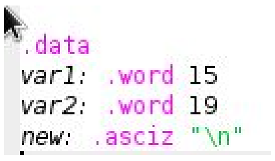
CENG3420 lab 1 Report

Wong Ka Lap 1155125399

Lab 1-1:

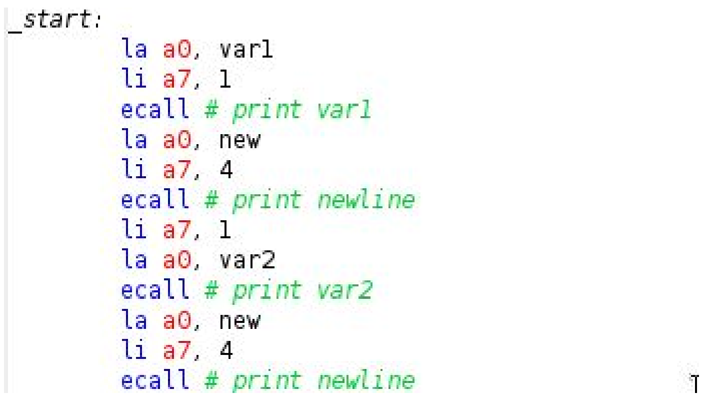
1. First we define two variables var1 and var2 in .data.



```
.data
var1: .word 15
var2: .word 19
new:  .asciz "\n"
```

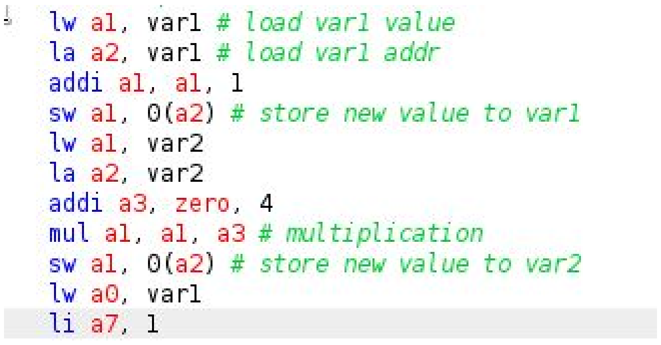
   A new line character is also defined for printing new line using .asciz type.

2. Use la to get to fetch var1 and var2 addresses. Use li to load corresponding syscall values to a7. 1 is for printing integers and 4 is for string of that address.



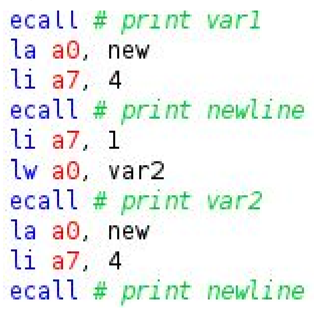
```
_start:
        la a0, var1
        li a7, 1
        ecall # print var1
        la a0, new
        li a7, 4
        ecall # print newline
        li a7, 1
        la a0, var2
        ecall # print var2
        la a0, new
        li a7, 4
        ecall # print newline
```

3. Use lw to fetch var1 and var2 addresses, then use la to load their values from addresses. For var1, use addi to do addition. For var2 use mul to perform multiplication. After calculation, use sw to store back their values to corresponding addresses.



