Lab 1.1

1. Step by step algorithm
   (1) Variable definition in *.data* section
   First, I need to define the 3 variables *var1*, *var2* and *var3*. Since the type and initial value is not specified, I just set them as *.word* type, and initialize them to 0 (explicitly, for clarity).
   Also, since we need to print out the calculated values of *var1*, *var2* and *var3* line by line, we have to print out the new line character. So I additionally defined the new line character *newline* of type *.asciz* and of value "\n".

   (2) Read variables from terminal
   By loading service number 5 into register *a7* by *li* and executing *ecall*, *ReadInt* is performed and the integer input from terminal is stored in register *a0*. Then I use *la* instruction to store the address of *var1*, *var2* and *var3* in register *t1*, *t2* and *t3* respectively, and use *sw* instruction to store the 3 integers read from terminal to the address *(0)t1*, *(0)t2* and *(0)t3*, so that the value of *var1*, *var2* and *var3* become the value read from terminal.

   (3) Increase *var1* by 3
   To change the value of *var1*, first I have to load the value of *var1* from the memory to the register, which is done by the *lw* instruction, storing the value at *(0)t1* (*t1 stores* the address of *var1* since step (2)) to *t4*. Then I use the *addi* to add an immediate value 3 to *t4*. Finally, the value in *t4* is stored back to memory at *(0)t1* using the *sw* instruction.

   (4) multiply *var2* by 2
   To change the value of *var2*, first I have to load the value of *var2* from the memory to the register, which is done by the *lw* instruction, storing the value at *(0)t2* (*t2 stores* the address of *var2* since step (2)) to *t5*. Since integers are stored in binary in computer, multiply a number by 2 or 0b10 is the same with shift every digit in the number left for 1 digit, which could be done by the *slli* instruction. Finally, the value in *t5* is stored back to memory at *(0)t2* using the *sw* instruction.

   (5) Increase *var3* by *var1* + *var2*
   To change the value of *var3*, first I have to load the value of *var3* from the memory to the register, which is done by the *lw* instruction, storing the value at *(0)t3* (*t2 stores* the address of *var3* since step (2)) to *t6*. Then I use *add* instruction twice to first add *t6* by *t4* (which is the value of increased *var1* since step (3)), then add *t6* by *t5* (which is the value of increased *var2* since step (4)). Finally, the value in *t6* is stored back to memory at *(0)t3* using the *sw* instruction.

(6) Print variables to terminal

By loading service number 1 into register *a7* by *li* and executing *ecall*, *PrintInt* is performed and the integer in *a0* will be printed in terminal. So to print the value of *var1*, *var2* and *var3*, I use *lw* instruction to load the value of them to *a0* and perform *ecall* one at a time.

Between printing each variable, we have to print an additional newline character. This is done by loading service number 4 (PrintString) into register *a7* by *li*, and loading the address of variable *newline* to *a0* by *la*, and finally execute the *ecall* instruction.

(7) Exit program

By loading service number 10 into register *a7* by *li* and executing *ecall*, the program will exit with code 0.

2. Main Code:

(1) Variable declaration:

```
.data
var1: .word 0
var2: .word 0
var3: .word 0
newline: .asciz "\n"
```

(2) Variable input

```
### load variable from console
# read input1
li a7, 5
ecall
# put input into var1
la t1, var1          # t1 is the address of var1
sw a0, 0(t1)

# read input2
li a7, 5
ecall
# put input into var2
la t2, var2          # t2 is the address of var2
sw a0, 0(t2)

# read input3
li a7, 5
ecall
# put input into var3
la t3, var3          # t3 is the address of var3
sw a0, 0(t3)
```

(3) Variable update

```
### increase var1 by 3
# addi t4, var1, 3 # add
lw t4, 0(t1)
addi t4, t4, 3
sw t4, 0(t1)        # t4 is value of var1


### multiply var2 by 2
lw t5, 0(t2)
add t5, t5, t5      # multiply
sw t5, 0(t2)        # t5 is value of var2


### increase var3
lw t6, 0(t3)        # t0 is temp sum, now var3
add t6, t6, t4
add t6, t6, t5
sw t6, 0(t3)
```

