# RECURSIVE SOLUTION OF THE TOWER OF HANOI PROBLEM IN RISC-V ASSEMBLY

COMPUTER ORGANIZATION AND ARCHITECTURE

daniel.juarez@iteso.mx, ID: 752809

2024-10-19

# Contents

# INTRODUCTION

## PURPOSE

The proposed solution contains RISC-V assembly code that aims to recursively solve the Tower of Hanoi problem. It also aims to be the translation of the C recursive solution from **https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi-2/**, with some differences that adapt certain elements that cannot be directly translated for practical needs such as showing in the STDOUT the disks movements, such problem required to show the intended movement in the **.Data** section.

## GITHUB REPOSITORY

HanoiASM

# SYSTEM OVERVIEW

## ARCHITECTURE

### ORIGINAL C CODE

The following code serves as a starting point to further implement the Tower of Hanoi problem in RISC-V assembly:

```c
#include <stdio.h>


void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
        return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
    return 0;
}
```

### RISC-V ASSEMBLY CODE

The recursive solution implemented in RISC-V assembly resulted to be the following:

```
# =======TOWER OF HANOI ON RISC-V ASSEMBLY=======
# AUTHOR: Daniel Alejandro Juarez Mota
# ===============================================
```

```
.text
    addi s0, zero, 0x3 # N
    lui a2, 0x10010 # POINTER TO TOWER A -> SRC
    addi a3, a2, 0x4 # TOWER B -> AUX
    addi a4, a2, 0x8 # TOWER C -> DST
    addi a3, a3, 0x60 # BOTTOM TOWER B
    addi a4, a4, 0x60 # BOTTOM TOWER C

LOOP:   blt t0, s0, ACCOMMODATE_DISKS # FOR LOOP TO ACCOMMODATE DISKS ON THE .DATA SECTION
    lui a2, 0x10010 # RESTORES TOWER A POINTER
    jal towerOfHanoi # FIRST CALL towerOfHanoi(n, 'A', 'C', 'B');
    jal exit

ACCOMMODATE_DISKS:
    addi t0, t0, 0x1 # FOR EACH DISK, ACCOMMODATE ON THE NEXT ROW OF THE FIRST .DATA SECTION
    ↪  COLUMN, WHERE ROWS ARE TOWERS
    sw t0, 0x0(a2)
    addi a2, a2, 0x20
    jal LOOP


towerOfHanoi:
    addi t0, zero, 0x1 # BASE CASE CONDITION
    beq s0, t0, TRUE # BASE CASE TO STOP RECURSIVE CALLS

    addi sp, sp, -0x4 # MEMORY ALLOCATION SPACE TO STORE "RA" AND N
    sw ra, 0x0(sp) # PUSH RA TO STACK
    addi sp, sp, -0x4
    sw s0, 0x0(sp) # PUSH N TO STACK
    addi s0, s0, -0x1 # (N - 1)

    # ----- SWAP ARGUMENTS (MOVE DISKS TO THE RIGHT) -----
    add t1, a3, zero # AUX -> TEMP
    add a3, a4, zero # AUX -> DST
    add a4, t1, zero # DST -> AUX/TEMP

    jal towerOfHanoi # FIRST RECURSIVE CALL towerOfHanoi(n-1, from_rod, aux_rod, to_rod)

    # ----- SWAP ARGUMENTS (MOVE DISKS TO THE LEFT) -----
    add t1, a3, zero # AUX -> TEMP
    add a3, a4, zero # AUX -> DST
    add a4, t1, zero # DST -> AUX/TEMP

    lw s0, 0x0(sp) # POP N OF THE STACK
    addi sp, sp, 0x4
    lw ra, 0x0(sp) # POP RA OF THE STACK
    addi sp, sp, 0x4


    sw zero, 0x0(a2) # CLEARS TOP OF TOWER
    addi a2, a2, 0x20 # DECREASE TOP OF TOWER
```

```
    addi a4, a4, -0x20 # INCREASE TOP OF TOWER
    sw s0, 0x0(a4)

    addi sp, sp, -0x4 # MEMORY ALLOCATION SPACE TO STORE "RA" AND N
    sw ra, 0x0(sp) # PUSH RA TO STACK
    addi sp, sp, -0x4
    sw s0, 0x0(sp) # PUSH N TO STACK
    addi s0, s0, -0x1 # (N - 1)

    add t1, a2, zero # TEMP -> SRC
    add a2, a3, zero # SRC -> AUX
    add a3, t1, zero # AUX -> TEMP

    jal towerOfHanoi # SECOND RECURSIVE CALL towerOfHanoi(n-1, aux_rod, to_rod, from_rod)

    add t1, a2, zero # TEMP -> SRC
    add a2, a3, zero # SRC -> AUX
    add a3, t1, zero # AUX -> TEMP

    lw s0, 0x0(sp) # POP N OF STACK
    addi sp, sp, 0x4
    lw ra, 0x0(sp) # POP RA OF THE STACK
    addi sp, sp, 0x4

    jalr ra


TRUE:
    # ----- MANAGE DISK MOVEMENT ON .DATA SECTION -----
    sw zero, 0x0(a2) # CLEARS TOP OF TOWER
    addi a2, a2, 0x20 # DECREASE TOP OF TOWER
    addi a4, a4, -0x20 # INCREASE TOP OF TOWER
    sw s0, 0x0(a4)


    jalr ra


exit: nop
```