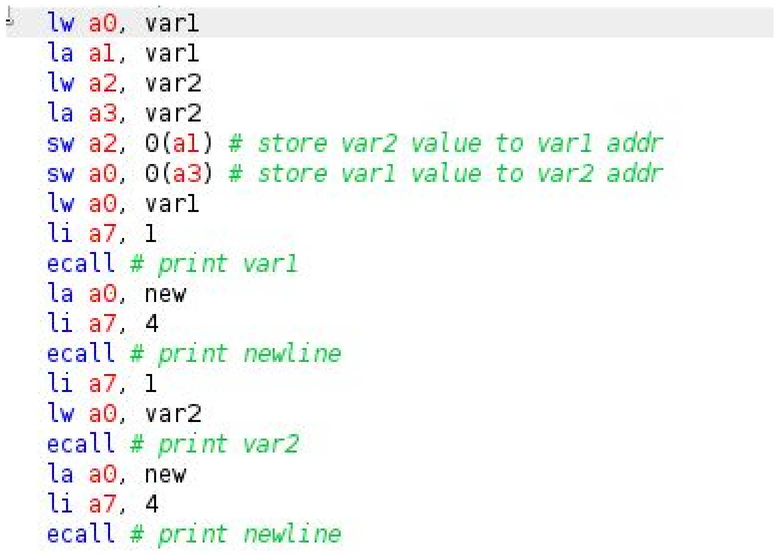
```
    lw a1, var1 # load var1 value
    la a2, var1 # load var1 addr
    addi a1, a1, 1
    sw a1, 0(a2) # store new value to var1
    lw a1, var2
    la a2, var2
    addi a3, zero, 4
    mul a1, a1, a3 # multiplication
    sw a1, 0(a2) # store new value to var2
    lw a0, var1
    li a7, 1
```

4. Again load the corresponding syscall signal to a7 then call ecall.



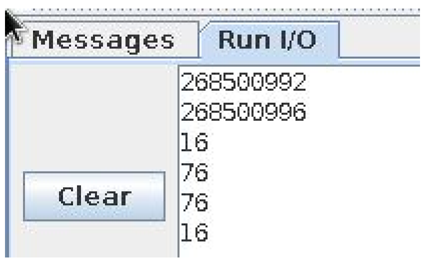
```
ecall # print var1
la a0, new
li a7, 4
ecall # print newline
li a7, 1
lw a0, var2
ecall # print var2
la a0, new
li a7, 4
ecall # print newline
```

5. Fetch both the addresses and values of var1 and var2 to registers. Then save the values of var1 to the address of var2, vice versa. Then use the same method in step 2 and step 4 to print output.



```
lw a0, var1
la a1, var1
lw a2, var2
la a3, var2
sw a2, 0(a1) # store var2 value to var1 addr
sw a0, 0(a3) # store var1 value to var2 addr
lw a0, var1
li a7, 1
ecall # print var1
la a0, new
li a7, 4
ecall # print newline
li a7, 1
lw a0, var2
ecall # print var2
la a0, new
li a7, 4
ecall # print newline
```

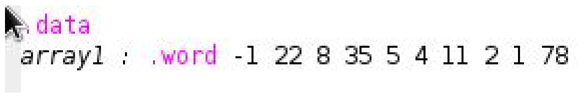
Output:



```
268500992
268500996
16
76
76
16
```

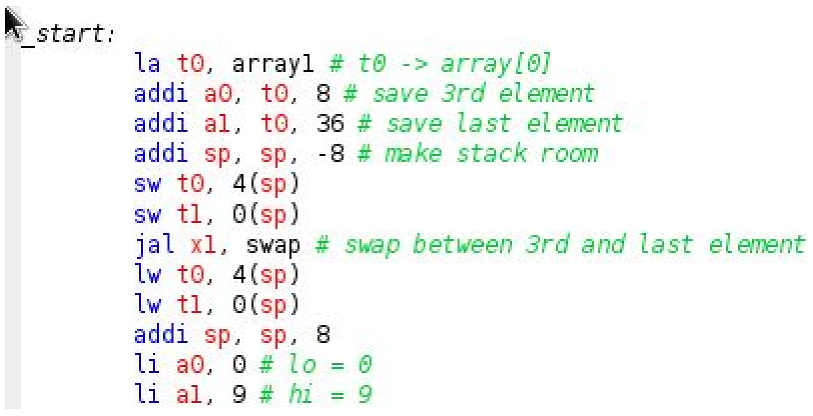
Lab 1-2:

1. We store the array in .data.



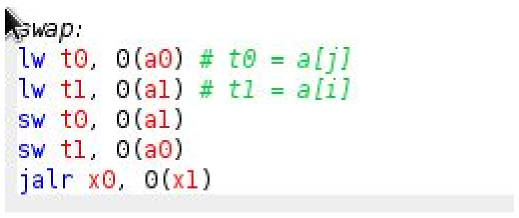
```
.data
array1 : .word -1 22 8 35 5 4 11 2 1 78
```

   Before doing the partitioning, we need to swap the 3rd element with the last element. The reason is I use the last element as the pivot. Also we set t0 as the base address of A.



```
_start:
        la t0, array1 # t0 -> array[0]
        addi a0, t0, 8 # save 3rd element
        addi a1, t0, 36 # save last element
        addi sp, sp, -8 # make stack room
        sw t0, 4(sp)
        sw t1, 0(sp)
        jal x1, swap # swap between 3rd and last element
        lw t0, 4(sp)
        lw t1, 0(sp)
        addi sp, sp, 8
        li a0, 0 # lo = 0
        li a1, 9 # hi = 9
```

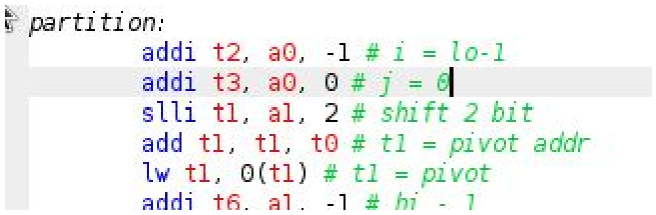
   Before swapping, we save t0 and t1 to stack to preserve their values.



```
swap:
lw t0, 0(a0) # t0 = a[j]
lw t1, 0(a1) # t1 = a[i]
sw t0, 0(a1)
sw t1, 0(a0)
jalr x0, 0(x1)
```

   In the swapping function, a1 and a0 will be swapped.
   After swapping, a0 and a1 will be used as lo and hi respectively.

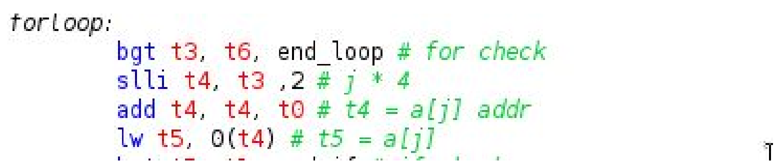
2. Then we will set t2 as i, t3 as j, t1 as the pivot.



```
partition:
        addi t2, a0, -1 # i = lo-1
        addi t3, a0, 0 # j = 0
        slli t1, a1, 2 # shift 2 bit
        add t1, t1, t0 # t1 = pivot addr
        lw t1, 0(t1) # t1 = pivot
        addi t6, a1, -1 # hi - 1
```

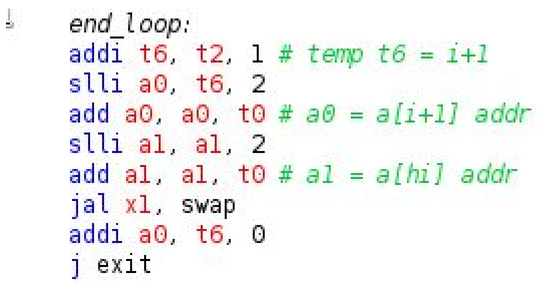
   Because we are fetching A[j], we shift the value of j (the same as j*4) and add the value of A (t0). The t6 register is for hi-1 which is used in the for loop check.