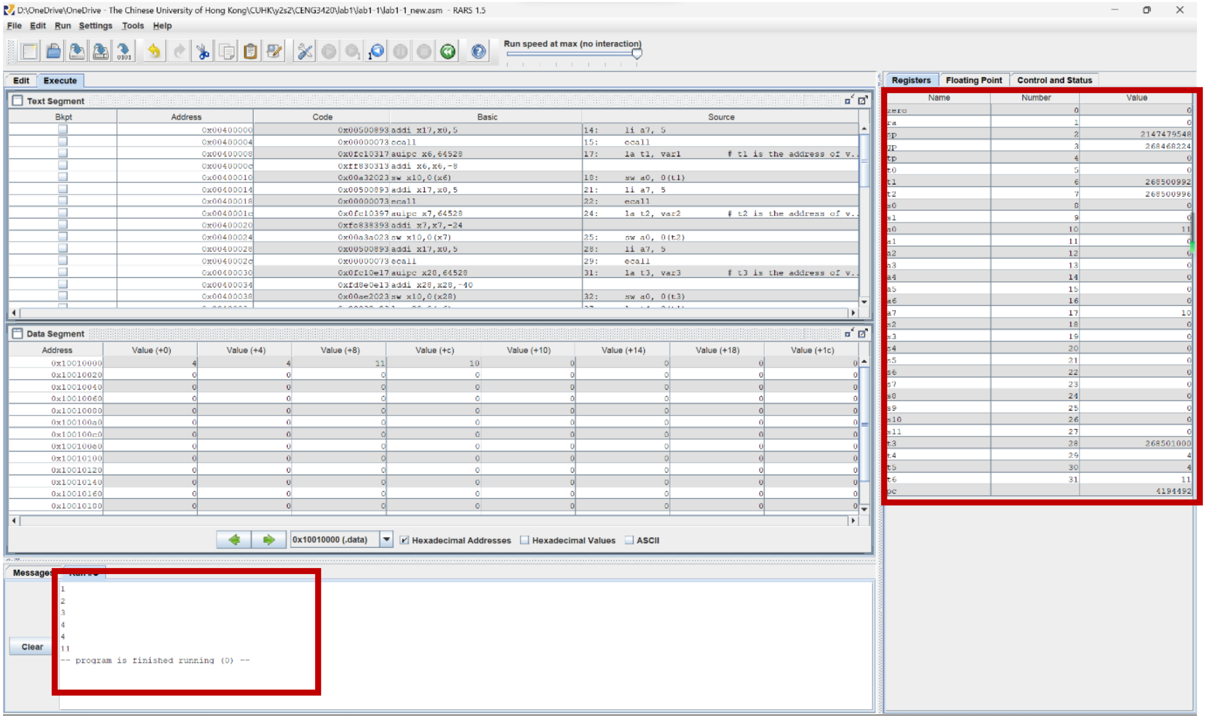
(4) Variable output

```
### print vars
# print var1
li a7, 1    # set to print integer
lw a0, var1# set int to print
ecall       # print

# print new line
li a7, 4
la a0, newline
ecall

# print var2
li a7, 1    # set to print integer
lw a0, var2# set int to print
ecall       # print

# print new line
li a7, 4
la a0, newline
ecall

# print var3
li a7, 1    # set to print integer
lw a0, var3# set int to print
ecall       # print
```

(5) Program exit

```
# exit
li a7, 10
ecall
```

3. Console results



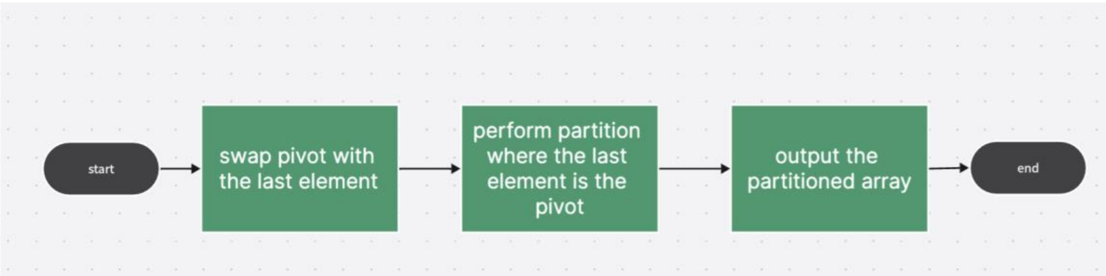
Lab 1.2

1. Problem analysis

   In lab 1.2, we are given a prestored array named *array1* that has 10 elements -1, 22, 8, 35, 5, 4, 11, 2, 1, 78. We want to partition the array with 8 (the 3rd element) as pivot, so that all the numbers smaller than 8 will be stored at the left of it, and all the numbers greater than 8 will be stored at the left of it. In other words, the program output should be -1, 5, 4, 2, 1, 8, 11, 35, 22, 78.

2. Algorithm overview

   Since in practice, the pivot of the partition mission could be different each time, it is tedious to develop a different partitioning algorithms for different pivots.
   So first, no matter where the pivot originally is, I swap it with the last element of the array.
   Then, we only need to develop an algorithm to perform partition where the last element is the pivot.
   Finally, we output the partitioned array.
   Graph representation:

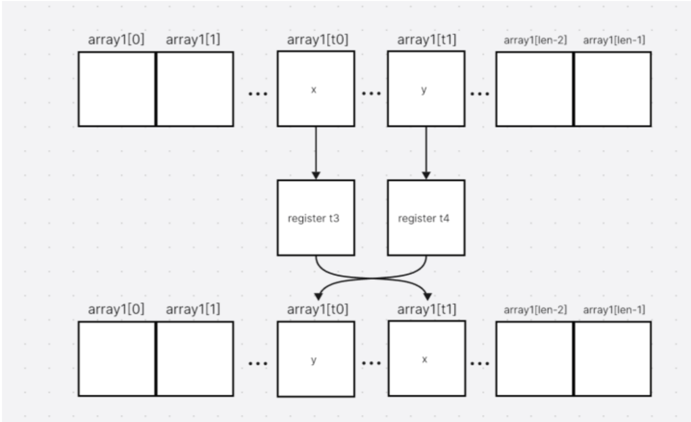


3. Function declaration

   In order to increase the make the program more efficient and portable, I declare functions *swap* and *partition*.
   Since these functions only involve a limited amount of registers, I think there is no needs to use a stack to store the original values of these registers and load them back.

