**Module: R2: Intro to RISC-V Assembly**

**Section:** CALL **Task:** RISC-V Function & Pointers

**LAB 4 -** [**https://github.com/ImNomanCR7/fa21-lab-starter.git**](https://github.com/ImNomanCR7/fa21-lab-starter.git)

**RISC-V Function & Pointers**

**Exercise 1: Function without Branches:**

* + **Code Snippet:**

f:

# YOUR CODE GOES HERE!

addi a0, a0, 3 # add 3 to a0 to get the index

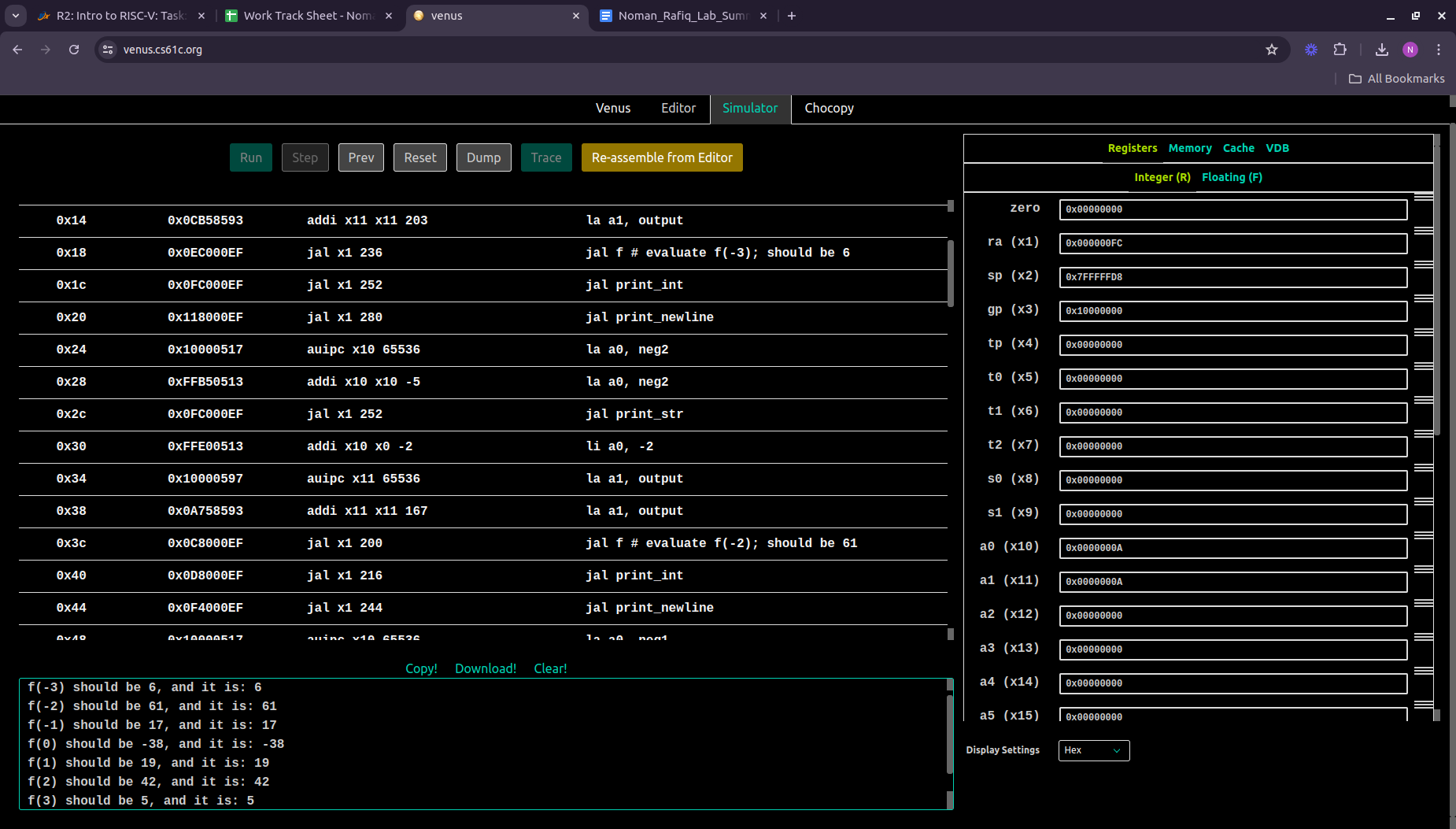
slli a0, a0, 2 # shift by 2 bits to calculate offset

add a1, a1, a0 # a1 = address of output[i]

lw a0, 0(a1) # a0 = a[i]

jr ra # Always remember to jr ra after your function!

**Terminal Output:**



**Exercise 2: Calling Convention Checker**

* + **Code Snippet:**

# Author: Noman Rafiq

# Date: July 2, 2024

# Fixed CC Violations

.globl simple\_fn naive\_pow inc\_arr

.data

failure\_message: .asciiz "Test failed for some reason.\n"

success\_message: .asciiz "Sanity checks passed! Make sure there are no CC violations.\n"

array:

.word 1 2 3 4 5

exp\_inc\_array\_result:

.word 2 3 4 5 6

.text

main:

# We test our program by loading a bunch of random values

# into a few saved registers - if any of these are modified

# after these functions return, then we know calling

# convention was broken by one of these functions

li s0, 2623

li s1, 2910

# ... skipping middle registers so the file isn't too long

# If we wanted to be rigorous, we would add checks for

# s2-s10 as well

li s11, 134

# Now, we call some functions

# simple\_fn: should return 1

jal simple\_fn # Shorthand for "jal ra, simple\_fn"

li t0, 1

bne a0, t0, failure

# naive\_pow: should return 2 \*\* 7 = 128

li a0, 2

li a1, 7

jal naive\_pow

li t0, 128

bne a0, t0, failure

# inc\_arr: increments "array" in place

la a0, array

li a1, 5

jal inc\_arr

jal check\_arr # Verifies inc\_arr and jumps to "failure" on failure

# Check the values in the saved registers for sanity

li t0, 2623

li t1, 2910

li t2, 134

bne s0, t0, failure

bne s1, t1, failure

bne s11, t2, failure

# If none of those branches were hit, print a message and exit normally

li a0, 4

la a1, success\_message

ecall

li a0, 10

ecall

# Just a simple function. Returns 1.

#

# FIXME Fix the reported error in this function (you can delete lines

# if necessary, as long as the function still returns 1 in a0).

simple\_fn:

li a0, 1

ret

# Computes a0 to the power of a1.

# This is analogous to the following C pseudocode:

#

# uint32\_t naive\_pow(uint32\_t a0, uint32\_t a1) {

# uint32\_t s0 = 1;

# while (a1 != 0) {

# s0 \*= a0;

# a1 -= 1;

# }

# return s0;

# }

#

# FIXME There's a CC error with this function!

# The big all-caps comments should give you a hint about what's

# missing. Another hint: what does the "s" in "s0" stand for?

naive\_pow:

