 1]
    jal  gcd
    lw   $a0, 4($sp)
    lw   $a1, 0($sp)
    move $t3, $v0         # $t3 = result of gcd(arr[index], arr[index + 1]
    div  $t4, $t4, $t3    # $t4 = arr[index] * arr[index + 1]) / gcd(arr[index], arr[index + 1]
    move $t1, $t4         # $t1 = lcm = arr[index] * arr[index + 1]) / gcd(arr[index], arr[index + 1]
    sll  $t3, $t0, 2      # $t3 = 4 * index
    add  $t3, $a0, $t3    # $t3 = addr of array[index]
    sw   $t1, 0($t3)      # array[index] = lcm
    addi $s2, $t0, 1      # $s2 = index + 1
    move $t2, $s2         # i = index + 1
    lw   $s3, 0($a1)      # $s3 = *a
    addi $s3, $s3, -1     # $s3 = *a - 1


loop3:  bge  $t2, $s3, exit3 # i < *n - 1
    sll  $s4, $t2, 2      # $s4 = 4i
    add  $s4, $s4, $a0    # $s4 = addr. of array[i]
    addi $s5, $t2, 1      # $s5 = i + 1
    sll  $s5, $s5, 2      # $s5 = 4*(i+1)
    add  $s5, $s5, $a0    # $s5 = addr. of array[i + 1]
    lw   $s6, 0($s5)      # $s6 = array[i + 1]
    sw   $s6, 0($s4)      # array[i] = array[i + 1]
    addi $t2, $t2, 1      # i++
    j    loop3
exit3:  sw   $s3, 0($a1)      # (*n)--
    move $t0, $zero       # index = 0
    j    loop2

ret3:   lw   $ra, 20($sp)
    lw   $t0, 16($sp)      # $t0 = index
    lw   $t1, 12($sp)      # $t1 = lcm
    lw   $t2, 8($sp)      # $t2 = i
    lw   $a0, 4($sp)
    lw   $a1, 0($sp)
    addi $sp, $sp, 24
    jr   $ra
```

Here we display the MIPS procedure that allows us to find the greatest common divisor.

```
gcd:
        addi $sp, $sp, -4
        sw   $t0, 0($sp)       # $t0 = temp
    loop8:  beq  $a1, $zero, ret8 # b != 0
        move $t0, $a1          # temp = b
        div  $a0, $a1          # a / b
        mfhi $t1            # a % b
        move $a1, $t1           # b = a % b
        move $a0, $t0          # a = temp
        j    loop8
    ret8:   lw   $t0, 0($sp)       # $t0 = temp
        addi $sp, $sp, 4
        move $v0, $a0          # $v0 = a, return a
        jr   $ra
```

Below are screenshots of the results we got while running the MIPS procedure prepared for this question.

```
Input Size: 6
Input: 6
4
3
2
7
13
The new array is: 12 7 13
-- program is finished running --
```

```
Input Size: 7
Input: 25
2
3
9
6
4
5
The new array is: 25 36 5
-- program is finished running --
```

Q3)

In the third question, we are expected to write a MIPS procedure that recursively shuffles the string we receive from the user. In this question, we used the same approach as in the previous questions. In other words, after analyzing the question, we created the C code in accordance with our analysis and created the MIPS procedure in accordance with this C code. Here we are expected to get two inputs from the user, the first of these is the string value that the user wants to mix and the second is the number of times the mixing process will be done, which we can characterize as the n value. The code we need to write is based on splitting the string in half and replacing the strings we split. And we continue this process iteratively as many times as the n value we receive from the user. The word "Computer" given as an example in the project should turn into "retupmoC" as output after our program runs. You can view the C code we have created accordingly below.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void swap(char *a, char *b, int n) {
    int i;  // "Computer"
    for (i = 0; i < n; i++) {
        char temp = *(a + i);
        *(a + i) = *(b + i);
        *(b + i) = temp;
    }
}

void shuffle(char *str, int len, int level, int n) {
    if (level == n)
        return;

    int half = len / 2;
    shuffle(str, half, level + 1, n);
    shuffle(str + half, half, level + 1, n);
    swap(str, str + half, half);
}

int main() {
    char str[] = "Computer";
    int n = 3;

    printf("Input: \"%s\"\n", str);
    shuffle(str, strlen(str), 0, n);
    printf("Output: \"%s\"\n", str);

    return 0;
}
```