   (1) *Swap*

   Since in both swapping pivot and partitioning array parts, we need to perform a lot of swapping, I declare a swap function to avoid tedious code.
   The function input is the indices of the 2 elements that will be swapped, stored in *t0* and *t1*.
   Since the number with index *i* will be in address *array1 + i * 4*, I first left shift *t0*, *t1* by 2 to get the quadruple of them, then I add them with the address of the first element to get the address of *array1[t0]* and *array1[t1]*.
   I then load *array1[t0]* and *array1[t1]* to registers *t3* and *t4* respectively using *lw*, then I use *sw* to load the value of *t3* and *t4* to *array1[t1]* and *array1[t0]*, thus done swapping.
   The graph representation is shown below:



(2) *partition*

The function *partition* follows the partition pseudo code given in lab 1.2 power point. Throughout the partition process, it maintains to index parameters *i* and *j*, so that it always satisfies that (a) all the elements with index <= *i* are smaller than pivot (b) all the elements with index > *i* and <= *j* are bigger than pivot. Every iteration, *j* increases by 1, and the new element *array1[j]* is compared with pivot. If *array1[j]* > pivot, no further manipulation is needed to satisfy (a) and (b); else if *array1[j -1]* < pivot, we increase *i* by 1 and swap *array1[i]* and *array1[j]* to satisfy (a) and (b). Finally, when j reaches the last element before pivot, swap *array1[i + 1]* with pivot to make all the numbers smaller than pivot stored at the left of it, and all the numbers greater than pivot stored at the left of it.

The pseudo code is shown below:

```
function PARTITION(A, lo, hi)
    pivot ← A[hi]
    i ← lo-1;
    for j = lo; j ≤ hi-1; j ← j+1 do
        if A[j] ≤ pivot then
            i ← i+1;
            swap A[i] with A[j];
        end if
    end for
    swap A[i+1] with A[hi];
    return i+1;
end function
```

The function in assembly language is explained below in detail.

(a) It takes the address of array *A(array1)*, the index of the element before pivot (which is the last element) *hi*, and the index of the first element (which is the 0 in this situation) *lo* as parameters, read from register *a0, a1, a2* respectively. The value of pivot is stored into *a3*, and *i*, *j* are initialized and stored in *a4* and *a5* respectively.

(b) Then there is a for loop marked by label *start_for* as start and *end_for* as stop.

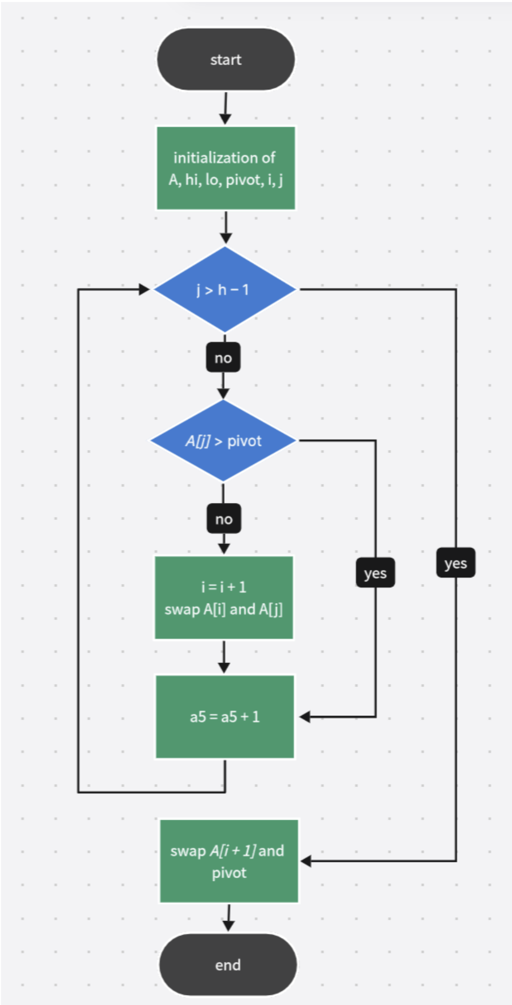
Inside the for loop, it first checks whether *a5* > *a2* – 1 (*j* > *hi - 1*) using *bgt*. If the condition is true, it will jump to *end_for* to end the for loop. Else the loop continues and goes into an if-block marked by label *end_if* as stop. It first checks whether *A[j] > a3* (pivot) using *bgt*. If the condition is true, the if-block is skipped by branching to *end_if*, else it will increase *i* by 1 and swap the elements with index *a4 (A[i])* and *a5 (A[j])* by calling the *swap* function.

After the if-block, *a5* increases by 1 and *pc* jumps to *start_for* again.

(c) After the for loop, it will swap elements with index *a4 + 1 (A[i + 1])* and *a2* (pivot)by calling the *swap* function.

(d) Finally it will return to the main function using *jr.*

The graph representation is shown below:



4. Using functions declared to perform partition

(1) Variable definition in *.data* section

First, to store the elements of *array1*, I put the numbers into consecutive memories by only defining a single *.word* type variable *array1* and put all 10 numbers after it. By this method, the first number will be in address *array1*, and the ith number will be in address *array1 + (i – 1) * 4*. I also created a variable named *len* to store the length of the array, which is 10, to make the program aware of where is the last element of the array.

Then, we can define a variable *original_pivot_index* to store the index of the pivot, which is 2.

Also, since we need to print out the array line by line, we have to print out the new line character. So I additionally defined the new line character *newline* of type *.asciz* and of value "\n".