3. We use bgt for branching to check if j <= hi-1. It will branch to end_loop if the loop is finished. For j++, it is placed in endif because it is the last instruction in the loop.



```
forloop:
        bgt t3, t6, end_loop # for check
        slli t4, t3 ,2 # j * 4
        add t4, t4, t0 # t4 = a[j] addr
        lw t5, 0(t4) # t5 = a[j]
```

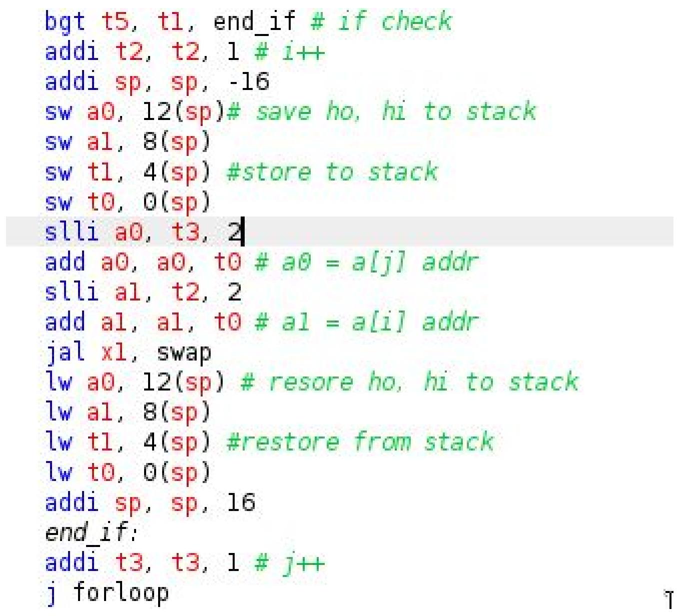
End_loop:



```
end_loop:
addi t6, t2, 1 # temp t6 = i+1
slli a0, t6, 2
add a0, a0, t0 # a0 = a[i+1] addr
slli a1, a1, 2
add a1, a1, t0 # a1 = a[hi] addr
jal x1, swap
addi a0, t6, 0
j exit
```

In the end_loop, we again swap the values of a[i+1] and a[hi] and return i+1 to a0 as return value.

4. Then in the if part. It will branch to endif when satisfied.

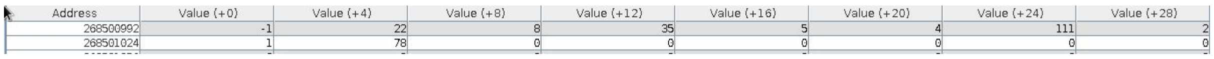


```
bgt t5, t1, end_if # if check
addi t2, t2, 1 # i++
addi sp, sp, -16
sw a0, 12(sp)# save ho, hi to stack
sw a1, 8(sp)
sw t1, 4(sp) #store to stack
sw t0, 0(sp)
slli a0, t3, 2
add a0, a0, t0 # a0 = a[j] addr
slli a1, t2, 2
add a1, a1, t0 # a1 = a[i] addr
jal x1, swap
lw a0, 12(sp) # resore ho, hi to stack
lw a1, 8(sp)
lw t1, 4(sp) #restore from stack
lw t0, 0(sp)
addi sp, sp, 16
end_if:
addi t3, t3, 1 # j++
j forloop
```

Inside the if statement, we need to save a0, a1, t0, t1 in the stack in order to perform swapping. Then load the value of a[j] to a0 and a[i] to a1 and jal to swap. After swapping perform lw to restore the registers values.

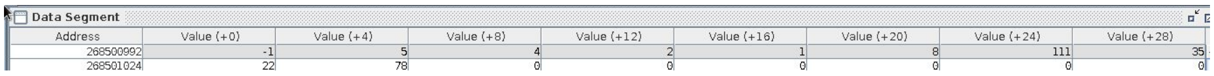
5. Sample run

Values before partitioning:



Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	-1	22	8	35	5	4	111	2
268501024	1	78	0	0	0	0	0	0

Values after partitioning:

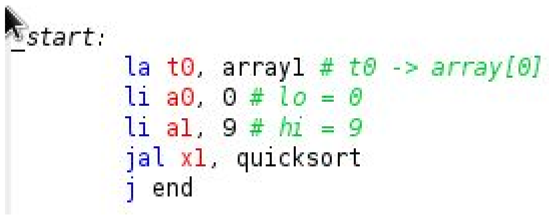


Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	-1	5	4	2	1	8	111	35
268501024	22	78	0	0	0	0	0	0