Here, in box 1, it is displayed that the input values we will receive from the user are received and after these values are received, the relevant method is called and then the value returned by this method is printed on the screen as Output. (In box 1, the input values to be received from the user are given as fixed to ensure a faster process, but in the MIPS procedure, this situation is arranged so that input is received from the user.) In box 2, we display the shuffle method called within the main function. This method includes the string we receive from the user as parameters, the length of this string, a level value (given as a constant 0), and the n value that determines the number of iterative operations we receive from the user. By taking the numerical value of half of the string we received in the method, we call the same method again with different parameters within the method. After doing this twice, we call the swap method. Swap method is our method that allows us to change the position of letters in the given string.

Now we can take a look at the MIPS procedure equivalent of the C code I have roughly explained. Here is the part where we get the "string" and "n" values we need to get from the user. We find the length value of the string value we received with the loop here and assign it to a register. Here is the section where we print the method we need to call afterwards and the output message based on the result. In summary, the operations that must be performed in box 1, which we have marked on the C code, are listed here.

```
.data

message: .asciiz "Input: "
outputMsg: .asciiz "Output: "
input: .space 64

.text

    main:
        li   $v0, 4
            la   $a0, message    # Please enter the coefficients:
            syscall

            li $v0, 8            # syscall code for reading string
            la $a0, input        # load address of buffer
            li $a1, 64           # maximum number of characters to read
            syscall

        li $v0, 5            # syscall code for reading integer
            syscall
            move $s0, $v0        # $s0 = n


        move $t4, $zero      # $t4 = len = 0
        la   $t0, input       # $t0 = addr. of input
        move $t1, $zero      # $t1 = 0 = i
    loop1:  add  $t2, $t0, $t1   # $t2 = addr. of input[i]
        lb   $t3, 0($t2)     # $t3 = input[i]
        beq  $t3, $zero, exit1
        addi $t4, $t4, 1     # counter++
        addi $t1, $t1, 1     # i++
        add  $t2, $t2, $t1
        j    loop1

    exit1:  addi $t4, $t4, -1
        move $a0, $t0     # $a0 = str
        move $a1, $t4     # $a1 = len
        move $a2, $zero   # $a2 = 0
        move $a3, $s0     # $a3 = n
        jal  shuffle
        li   $v0, 4
            la   $a0, outputMsg    # Please enter the coefficients:
            syscall
            li   $v0, 4
            la   $a0, 0($t0)    # Please enter the coefficients:
            syscall

            li    $v0, 10
        syscall
```

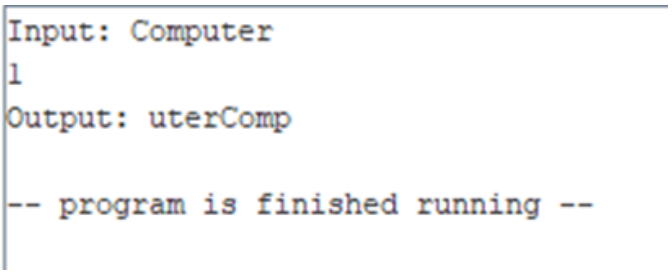
Here is the MIPS procedure, where we display the transactions taking place in box 2, that is, the shuffle method, which we specified in the C code. The reason why the process here is a little different compared to other examples is that the same method is called more than once within the shuffle method. Therefore, a MIPS procedure was designed here in accordance with the stack structure.