# BEGIN PROLOGUE

addi sp, sp, -4

sw s0, 0(sp)

# END PROLOGUE

li s0, 1

naive\_pow\_loop:

beq a1, zero, naive\_pow\_end

mul s0, s0, a0

addi a1, a1, -1

j naive\_pow\_loop

naive\_pow\_end:

mv a0, s0

# BEGIN EPILOGUE

lw s0, 0(sp)

addi sp, sp, 4

# END EPILOGUE

ret

# Increments the elements of an array in-place.

# a0 holds the address of the start of the array, and a1 holds

# the number of elements it contains.

#

# This function calls the "helper\_fn" function, which takes in an

# address as argument and increments the 32-bit value stored there.

inc\_arr:

# BEGIN PROLOGUE

# FIXME What other registers need to be saved? s0 & s1!

addi sp, sp, -12

sw ra, 0(sp)

sw s0, 4(sp)

sw s1, 8(sp)

# END PROLOGUE

mv s0, a0 # Copy start of array to saved register

mv s1, a1 # Copy length of array to saved register

li t0, 0 # Initialize counter to 0

inc\_arr\_loop:

beq t0, s1, inc\_arr\_end

slli t1, t0, 2 # Convert array index to byte offset

add a0, s0, t1 # Add offset to start of array

# Prepare to call helper\_fn

addi sp, sp, -4

sw t0, 0(sp)

# FIXME Add code to preserve the value in t0 before we call helper\_fn

# Hint: What does the "t" in "t0" stand for?

# Also ask yourself this: why don't we need to preserve t1?

#

jal helper\_fn

# Finished call for helper\_fn

lw t0, 0(sp)

addi sp, sp, 4

addi t0, t0, 1 # Increment counter

j inc\_arr\_loop

inc\_arr\_end:

# BEGIN EPILOGUE

lw ra, 0(sp)

lw s0, 4(sp)

lw s1, 8(sp)

addi sp, sp, 12

# END EPILOGUE

ret

# This helper function adds 1 to the value at the memory address in a0.

# It doesn't return anything.

# C pseudocode for what it does: "\*a0 = \*a0 + 1"

#

# FIXME This function also violates calling convention, but it might not

# be reported by the Venus CC checker (try and figure out why).

# You should fix the bug anyway by filling in the prologue and epilogue

# as appropriate.

helper\_fn:

# BEGIN PROLOGUE

addi sp, sp, -4

sw s0, 0(sp)

# END PROLOGUE

lw t1, 0(a0)

addi s0, t1, 1

sw s0, 0(a0)

# BEGIN EPILOGUE

lw s0, 0(sp)

addi sp, sp, 4

# END EPILOGUE

ret

# YOU CAN IGNORE EVERYTHING BELOW THIS COMMENT

# Checks the result of inc\_arr, which should contain 2 3 4 5 6 after

# one call.

# You can safely ignore this function; it has no errors.

check\_arr:

la t0, exp\_inc\_array\_result

la t1, array

addi t2, t1, 20 # Last element is 5\*4 bytes off

check\_arr\_loop:

beq t1, t2, check\_arr\_end

lw t3, 0(t0)

lw t4, 0(t1)

bne t3, t4, failure

addi t0, t0, 4

addi t1, t1, 4

j check\_arr\_loop

check\_arr\_end:

ret

# This isn't really a function - it just prints a message, then

# terminates the program on failure. Think of it like an exception.

failure:

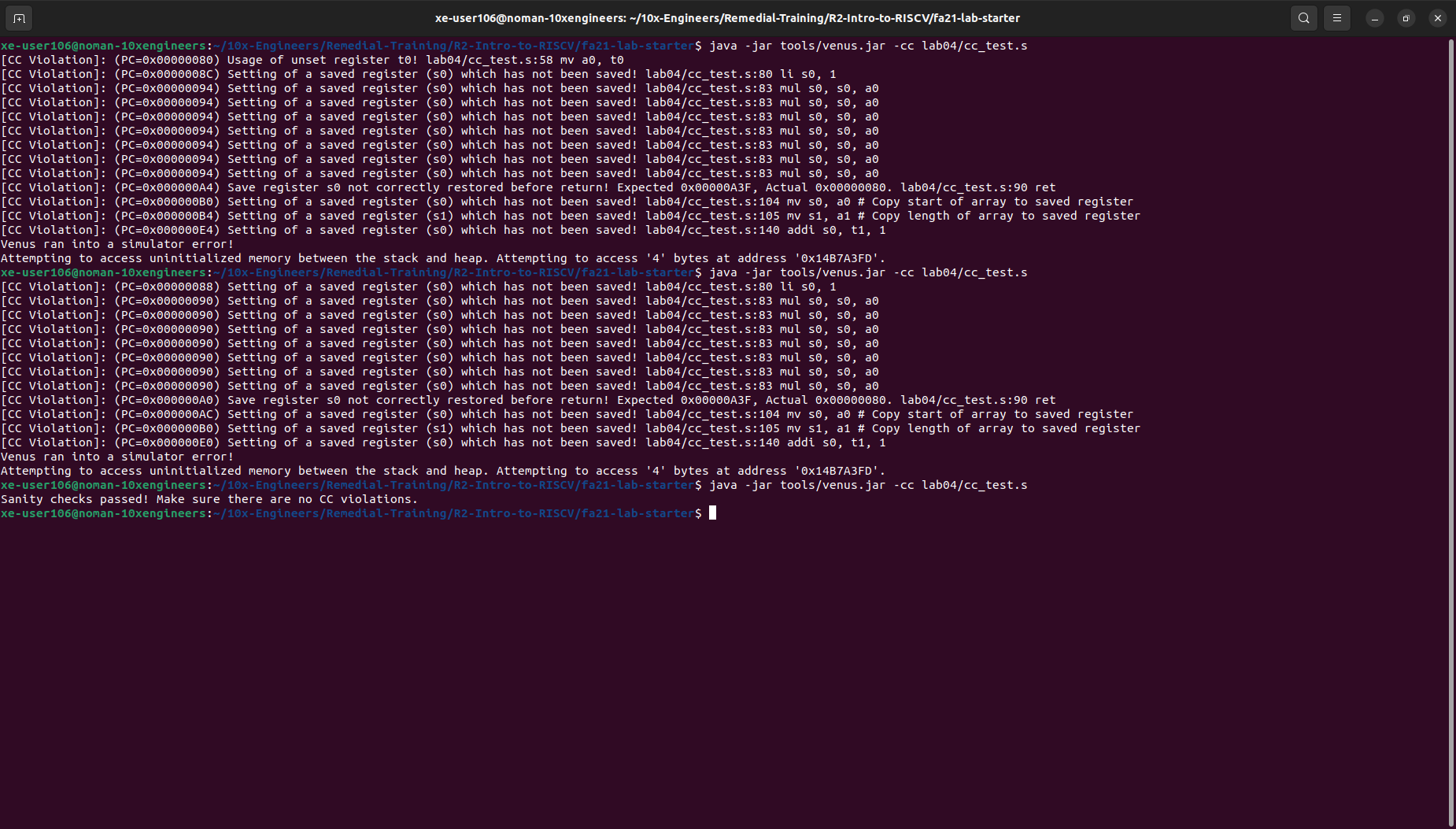
li a0, 4 # String print ecall

la a1, failure\_message

ecall

li a0, 10 # Exit ecall

ecall



* **What caused the errors in simple\_fn, naive\_pow, and inc\_arr that were‬ ‭reported by the Venus CC checker?**‬
  + **simple\_fn:** **t0** was used before it was initialized.‬
  + **naive\_pow:** Missing epilogue and prologues.‬
  + **inc\_arr:** Failure to save **s0**, **s1** in prologue/epilogue, and failure to save **t0** before‬ ‭calling **helper\_fn**.
* **In RISC-V, we call functions by jumping to them and storing the return‬ ‭address in the ra register. Does calling convention apply to the jumps to‬ ‭the naive\_pow\_loop or naive\_pow\_end labels?‬**
  + No, since they’re not functions, we don’t need to return to the location the function was‬ ‭called from.
* **Why do we need to store ra in the prologue for inc\_arr, but not in any‬ ‭other‬ Function?**
  + i‭nc\_arr itself calls another function‬ so it is a caller as well.
  + ‭Since **ra** holds the address of the instruction to continue executing after returning, which is‬ ‭overwritten when we call another function. Therefore we need to save the **ra** before it is overwritten by another call.
* **Why wasn't the calling convention error in helper\_fn reported by the CC‬ ‭checker?‬**
  + Because it is not declared **.globl.**

**Exercise 3: Debugging:**

* + **Code Snippet:**

.globl map

.data

arrays: .word 5, 6, 7, 8, 9

.word 1, 2, 3, 4, 7

.word 5, 2, 7, 4, 3

.word 1, 6, 3, 8, 4

.word 5, 2, 7, 8, 1

start\_msg: .asciiz "Lists before: \n"

end\_msg: .asciiz "Lists after: \n"

.text

main:

jal create\_default\_list

mv s0, a0 # v0 = s0 is head of node list

#print "lists before: "

la a1, start\_msg

li a0, 4

ecall

#print the list

add a0, s0, x0

jal print\_list

# print a newline

jal print\_newline

# issue the map call

add a0, s0, x0 # load the address of the first node into a0

la a1, mystery # load the address of the function into a1

jal map

# print "lists after: "

la a1, end\_msg

li a0, 4

ecall

# print the list

add a0, s0, x0

jal print\_list

li a0, 10

ecall

map:

addi sp, sp, -16

sw ra, 0(sp)

sw s1, 4(sp)

sw s0, 8(sp)

beq a0, x0, done # if we were given a null pointer, we're done.

add s0, a0, x0 # save address of this node in s0

add s1, a1, x0 # save address of function in s1

add t0, x0, x0 # t0 is a counter

# remember that each node is 12 bytes long:

# - 4 for the array pointer

# - 4 for the size of the array

# - 4 more for the pointer to the next node

# also keep in mind that we should not make ANY assumption on which registers

# are modified by the callees, even when we know the content inside the functions

# we call. this is to enforce the abstraction barrier of calling convention.

mapLoop:

lw t1, 0(s0) # load the address of the array of current node into t1

lw t2, 4(s0) # load the size of the node's array into t2

slli t3, t0, 2

add t3, t1, t3 # offset the array address by the count

lw a0, 0(t3) # load the value at that address into a0

jalr s1 # call the function on that value.

sw a0, 0(t3) # store the returned value back into the array

addi t0, t0, 1 # increment the count

bne t0, t2, mapLoop # repeat if we haven't reached the array size yet

lw a0, 8(s0) # load the address of the next node into a0

mv a1, s1 # put the address of the function back into a1 to prepare for the recursion

jal map # recurse

done:

lw s0, 8(sp)

lw s1, 4(sp)

lw ra, 0(sp)

addi sp, sp, 16

print\_newline:

li a1, '\n'

li a0, 11

ecall

jr ra

mystery:

mul t1, a0, a0

add a0, t1, a0

jr ra

create\_default\_list:

addi sp, sp, -24

sw ra, 0(sp)

sw s0, 4(sp)

sw s1, 8(sp)

sw s2, 12(sp)

sw s3, 16(sp)

sw s4, 20(sp)

li s0, 0 # pointer to the last node we handled

li s1, 0 # number of nodes handled

li s2, 5 # size

la s3, arrays

loop: #do...