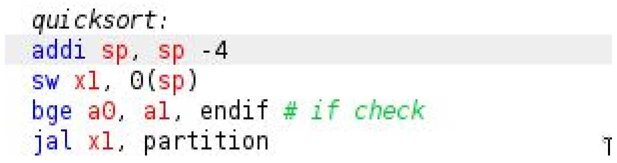
Lab 1-3

1. We set t0 = A, a0 = lo, a1 = hi. Then jal to quicksort algorithm.



```
start:
        la t0, array1 # t0 -> array[0]
        li a0, 0 # lo = 0
        li a1, 9 # hi = 9
        jal x1, quicksort
        j end
```

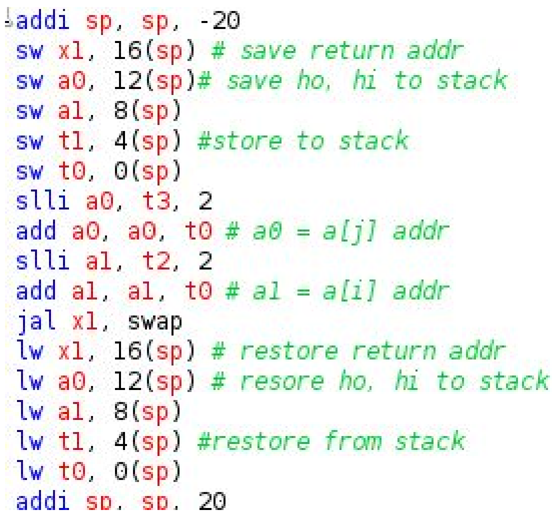
2. First we save the return address to stack. When the whole quicksort is done we need this address to return to “j end”.



```
quicksort:
addi sp, sp -4
sw x1, 0(sp)
bge a0, a1, endif # if check
jal x1, partition
```

If lo < hi jump to partition which is similar to lab 1-2,

3. In the partition function, the difference between this version and lab 1-2 is that we store x1 (ra, return address) in every swap call.

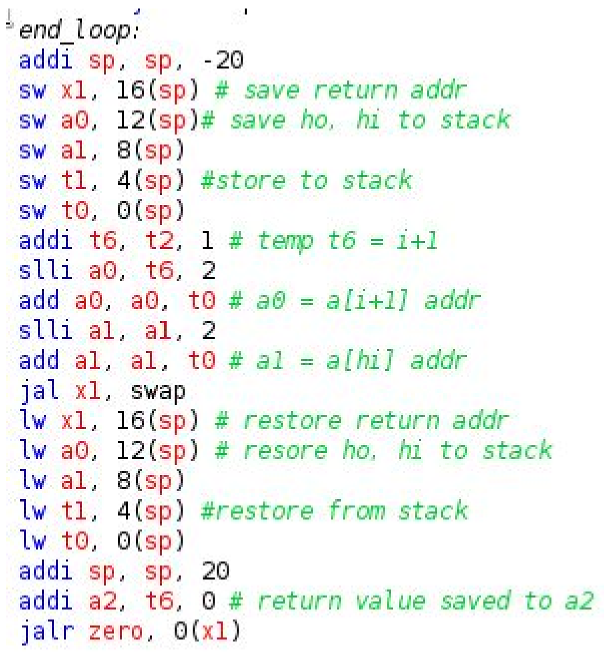


```
addi sp, sp, -20
sw x1, 16(sp) # save return addr
sw a0, 12(sp)# save ho, hi to stack
sw a1, 8(sp)
sw t1, 4(sp) #store to stack
sw t0, 0(sp)
slli a0, t3, 2
add a0, a0, t0 # a0 = a[j] addr
slli a1, t2, 2
add a1, a1, t0 # a1 = a[i] addr
jal x1, swap
lw x1, 16(sp) # restore return addr
lw a0, 12(sp) # resore ho, hi to stack
lw a1, 8(sp)
lw t1, 4(sp) #restore from stack
lw t0, 0(sp)
addi sp, sp, 20
```