## ASSEMBLY CODE SUMMARIZED EXPLANATION

- Registers used and purpose:

| register | usage |
|---|---|
| s0 | number of disks (int n) |
| a2 | pointer to tower A |

| register | usage |
|---|---|
| a3 | pointer to tower B |
| a4 | pointer to tower C |
| t0 | disk counter to accommodate disks |
| t1 | temporary register to avoid data loss |
| t2 | offset |

**CODE BREAKDOWN:**

```
.text
    addi s0, zero, 0x8 # N
    addi t2, zero, 0x20 # OFFSET
    mul t2, t2, s0
    lui s1, 0x10010 # POINTER TO TOWER A -> SRC
    addi s2, s1, 0x4 # TOWER B -> AUX
    addi s3, s1, 0x8 # TOWER C -> DST
    add s2, s2, t2 # BOTTOM TOWER B
    add s3, s3, t2 # BOTTOM TOWER

LOOP:   blt t0, s0, ACCOMMODATE_DISKS # FOR LOOP TO ACCOMMODATE DISKS ON THE .DATA SECTION
    lui a2, 0x10010 # RESTORES TOWER A POINTER
    jal towerOfHanoi # FIRST CALL towerOfHanoi(n, 'A', 'C', 'B');
    jal exit

ACCOMMODATE_DISKS:
    addi t0, t0, 0x1 # FOR EACH DISK, ACCOMMODATE ON THE NEXT ROW OF THE FIRST .DATA SECTION
↪   COLUMN, WHERE ROWS ARE TOWERS
    sw t0, 0x0(a2)
    addi a2, a2, 0x20
    jal LOOP
```

The code initializes necessary registers for needed values to call the **towerOfHanoi** function. The **main()** section begins initializing **N** disks in the **s0** registers as required, then it proceeds to load the offset between each section in .Data, then loads a memory address from the **.Data** section that serves as the top of the tower A, based on the value **a2** holds, the program will load the very next memory address for the rest of the towers. Once the top of the three towers is set, the program will place **a3** and **a4** at the bottom of their memory section to avoid placing disks one section above or below of their tower.

```
.text
    addi s0, zero, 0x8 # N
    addi t2, zero, 0x20 # OFFSET
    mul t2, t2, s0
    lui s1, 0x10010 # POINTER TO TOWER A -> SRC
    addi s2, s1, 0x4 # TOWER B -> AUX
    addi s3, s1, 0x8 # TOWER C -> DST
    add s2, s2, t2 # BOTTOM TOWER B
    add s3, s3, t2 # BOTTOM TOWER
```

After initializing necessary data, the program accommodates all **N** disks by the use of a loop, where if **t0** (which does not require initialization as it holds a value of **0x0** by default), is less than the number of disks, then **t0** will be updated indicating the program will load one disk, then store the **Nth** disk on **a2** and then add the value **0x20** to **a2**, where **0x20** is the value to jump to the next row in the **.Data** section. Once the loop accommodates all disks, it restores the value of **a2** and jump to the **towerOfHanoi** function.

```
LOOP:   blt t0, s0, ACCOMMODATE_DISKS # FOR LOOP TO ACCOMMODATE DISKS ON THE .DATA SECTION
    lui a2, 0x10010 # RESTORES TOWER A POINTER
    jal towerOfHanoi # FIRST CALL towerOfHanoi(n, 'A', 'C', 'B');
    jal exit

ACCOMMODATE_DISKS:
    addi t0, t0, 0x1 # FOR EACH DISK, ACCOMMODATE ON THE NEXT ROW OF THE FIRST .DATA SECTION
↪   COLUMN, WHERE ROWS ARE TOWERS
    sw t0, 0x0(a2)
    addi a2, a2, 0x20
    jal LOOP
```

The **towerOfHanoi** function contains the following base condition **(if n == 1)**:

```
addi t0, zero, 0x1 # BASE CASE CONDITION
beq s0, t0, TRUE # BASE CASE TO STOP RECURSIVE CALLS
```