li a0, 12

jal malloc # get memory for the next node

mv s4, a0

li a0, 20

jal malloc # get memory for this array

sw a0, 0(s4) # node->arr = malloc

lw a0, 0(s4)

mv a1, s3

jal fillArray # copy ints over to node->arr

sw s2, 4(s4) # node->size = size (4)

sw s0, 8(s4) # node-> next = previously created node

add s0, x0, s4 # last = node

addi s1, s1, 1 # i++

addi s3, s3, 20 # s3 points at next set of ints

li t6 5

bne s1, t6, loop # ... while i!= 5

mv a0, s4

lw ra, 0(sp)

lw s0, 4(sp)

lw s1, 8(sp)

lw s2, 12(sp)

lw s3, 16(sp)

lw s4, 20(sp)

addi sp, sp, 24

jr ra

fillArray: lw t0, 0(a1) #t0 gets array element

sw t0, 0(a0) #node->arr gets array element

lw t0, 4(a1)

sw t0, 4(a0)

lw t0, 8(a1)

sw t0, 8(a0)

lw t0, 12(a1)

sw t0, 12(a0)

lw t0, 16(a1)

sw t0, 16(a0)

jr ra

print\_list:

bne a0, x0, printMeAndRecurse

jr ra # nothing to print

printMeAndRecurse:

mv t0, a0 # t0 gets address of current node

lw t3, 0(a0) # t3 gets array of current node

li t1, 0 # t1 is index into array

printLoop:

slli t2, t1, 2

add t4, t3, t2

lw a1, 0(t4) # a0 gets value in current node's array at index t1

li a0, 1 # preparte for print integer ecall

ecall

li a1, ' ' # a0 gets address of string containing space

li a0, 11 # prepare for print string ecall

ecall

addi t1, t1, 1

li t6 5

bne t1, t6, printLoop # ... while i!= 5

li a1, '\n'

li a0, 11

ecall

lw a0, 8(t0) # a0 gets address of next node

j print\_list # recurse. We don't have to use jal because we already have where we want to return to in ra

malloc:

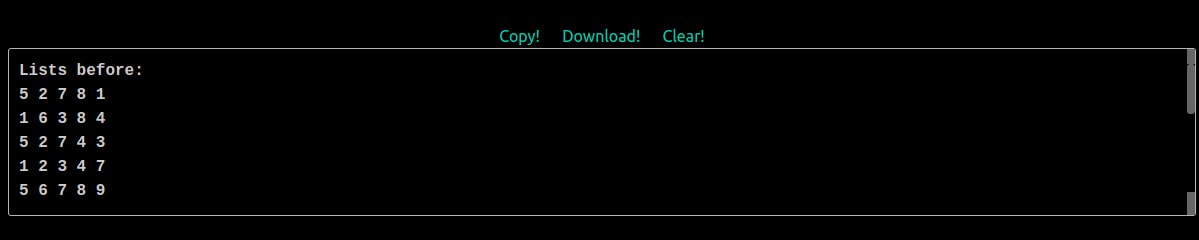
mv a1, a0 # Move a0 into a1 so that we can do the syscall correctly

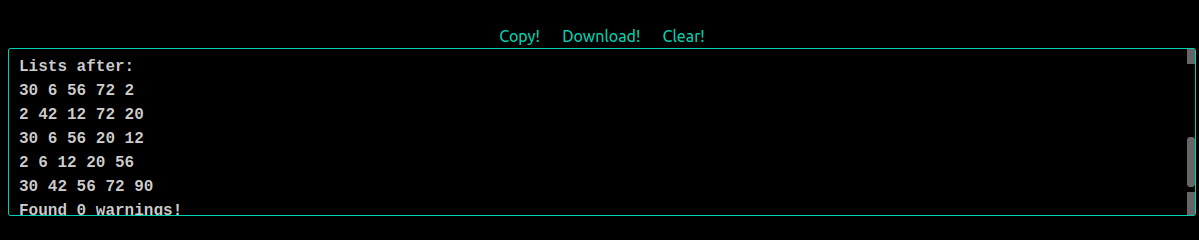
li a0, 9

ecall

jr ra

**Terminal Output:**

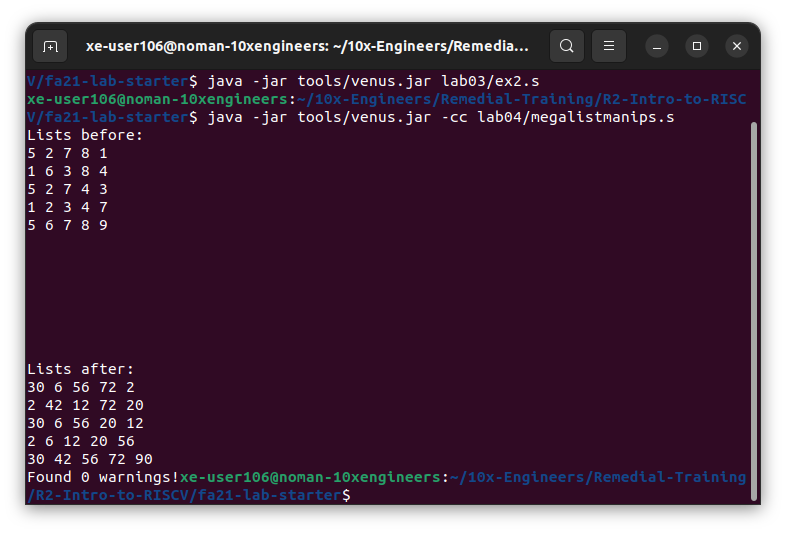




**‭Action Items:‬**

* **‭Find the six mistakes inside the map function in megalistmanips.s.**
  + In the **map** label (line 2) the stack space allocated was of 12 bytes but we need‬ actually 16 bytes since each node is of 12 bytes (- 4 for the array pointer - 4 for the size‬ of the array - 4 more for the pointer to the next node) we need to save these 12 bytes‬ plus the return address on the stack. Total of 16 bytes required.‬
  + In the **mapLoop** label we have to load the address of the array of current‬ node into **t1**. So in order to load the address into **t1** we should use **lw** instead of add‬ ‭Instruction.‬
  + In the **mapLoop** label we have to offset the array address by the count but‬ ‭here we are just adding the count to the address and not adding the offset at all.‬
  + In the **mapLoop** label we have to load the address of next node in **a0** so‬ ‭instead of using the **la** here which takes the label as an input we should use **lw‬** ‭instructions here.
  + In line 84 we have to put the address back into **a1**. We should just use **mv‬** ‭instructions to copy the address back into a1.
  + In the **done** label the stack space deallocated should be of 16 bytes rather than 12‬ bytes and at the end the it should return back to function. So **ret** instructions should be‬ ‭added at the end.‬