(stack frame of swapping)

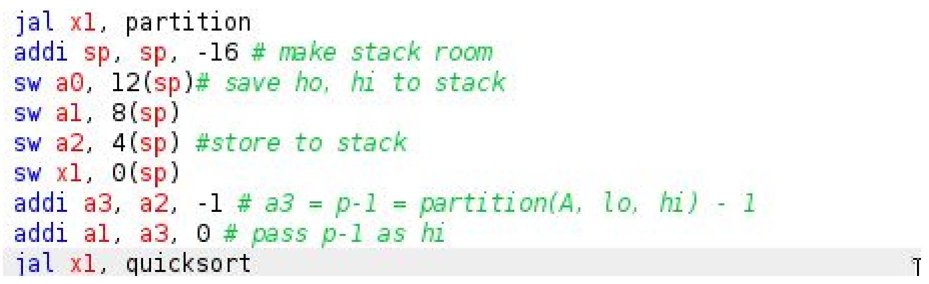
The reason is that in quicksort we need to call functions recursively, the program needs to remember where to return every time it jal or jalr.

Also after the end of partition, we use jalr to return to where it was called instead of jump to the end.



```
end_loop:
addi sp, sp, -20
sw x1, 16(sp) # save return addr
sw a0, 12(sp)# save ho, hi to stack
sw a1, 8(sp)
sw t1, 4(sp) #store to stack
sw t0, 0(sp)
addi t6, t2, 1 # temp t6 = i+1
slli a0, t6, 2
add a0, a0, t0 # a0 = a[i+1] addr
slli a1, a1, 2
add a1, a1, t0 # a1 = a[hi] addr
jal x1, swap
lw x1, 16(sp) # restore return addr
lw a0, 12(sp) # resore ho, hi to stack
lw a1, 8(sp)
lw t1, 4(sp) #restore from stack
lw t0, 0(sp)
addi sp, sp, 20
addi a2, t6, 0 # return value saved to a2
jalr zero, 0(x1)
```

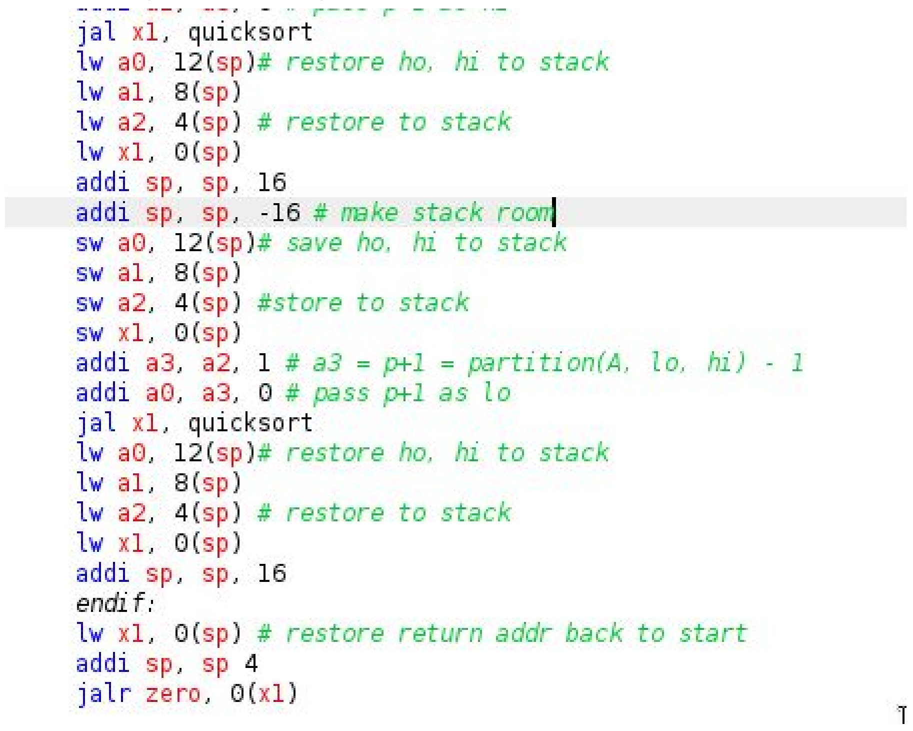
4. After partition is finished, it will return to quicksort and proceed.



```
jal x1, partition
addi sp, sp, -16 # make stack room
sw a0, 12(sp)# save ho, hi to stack
sw a1, 8(sp)
sw a2, 4(sp) #store to stack
sw x1, 0(sp)
addi a3, a2, -1 # a3 = p-1 = partition(A, lo, hi) - 1
addi a1, a3, 0 # pass p-1 as hi
jal x1, quicksort
```