If such condition is met, it jumps to the **TRUE** label:

```
TRUE:
    # ----- MANAGE DISK MOVEMENT ON .DATA SECTION -----
    sw zero, 0x0(a2) # CLEARS TOP OF TOWER
    addi a2, a2, 0x20 # DECREASE TOP OF TOWER
    addi a4, a4, -0x20 # INCREASE TOP OF TOWER
    sw s0, 0x0(a4)


    jalr ra
```

When the program enters in this section, it will clear whatever is on the top of the current **a2** tower, then as it removed a value, it decreases the top of the tower in one position and increase **a4** to store **s0** at the new top of **a4**. But in case the base condition is not met, it will execute the following code:

```
addi sp, sp, -0x4 # MEMORY ALLOCATION SPACE TO STORE "RA" AND N
sw ra, 0x0(sp) # PUSH RA TO STACK
addi sp, sp, -0x4
sw s0, 0x0(sp) # PUSH N TO STACK
addi s0, s0, -0x1 # (N - 1)

# ----- SWAP ARGUMENTS (MOVE DISKS TO THE RIGHT) -----
add t1, a3, zero # AUX -> TEMP
add a3, a4, zero # AUX -> DST
add a4, t1, zero # DST -> AUX/TEMP

jal towerOfHanoi
```

Before calling the function again, it allocates memory to firstly store tha **ra** value (first return address) and then it will push the initial number of disks onto the stack. Then it proceeds to swap the function arguments that correspond to the first function call and correspond to a disk movement to the right. Once the first recursive operation is done, the program proceeds to execute the following block:

```
add t1, a3, zero # AUX -> TEMP
add a3, a4, zero # AUX -> DST
add a4, t1, zero # DST -> AUX/TEMP

lw s0, 0x0(sp) # POP N OF THE STACK
addi sp, sp, 0x4
lw ra, 0x0(sp) # POP RA OF THE STACK
addi sp, sp, 0x4


sw zero, 0x0(a2) # CLEARS TOP OF TOWER
addi a2, a2, 0x20 # DECREASE TOP OF TOWER
addi a4, a4, -0x20 # INCREASE TOP OF TOWER
sw s0, 0x0(a4)
addi sp, sp, -0x4 # MEMORY ALLOCATION SPACE TO STORE "RA" AND N
sw ra, 0x0(sp) # PUSH RA TO STACK
addi sp, sp, -0x4
sw s0, 0x0(sp) # PUSH N TO STACK
addi s0, s0, -0x1 # (N - 1)

add t1, a2, zero # TEMP -> SRC
add a2, a3, zero # SRC -> AUX
add a3, t1, zero # AUX -> TEMP
```

The program will first swap arguments again emulating a disk movement to the left, the swap operation will not move a certain disk but load the target address to write the **N** disk on. After the swap operation,

it pops values pushed on the stack and porceeds to perform the same operaton found in the base case condition, finnally, the program will swap the arguments that correspond to the second recursive call. Once the second call is performed, it will only execute the following section:

```
add t1, a2, zero # TEMP -> SRC
add a2, a3, zero # SRC -> AUX
add a3, t1, zero # AUX -> TEMP

lw s0, 0x0(sp) # POP N OF STACK
addi sp, sp, 0x4
lw ra, 0x0(sp) # POP RA OF THE STACK
addi sp, sp, 0x4

jalr ra
```

The program will swap again the function arguments and pop data from stack, then it will return to the value **ra** holds at the moment of the execution.

## FLOWCHART

Flowchart

# TESTING

## TEST PLAN

The recursive solution will be tested using the following values applied to N:

- N = 3: Screenshots of the stack behaviour and .Data memory section will be taken.
- N = 8: IC to show the percentage of certain instructions needed.
- 4 <= N <= 15: Graph to show how N increases.