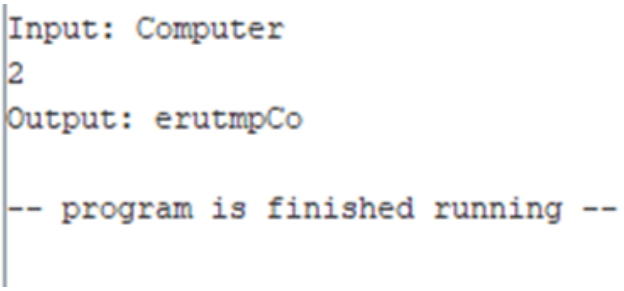
```
shuffle:
        addi $sp, $sp, -24
        sw   $ra, 20($sp)
        sw   $a0, 16($sp)
    sw   $a1, 12($sp)
    sw   $a2, 8($sp)
    sw   $a3, 4($sp)
    sw   $s0, 0($sp)      # $s0 = half
    beq  $a2, $a3, ret
    srl  $s0, $a1, 1    # half = len / 2
    move $a1, $s0
    addi $a2, $a2, 1    # level + 1
    jal  shuffle
    add  $a0, $a0, $s0 # str + half
    jal  shuffle
    lw   $a0, 16($sp)
    add  $a1, $a0, $s0
    move $a2, $s0
    jal  swap
ret:    lw   $ra, 20($sp)
        lw   $a0, 16($sp)
    lw   $a1, 12($sp)
    lw   $a2, 8($sp)
    lw   $a3, 4($sp)
    lw   $s0, 0($sp)
    addi $sp, $sp, 24
    jr   $ra
```

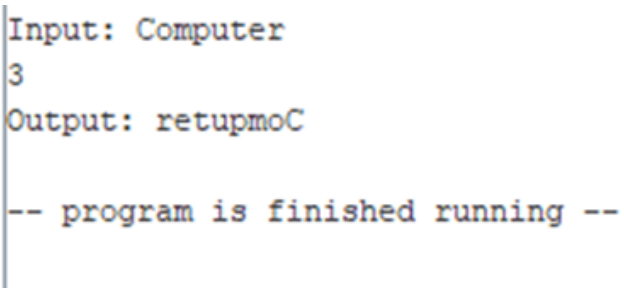
Finally, we see the structure of the swap method in the MIPS procedure. Here too, the procedure was designed in accordance with the stack structure.

```
swap:
        addi $sp, $sp, -8
        sw   $t0, 4($sp)    # $t0 = i
        sw   $t1, 0($sp)    # $t1 = temp
        move $t0, $zero
    loop2:  bge  $t0, $a2, exit2
        add  $t2, $a0, $t0 # $t2 = a + i
        lb   $t3, 0($t2)    # $t3 = *(a + i)
        move $t1, $t3       # temp = *(a + i)
        add  $t4, $a1, $t0 # $t4 = b + i
        lb   $t5, 0($t4)    # $t5 = *(b + i)
        sb   $t5, 0($t2)    # *(a + i) = *(b + i)
        sb   $t1, 0($t4)    # *(b + i) = temp
        addi $t0, $t0, 1    # i++
        j    loop2
exit2:  lw   $t0, 4($sp)    # $t0 = i
        lw   $t1, 0($sp)    # $t1 = temp
        addi $sp, $sp, 8
    jr   $ra
```

Below are screenshots of the results we got while running the MIPS procedure prepared for this question.







Q4)

The C software that is provided carries out an algorithm that finds the biggest island in a binary matrix and counts the ones on that island. Land is represented by ones in the binary matrix, which is a 2D array called matrix, and water is represented by zeros. As part of the island exploration process, a second 2D array called temp_matrix is initialized to track visited cells. When a cell in temp_matrix contains a 1, the initialization method initialize_temp_matrix() sets that cell to 1 as well. Depth-First Search (DFS), which is carried out via the search_neighbor(row, col) function, is the exploration method used by the algorithm. Starting from a particular cell (row, col), this recursive DFS searches the island, increasing a total counter with each cell visited. Cells marked as visited or out of bounds are not selected. Following DFS, the maximum number of ones in the investigated island is compared with it, and updated if needed. The process is coordinated by the main() function, which iterates through every cell in the initial matrix and starts DFS from cells that include ones that have not been visited. The software publishes the temporary matrix, showing visited cells, then outputs the count of ones in the largest island discovered once all islands have been investigated. This technique tracks the size of each island using DFS and systematically traverses the matrix to identify the largest island.

```c
#include <stdio.h>
#include <stdlib.h>

#define ROW 10
#define COL 10

int total = 0;


int matrix[ROW][COL] = {
                {0, 0, 0, 0, 1, 0, 0, 1, 1, 0},
                {0, 0, 0, 1, 1, 0, 0, 0, 0, 0},
                {0, 0, 0, 1, 1, 0, 1, 1, 1, 0},
                {0, 0, 0, 1, 1, 0, 0, 0, 1, 0},
                {0, 1, 1, 1, 0, 0, 1, 0, 0, 0},
                {1, 1, 0, 1, 1, 0, 0, 0, 0, 0},
                {1, 0, 0, 1, 1, 1, 0, 1, 1, 1},
                {1, 1, 0, 1, 1, 0, 0, 0, 1, 1},
                {0, 1, 1, 0, 1, 0, 0, 0, 1, 1},
                {0, 0, 0, 1, 1, 1, 0, 0, 0, 1}

            };

int temp_matrix[ROW][COL] = {
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
            };
```