The value returned by partition is placed in a2. In order to call quicksort(A, lo, p - 1), we need to make a stack frame to store the current variables (a0,a1,a2,x1) by expanding sp by -16 and sw the corresponding register to the stack. Then we passed the new parameters to a0, a1 and called quicksort(A, lo, p - 1).

5. It will recursively partition the first half of the array until it cannot be partitioned (The smallest partition case). Then it is time for quicksort(A, p + 1, hi).



```
jal x1, quicksort
lw a0, 12(sp)# restore ho, hi to stack
lw a1, 8(sp)
lw a2, 4(sp) # restore to stack
lw x1, 0(sp)
addi sp, sp, 16
addi sp, sp, -16 # make stack room
sw a0, 12(sp)# save ho, hi to stack
sw a1, 8(sp)
sw a2, 4(sp) #store to stack
sw x1, 0(sp)
addi a3, a2, 1 # a3 = p+1 = partition(A, lo, hi) - 1
addi a0, a3, 0 # pass p+1 as lo
jal x1, quicksort
lw a0, 12(sp)# restore ho, hi to stack
lw a1, 8(sp)
lw a2, 4(sp) # restore to stack
lw x1, 0(sp)
addi sp, sp, 16
endif:
lw x1, 0(sp) # restore return addr back to start
addi sp, sp 4
jalr zero, 0(x1)
```

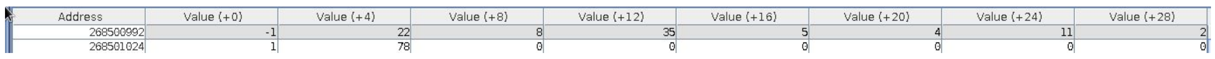
After quicksort(A, lo, p - 1) is returned, we use lw to restore the stack frame, the registers' values are restored. Then we use pass p+1 to a3. Then copy a3 to a0 to change the parameter to p+1. When quicksort is called, quicksort(A, p + 1, hi) is called. Before it is called, we put (a0,a1,a2,x1) to stack and restore them back after the quicksort is returned.

6. When all sub-routine is returned, the first quicksort called which is quicksort(A, lo, hi) is returned. The "j end" jumps to the end of the program and terminates the program.

```
end:
```

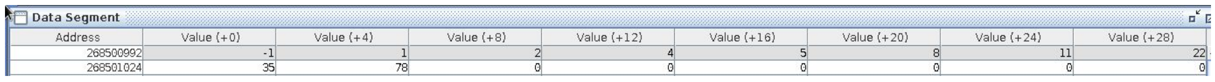
Output:

Initial array: (unsorted)



Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	-1	22	8	35	5	4	11	2
268501024	1	78	0	0	0	0	0	0

Final result: (sorted)



Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	-1	1	2	4	5	8	11	22
268501024	35	78	0	0	0	0	0	0