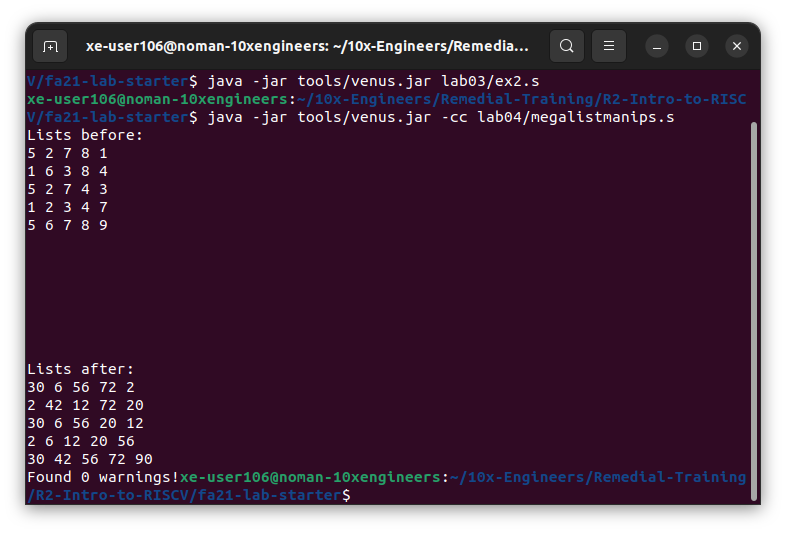
After resolving the above bugs, the code execution using -cc to check for calling convention violations. Here’s the output:



* **For this exercise, we are requiring that you don't use any extra save registers‬ ‭in your implementation. While you normally can use the save registers to store‬ ‭values that you want to use after returning from a function (in this case, when‬ ‭we're calling f in map), we want you to use temporary registers instead and follow‬ ‭their caller/callee conventions. The provided map implementation only uses the‬ ‭s0 and s1 registers, so we'll require that you don't use s2-s11.‬**
  + ‭Yes! I have not used any saved register other than **s0** and **s1** but I have used a‬ ‭temporary register **t3** to preserve the value of **t1** when **jalr s1** was called.
* **Make an ordered list of each of the six mistakes in the‬ ‭megalistmanips\_answers.txt file, and the corrections you made to fix them.**
  + Please refer to the attached **megalistmanips\_answers.txt** file. Alternatively, you can find the file here:

<https://github.com/ImNomanCR7/fa21-lab-starter/blob/main/lab04/megalistmanips_answers.txt>

* **Save your corrected code in the megalistmanips.s file. Use the -cc flag to run a‬ ‭basic calling convention check on your code locally:‬**

****

**Exercise 4: Finding and Solving Bugs:**

**‭Action Items:‬**

1. **‭Find the bugs in four of the five accumulators‬.**
   * **‬‭accumulatorone:** ‬‭In this accumulator the **s0** register was not saved onto the stack‬ ‭before it was used and so the value of **s0** was not preserved.‬
   * **‬‭accumulatortwo:** ‬‭The stack pointer is incremented by four in the prologue, and‬ ‭decremented by four in the epilogue. This breaks the stack of the caller.**‬**
   * **‬‭accumulatorthree: ‬‭**This accumulator is correct there is no issue in it.‬
   * **‬‭accumulatorfour: ‬‭**The function relies on **t2** being set to zero before running.‬ Otherwise, it returns t2 plus the desired sum.‬
   * **‬‭accumulatorfive:** ‬‭The function doesn't check if the first element is zero. As such, it‬ ‭fails to provide the correct result if the first element is zero, and if the second element is‬ ‭nonzero.

‬

1. **For each broken accumulator, write a test that fails on the broken one, but‬ ‭passes the correct implementation.‬**
   * + **Code Snippet:**

.import lotsofaccumulators.s

.data

inputarray: .word 1,2,3,4,5,6,7,0

inputarray1: .word 0,1,2,3,4,5,6,7,0

TestPassed: .asciiz "Test Passed!"

TestFailed: .asciiz "Test Failed!"

.text

# Tests if the given implementation of accumulate is correct.

#Input: a0 contains a pointer to the version of accumulate in question. See lotsofaccumulators.s for more details

#

#

#

#The main function currently runs a simple test that checks if accumulator works on the given input array. All versions of accumulate should pass this.

#Modify the test so that you can catch the bugs in four of the five solutions!

main:

la a0, inputarray # for accumulators 1 to 4

# \*\*\*\*\*\* Uncomment below statement for Accumulator 5 only \*\*\*\*\*\*\* #

la a0 inputarray1 # fail accumulatorfive

li s0, 3 #fail accumulatorone

li t2, 12 #fail accumulatortwo

addi sp, sp, -4

sw t2, 0(sp)

jal accumulatorfive

lw t1, 0(sp)

addi sp, sp, 4

li t3, 3

bne t3, s0, Fail #fail accumulatorone

bne t2, t1, Fail #fail accumulatortwo & accumulatorfour

beq a0 x0 Fail

li t0, 28

li t0, 0 # fail accumulatorfive (please uncomment for accumulator 5)

bne a0, t0, Fail

j Pass

Fail:

la a0 TestFailed

jal print\_string

j End

Pass:

la a0 TestPassed

jal print\_string

End:

jal exit

print\_int:

mv a1 a0

li a0 1

ecall

jr ra

print\_string:

mv a1 a0

li a0 4

ecall

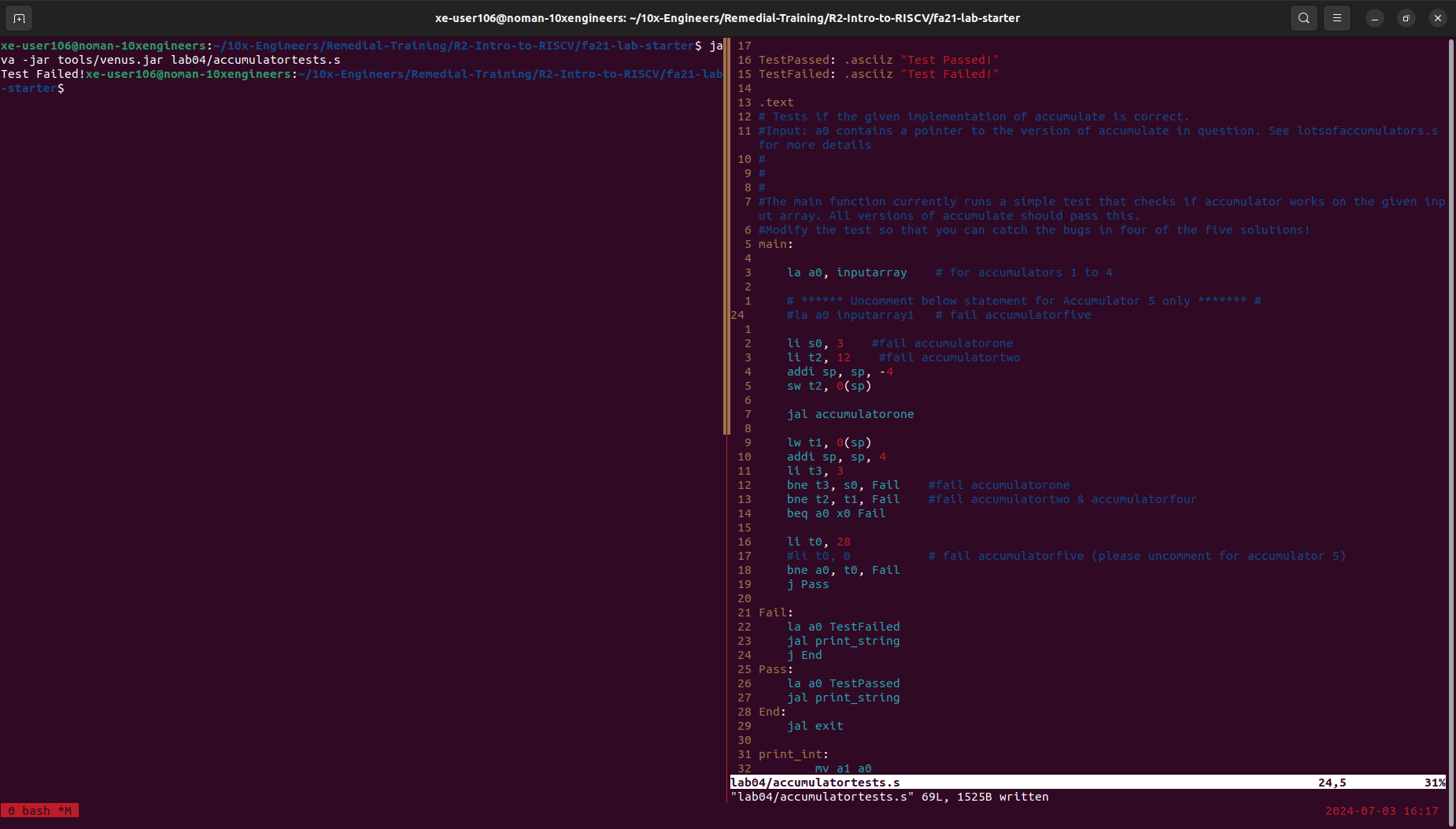
jr ra

exit:

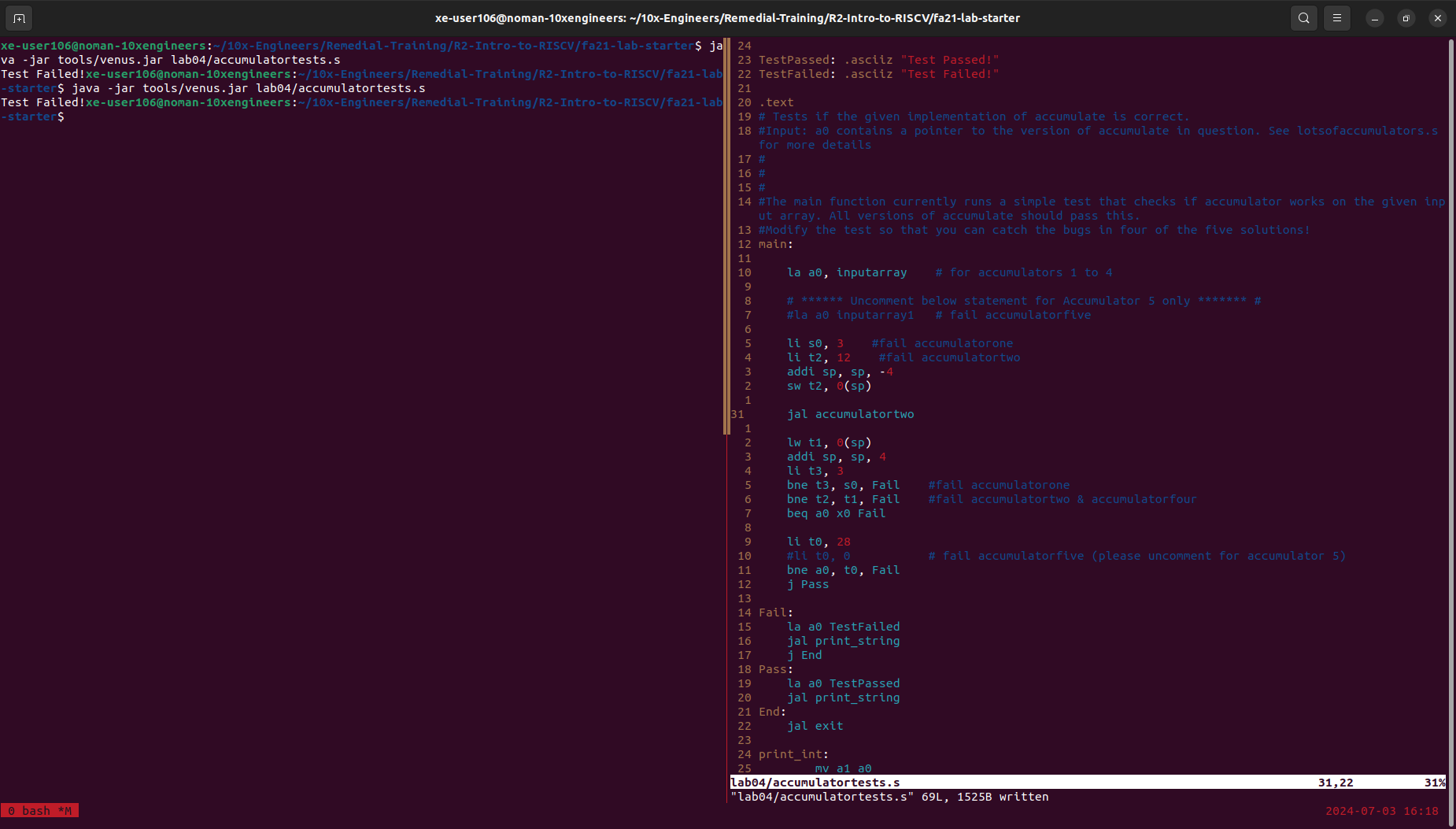
li a0 10

ecall

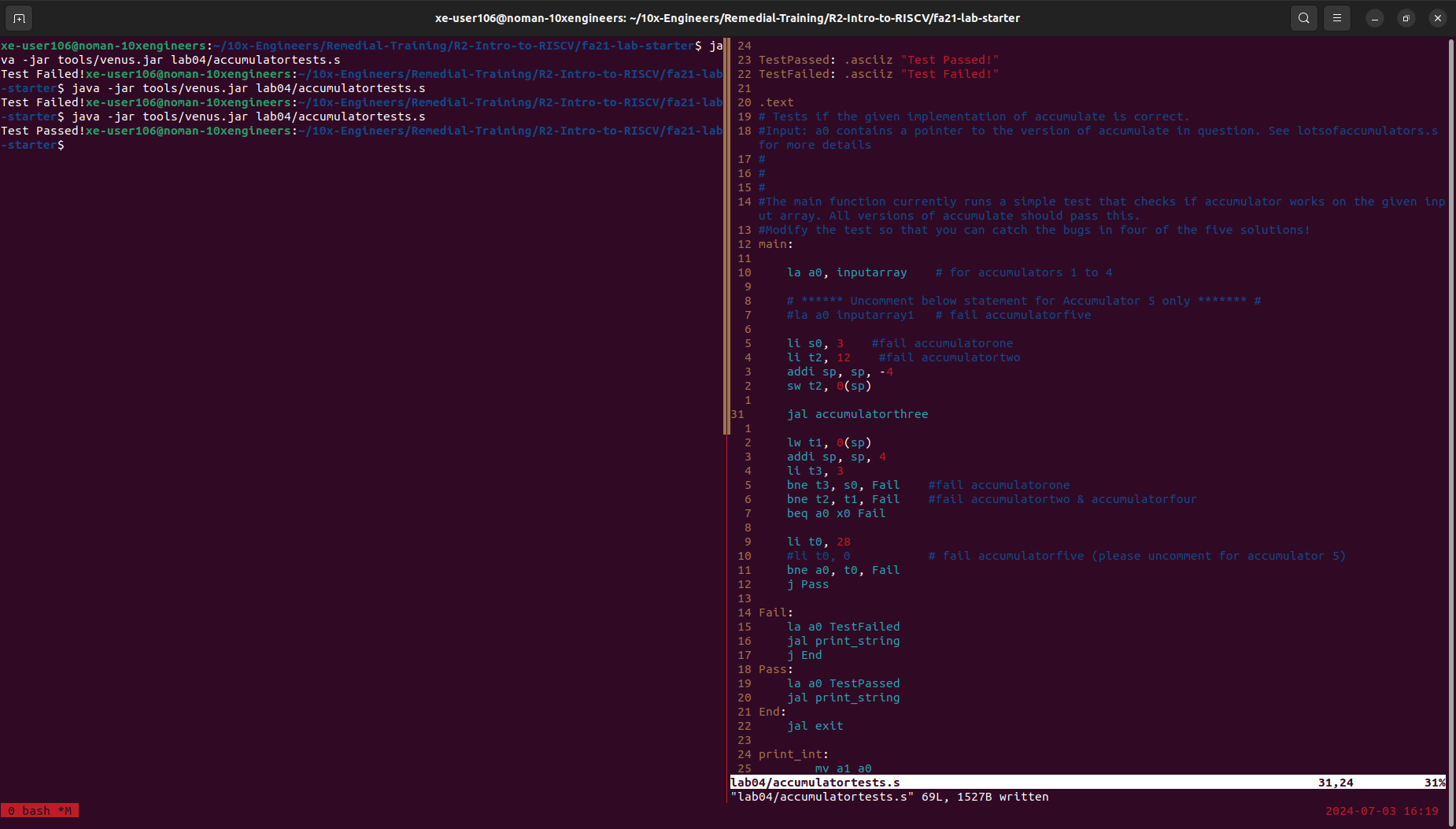
1. **Terminal Outputs for the Accumulators.**
   * **‬‭accumulatorone:**