(2) Swap pivot to the last element

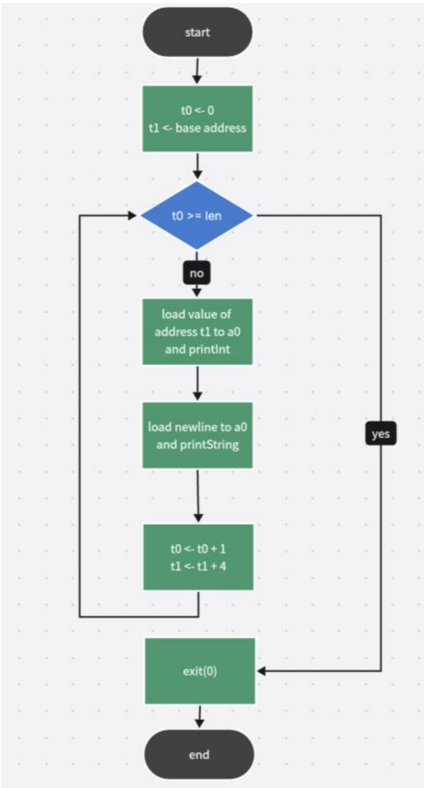
I first set *t0* as *original_pivot_index* and set *t1* as *len* – 1, then I call *swap* function to swap pivot to the last element.

(3) Perform partition where the last element is the pivot

(4) I first set *a0, a1, a2* to *array1*, 0 and *len* – 1 respectively, then I call *partition* to perform partition where the last element is the pivot.

(5) Output the partitioned array

Using a for loop similar to the one in *partition* function, we can print out the element of the array one by one, each followed with a newline character.



5. Main code

(1) Variable declaration

```
.data
array1: .word -1, 22, 8, 35, 5, 4, 11, 2, 1, 78
len: .word 10
original_pivot_index: .word 2
newline: .string "\n"
```

(2) Calling swap and partition function

```
# swap pivot to the last element
lw t0, original_pivot_index        # a0 set to original_pivot_index

lw t1, len
addi t1, t1, -1                    # a1 set to last element index

jal ra, swap                       # swap

# perform partition
la a0, array1                      # a0: address of the array
li a1, 0                           # a1: lo = 0
lw a2, len
addi a2, a2, -1                    # a2: hi = len - 1

jal a6, partition
```

(3) Output the partitioned array using for loop and exit

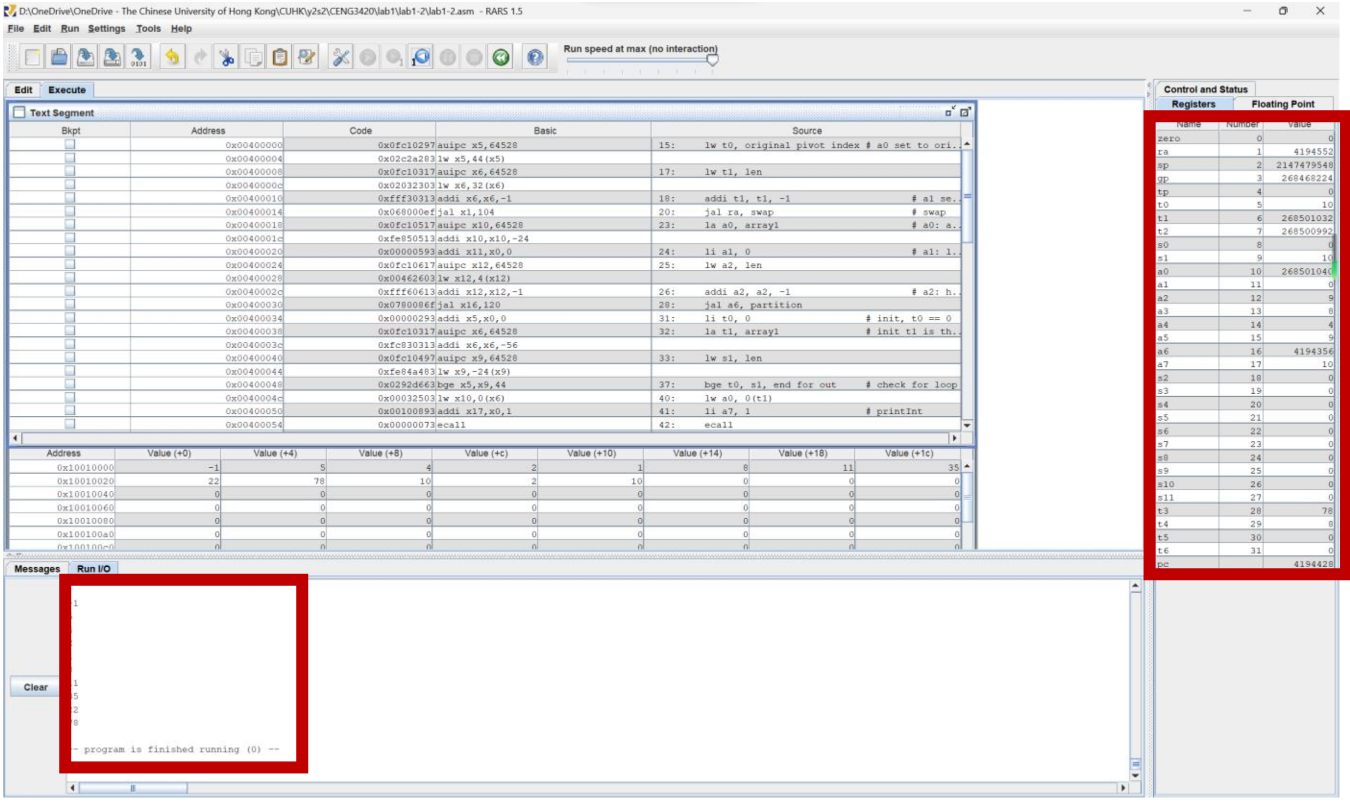
```
    # print
    li t0, 0                    # init, t0 == 0
    la t1, array1               # init t1 is the address of current int to print
    lw s1, len

    ## for_loop to output
start_for_out:
    bge t0, s1, end_for_out     # check for loop

    ## load the int to print to a0 and output
    lw a0, 0(t1)
    li a7, 1                    # printInt
    ecall

    ## ouput a newline
    la a0, newline
    li a7, 4                    # printString
    ecall

    ## update
    addi t0, t0, 1              # t0 ++
    addi t1, t1, 4              # t1 = t1 + 4

    ## jump
    j start_for_out
end_for_out:

    # exit
    li a7, 10
    ecall
```