We tried to run 10x10 matrix input as it seems

```c
void initialize_temp_matrix(){
    int i, j;
    for(i = 0; i < ROW; i++){
        for(j = 0; j < COL; j++){
            if (matrix[i][j] == 1){
                temp_matrix[i][j] = 1;
            }
        }
    }
}
void search_neighboor(row, col){

    if (row < 0 || row > ROW - 1 || col < 0 || col > COL - 1){
        return;
    }


    if (matrix[row][col] == 0){
        return;
    }

    total++;
    matrix[row][col] = 0;

    search_neighboor(row + 1, col);
    search_neighboor(row - 1, col);
    search_neighboor(row, col + 1);
    search_neighboor(row, col - 1);
}

void print_temp_matrix(){
    int i, j;
    for(i = 0; i < ROW; i++){
        for(j = 0; j < COL; j++){
            printf("%d ", temp_matrix[i][j]);
        }
        printf("\n");
    }
}
```

```c
int main(){
    initialize_temp_matrix();

    int max = 0;
    int i, j;
    for(i = 0; i < ROW; i++){
        for(j = 0; j < COL; j++){
            if(matrix[i][j] == 1){
                search_neighboor(i, j);
                if(total > max){
                    max = total;
                }
                total = 0;
            }
        }
    }

    print_temp_matrix();
    printf("The number of the 1s on the largest island is %d.", max);

    return 0;
}
```

```
exit5:  jal   print_temp_matrix
        li    $v0, 4              # for print space
        la    $a0, message        # for print space
    syscall             # for print space
    move  $a0, $s2          # for print
    li    $v0, 1            # for print
    syscall               # for print
    li    $v0, 4              # for print space
        la    $a0, dot            # for print space
    syscall
        lw    $ra, 12($sp)
        lw    $s0, 8($sp)
        lw    $s1, 4($sp)
        lw    $s2, 0($sp)
    addi  $sp, $sp, 16
    li    $v0, 10
    syscall


initialize_temp_matrix:
    addi $sp, $sp, -8     # for local variables, it is used stack for 2 items
    sw   $s0, 4($sp)      # storing s0, s0 is for i variable
    sw   $s1, 0($sp)      # storing s1, s1 is for j variable
    la   $s2, matrix      # base address of matrix
    lb   $t0, 0($s2)      # $t0 = ROW
    lb   $t1, 1($s2)      # $t1 = COL
    move $s0, $zero       # i = 0
loop1:  bge  $s0, $t0, exit1  # i < ROW
    move $s1, $zero       # j = 0
loop2:  bge  $s1, $t1, exit2  # j < COL
    mul  $t2, $s0, $t1
    add  $t2, $t2, $s1
    addi $t3, $t2, 2
    lb   $s3, matrix($t3)      # $s3 = matrix[i][j]
    sb   $s3, temp_matrix($t2) # temp_matrix[i][j] = matrix[i][j]
incr1:  addi $s1, $s1, 1      # j++
    j loop2               # go loop2
exit2:  addi $s0, $s0, 1      # i++
    j loop1           # go loop1



search_neighboor:                     # $a0 = i, $a1 = j
    addi $sp, $sp, -12
    sw   $ra, 8($sp)
    sw   $a0, 4($sp)
    sw   $a1, 0($sp)
    addi $t3, $t0, -1           # $t3 = ROW - 1
    addi $t4, $t1, -1           # $t4 = COL - 1
    slt  $t5, $a0, $zero         # $t5 = i < 0