****

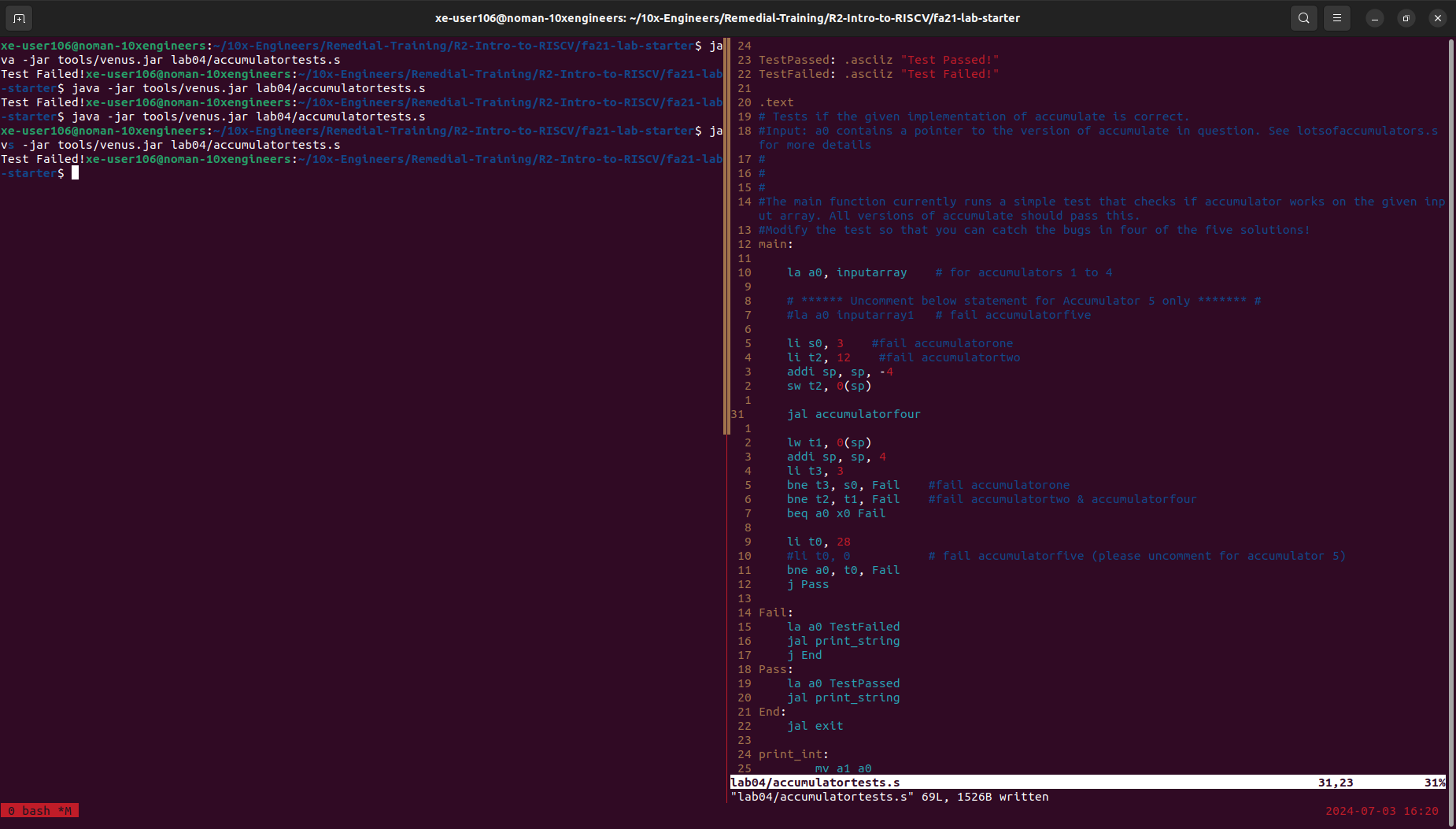
* + **‬‭accumulatortwo:**

****

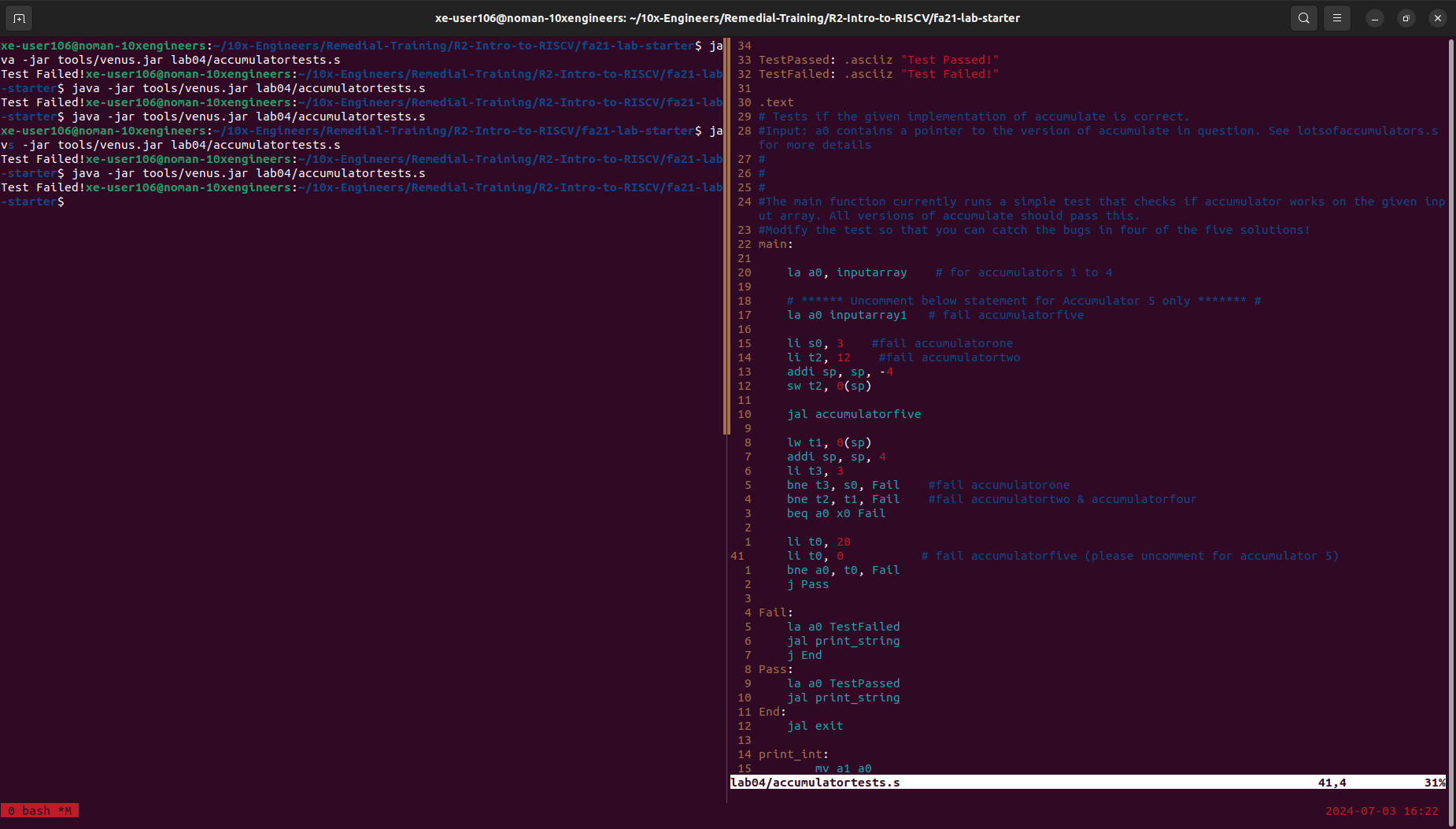
* + **‬‭accumulatorthree:**

****

* + **‬‭accumulatorfour:**

****

* + **accumulatorfive:**

****