6. Console results

(1) Input

```
array1: .word -1, 22, 8, 35, 5, 4, 11, 2, 1, 78
len: .word 10
original_pivot_index: .word 2
```

(2) Output




```
-1
5
4
2
1
8
11
35
22
78

-- program is finished running (0) --
```

Lab 1.3

1. Problem analysis

   In lab 1.3, we are given an array from console input. We need to sort it into ascending order and print it out line by line.

   The task mainly consists of 3 parts: input, sort and output. The input and output part could be easily done with a for loop similar to the one implemented in lab 1.2. The sorting part can be done with quicksort, and the sorting is a recursion process of first partition the array based on the pivot and recursively sort the 2 subarrays, and the partition part is also similar to the partition function implemented in lab 1.2.

2. Recursive implementation of *qsort* function.

   (1) Algorithm overview

   Quick sort algorithm follows the divide and conquer principle. Given any array, we can pick a pivot and partition the array into 2 subarrays where the lower array has all the elements that are smaller than pivot and the higher array has all the elements that are bigger than pivot. Then we can do the same quick sort to the 2 subarrays just as what is done to the original array to generate more partitioned subarrays until each subarray has only 1 element. Since all the subarrays are the result of partition, the elements in different subarrays must follow the ascending order. So, when all the subarrays have only one element, the sorting of the whole array is finished.

   C code representation of *qsort*:

```c
void qsort(int lo, int hi)
{
    // check if end recursion
    if (lo >= hi)
    {
        return;
    }

    int pivot = array[hi];
    int i = lo - 1;

    // partition
    for (int j = lo; j < hi; j++)
    {
        if (array[j] <= pivot)
        {
            i++;
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }

    int temp = array[i + 1];
    array[i + 1] = array[hi];
    array[hi] = temp;

    // qsort sub-arrays
    qsort(array, lo, i);
    qsort(array, i + 2, hi);
}
```

(2) Detailed implementation in assembly language

(a) Variables used in *qsort*

Throughout *qsort* process, register *a0* always stores the base address of the array to sort. *a1* is the index of the smallest address of the array to partition (*lo* in C code), *a2* is the index of the smallest address of the array to partition (*hi* in C code). *a1* and *a2* are parameters defined by caller. In *qsort*, I always see the last element in the array to partition as pivot, and store its value in *a3*. Similar to the partition algorithm in lab 1.2 *a4*, *a5* stores the index *i*, *j* that marks the end of the partitioned elements smaller and bigger than pivot respectively. *ra* stores the return address of the callee. Besides, some temporary registers are also used in the function but are just temporary such that modifying their values by callee won't have an effect on the caller.

(b) Push the used parameters into stack

Since *qsort* is a recursive function and uses a lot of variables, we need to push the useful values in the registers that *qsort* might use into stack before running the *qsort* function, in order to maintain these values unchanged after the function finishes.

Throughout the code, only 6 registers *ra*, *a1* to *a5* are modified and might contain important values that might be used by caller, so we only need to push these 6 values into stack. This is done by subtracting *sp* by 24 and storing these values one by one to the corresponding address.

(c) Read *a1* and *a2*

a1 and *a2* are variables that are read from caller. Since *qsort* is a recursive function, the caller shouldn't directly store the *a1* and *a2* of the callee directly into *a1* and *a2* because *a1* and *a2* of the caller also have meaning, and modifying them might cause an error in the subsequent instructions. One of the ways to solve this problem is to first store the *a1* and *a2* of the callee in temporary register *t0* and *t1*, and let callee load them to *a1* and *a2* after the callee has already stored the *a1* and *a2* of the caller into stack. By this method, *a1* and *a2* won't change in caller's view since the value is pushed into stack by callee and popped back when callee finishes its program, and thus won't cause an error.

Therefore, after pushing the used parameters into stack, I copy the value of *t0* and *t1* into *a1* and *a2.*

(d) Check if end recursion

If *a2* (*hi*)<= *a1* (*lo*), it means that there are at most 1 element in the subarray, thus the partition is finished. So we don't need to do partition and recursive quicksort, we can directly branch to the step to pop the registers out of stack.

(e) Partition the array between *array[lo]* and *array[hi]*

Since we always set the pivot to be the last element of the array, the partition process is exactly the same with the *partition* process done in lab 1.2.