    sgt  $t6, $a0, $t3          # $t6 = i > ROW - 1
    slt  $s4, $a1, $zero        # $s4 = j < 0
    sgt  $s5, $a1, $t4          # $s5 = j > COL - 1
    or   $t3, $t5, $t6          # $t3 = i < 0 || i > ROW - 1
    or   $t4, $s4, $s5          # $t4 = j < 0 || j > COL - 1
    or   $t3, $t3, $t4          # or for all of them in $t3
    addi $t4, $zero, 1          # $t4 = 1
    beq  $t3, $t4, ret          # if $t3 == 1 go -> return
    mul  $t3, $a0, $t1
    add  $t3, $t3, $a1
    addi $t3, $t3, 2
    lb   $t4, matrix($t3)             # $t4 = matrix[i][j]
    beq  $t4, $zero, ret        # if (matrix[i][j] == 0) -> go return
    lw   $t4, total           # $t4 = total
    addi $t4, $t4, 1            # total++
    sw   $t4, total             # total = $t4
    move $s4, $zero             # $s4 = 0
    sb   $s4, matrix($t3)             # matrix[i][j] = 0
    addi $a0, $a0, 1            # i+1, j for parameters
    jal  search_neighboor       # and call function recursively
    lw   $a0, 4($sp)
    lw   $a1, 0($sp)
    addi $a0, $a0, -1           # i-1, j for parameters
    jal  search_neighboor       # and call it
    lw   $a0, 4($sp)
    lw   $a1, 0($sp)
    addi $a1, $a1, 1            # i, j+1 for parameters
    jal  search_neighboor       # and call it
    lw   $a0, 4($sp)
    lw   $a1, 0($sp)
    addi $a1, $a1, -1           # i, j-1 for parameters
    jal  search_neighboor       # and call it
```

```
ret:    lw   $ra, 8($sp)
    lw   $a0, 4($sp)
    lw   $a1, 0($sp)
    addi $sp, $sp, 12
    jr   $ra


print_temp_matrix:
        addi $sp, $sp, -8
    sw   $s0, 4($sp)
    sw   $s1, 0($sp)
    la   $s3, matrix
    lb   $t0, 0($s3)       # $t0 = ROW
    lb   $t1, 1($s3)       # $t1 = COL
    move $s0, $zero
loop3:  bge  $s0, $t0, exit3
    move $s1, $zero
loop4:  bge  $s1, $t1, exit4
    mul  $t2, $s0, $t1
    add  $t2, $t2, $s1
    lb   $s4, temp_matrix($t2)      # $s3 = temp_matrix[i][j]
    move $a0, $s4           # for print
    li   $v0, 1             # for print
    syscall                 # for print
    li $v0, 4               # for print space
        la $a0, space           # for print space
    syscall             # for print space
    addi $s1, $s1, 1        # j++
    j loop4                 # go loop4
exit4:  li $v0, 4               # for print \n
        la $a0, newLine         # for print \n
    syscall                 # for print \n
    addi $s0, $s0, 1        # i++
    j loop3
exit3:  lw   $s0, 4($sp)
    lw   $s1, 0($sp)
    addi $sp, $sp, 8
    jr $ra
```

We can look at comments in assembly code, each row comment represents the c code of the instruction as it seems.

Sample Run:

Our input is -> matrix: .byte 10, 10, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1

Result is

```
0 0 0 0 1 0 0 1 1 0
0 0 0 1 1 0 0 0 0 0
0 0 0 1 1 0 1 1 1 0
0 0 0 1 1 0 0 0 1 0
0 1 1 1 0 0 1 0 0 0
1 1 0 1 1 0 0 0 0 0
1 0 0 1 1 1 0 1 1 1
1 1 0 1 1 0 0 0 1 1
0 1 1 0 1 0 0 0 1 1
0 0 0 1 1 1 0 0 0 1
The number of the 1s on the largest island is 28.
-- program is finished running --
```