(f) Perform *qsort* on 2 subarrays

After partition, all the elements who are smaller than pivot is at the left of pivot and all the elements who is bigger than pivot is at the right of pivot. So we only need to recursively quick sort the 2 subarrays to finish *qsort*.

From the algorithm above, we know that at the end of the partition, the index of *i* is stored in *a4*, so the index of pivot is *a4* + 1. Therefore, the *lo* of smaller array is *lo* of original array, the *hi* of smaller array is *a4* + 1 – 1 = *a4,* the *lo* of bigger array is *a4* + 1 + 1 = *a4* + 2, the *hi* of bigger array is *hi* of original array.

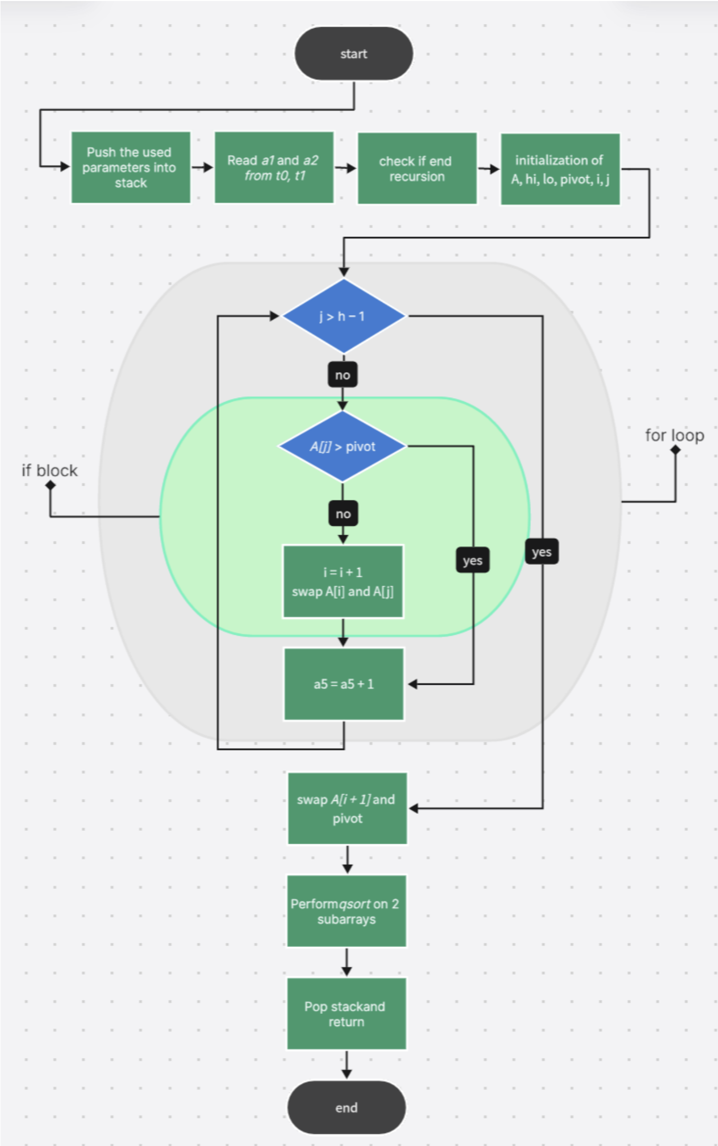
By setting these values to *t0* and *t1*, *qsort* on 2 subarrays can be performed.

(g) Pop stack and return

After all the instructions of *qsort* are performed, we should pop the values of the registers stored in stack out. This can be done by first use lw to load back all the values of registers in (b) and then adding *sp* by 24.

Finally, we can jump to the address stored by *ra* to hand the control to caller.

The graph of *qsort* algorithm is shown below:



3. Main function

(1) Variable definition in *.data* section

Since we need to print out the array line by line, we have to print out the new line character. So I additionally defined the new line character *newline* of type *.asciz (.string)* and of value "\n".

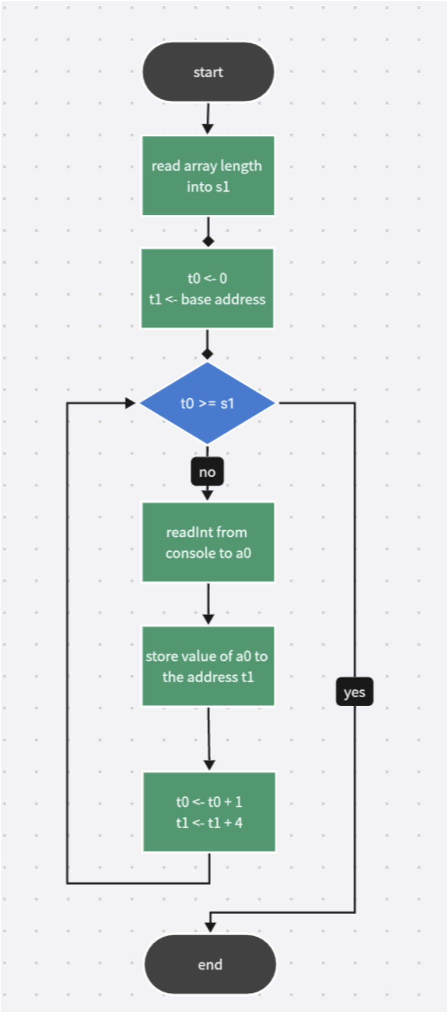
Similar to lab 1.2, we can just declare a single *.word* variable *array* as the first element of the array, then the address of the rest of the elements can be calculated based on the base address and the index of the element.

(2) Input the array from console

I use *s1* to store the length of the array, which is the first input of the console. Similar to lab 1.1, this can be done by loading service number 5 (*readInt*) into register *a7* by *li* and executing *ecall*.

Then with the number of elements that will be inputted from the console known, we can use a for loop to input the array elements also using *readInt*.

The graph representation is shown below:



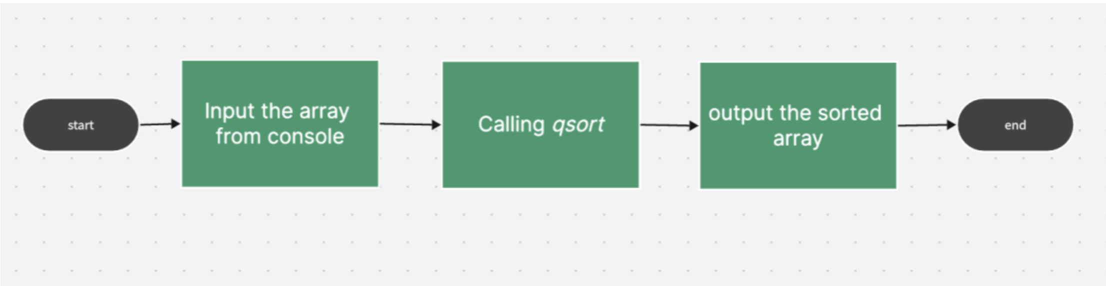
(3) Calling *qsort*

First initialize *a0* to the base address of the array, and initialize the required input parameters *t0 (lo)*, *t1 (hi)* to 0 and *s1* – 1 respectively. Then call the function *qsort* by *jal* command.

(4) Output the sorted array

This step should be the same with lab 1.2: output the partitioned array.

The overall graph representation is shown below:



4. Main code:

(1) *qsort* function

```
# qsort(hi, lo)
qsort:
    # description: given lo, hi INDEX of an array, sort it in ascending order (assume last element is pivot)
    # input
    ## t0: lo idx
    ## t1: hi idx
    # var_used:
    ## a0: address of array
    ## a1: (from input param t0) lo -- index
    ## a2: (from input param t1) hi -- index
    ## a3: pivot
    ## a4: i -- index
    ## a5: j -- index
    ## temporary registers

    # push
    addi sp, sp, -24
    sw ra, 20(sp)
    sw a1, 16(sp)
    sw a2, 12(sp)
    sw a3, 8(sp)
    sw a4, 4(sp)
    sw a5, 0(sp)
```

```
    # read input param
    mv a1, t0
    mv a2, t1

    # check if end recursion
    ble a2, a1, return_qsort

    # partition
    ## init i
    addi a4, a1, -1

    ## init j
    addi a5, a1, 0

    ## load pivot
    slli t0, a2, 2
    add t0, t0, a0
    lw a3, 0(t0)


    ## for loop
start_for_par:

    ## for loop check
    addi t3, a2, -1
    bgt a5, t3, end_for_par    # j > hi - 1

    ## if check
    slli t0, a5, 2
    add t0, t0, a0
    lw t3, 0(t0)               # t3 = A[j]
    bgt t3, a3, end_if_par             # A[j] > pivot

    ## inside if
    addi a4, a4, 1             # i = i + 1

    mv t0, a4                  # call swap function
    mv t1, a5
    jal ra, swap

end_if_par:
    ## end if
    addi a5, a5, 1
    j start_for_par
```

```
end_for_par:
    ## swap
    addi t0, a4, 1
    mv t1, a2
    jal ra, swap


    # qsort 2 subarrays
    addi t2, a4, 1             # t2 <- pivot_idx = i + 1
    addi t3, t2, -1            # t3 <- the array that is smaller than pivot's hi = t0 - 1
    addi t4, t2, 1             # t4 <- the array that is bigger than pivot's hi = t0 + 1

    ## sort lower array
    mv t0, a1                  # lo_lo = lo
    mv t1, t3                  # lo_hi = pivot_idx - 1
    jal ra, qsort

    ## sort higher array
    mv t0, t4                  # hi_lo = pivot_idx + 1
    mv t1, a2                  # lo_hi = hi
    jal ra, qsort

return_qsort:
    # pop stack
    lw a5, 0(sp)
    lw a4, 4(sp)
    lw a3, 8(sp)
    lw a2, 12(sp)
    lw a1, 16(sp)
    lw ra, 20(sp)
    addi sp, sp, 24

    # return
    jr ra
```

(2) variable declaration

```
.data
space: .string " "
array: .word 0
```

(3) input array

```
    # input
    ## input lenth and store in len
    li a7, 5
    ecall
    mv s1, a0

    ## input the array
    ### t0: for_loop counter
    ### t1: the address of current spot for the array
    la t1, array

start_for:
    bge t0, s1, end_for        # check for loop, initially t0 == 0

    ### input, a7 == 5 already
    ecall

    ### store
    sw a0, 0(t1)

    ### update
    addi t0, t0, 1             # t0 ++
    addi t1, t1, 4             # t1 = t1 + 4

    ### jump
    j start_for
end_for:
```

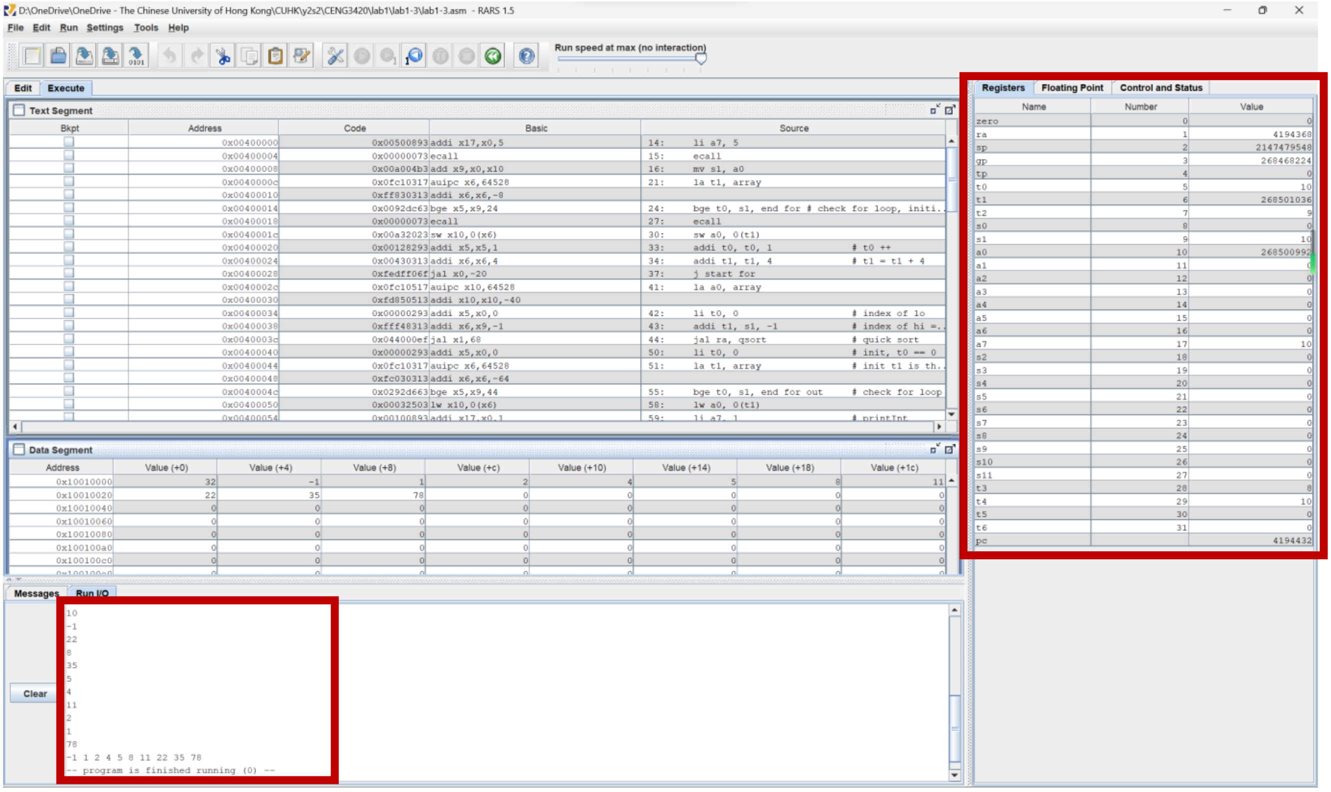
(4) calling *qsort* function

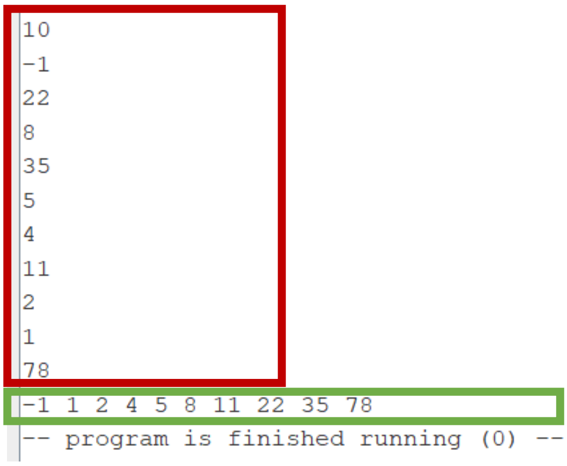
```
# qsort
la a0, array
li t0, 0                    # index of lo
addi t1, s1, -1             # index of hi = s1 - 1
jal ra, qsort               # quick sort
```

(5) output array and exit

```
    # output
    ## vars:
    ### t0: for_loop counter
    ### t1: the address of current spot for the array
    li t0, 0                    # init, t0 == 0
    la t1, array                # init t1 is the address of current int to print

    ## for_loop to output
start_for_out:
    bge t0, s1, end_for_out     # check for loop

    ## load the int to print to a0 and output
    lw a0, 0(t1)
    li a7, 1                    # printInt
    ecall

    ## ouput a space
    la a0, space
    li a7, 4                    # printString
    ecall

    ## update
    addi t0, t0, 1              # t0 ++
    addi t1, t1, 4              # t1 = t1 + 4

    ## jump
    j start_for_out
end_for_out:

    # exit
    li a7, 10
    ecall
```

5. Console results





Reference:

TextBook -Computer Organization and Design_ The Hardware Software Interface [RISC-V Edition]