## TEST CASES

### N = 3

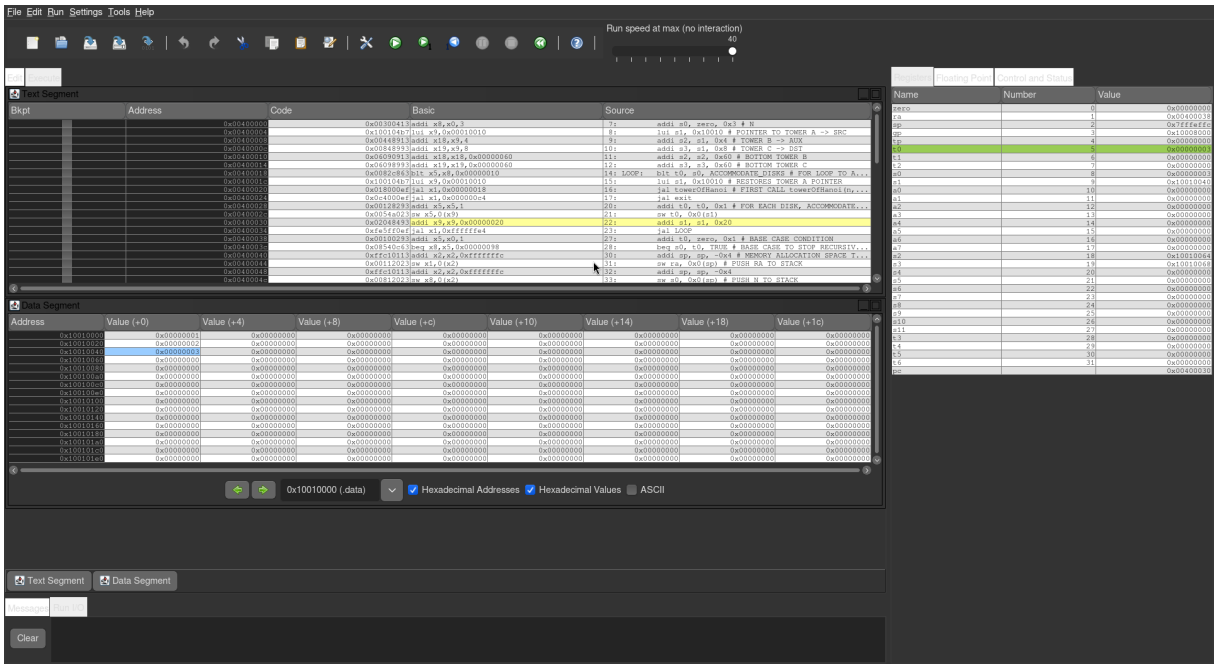


**Figure 1:** Disks loaded on .Data section

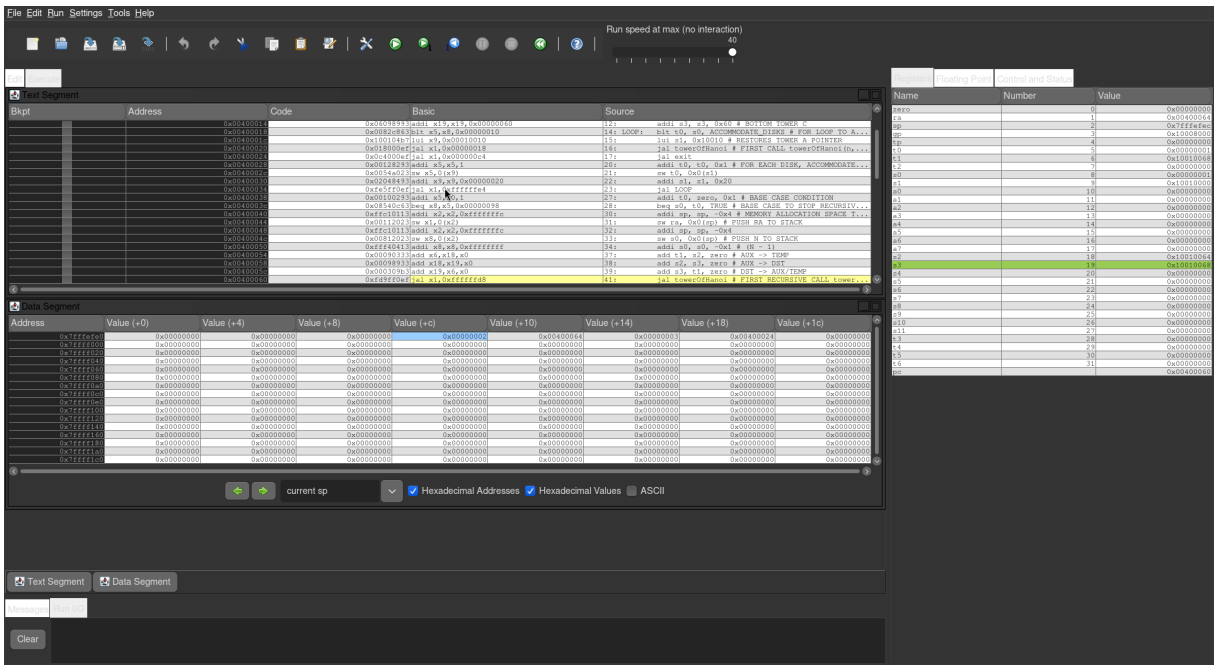

**Figure 2:** Stack values when reaching the first recursive call

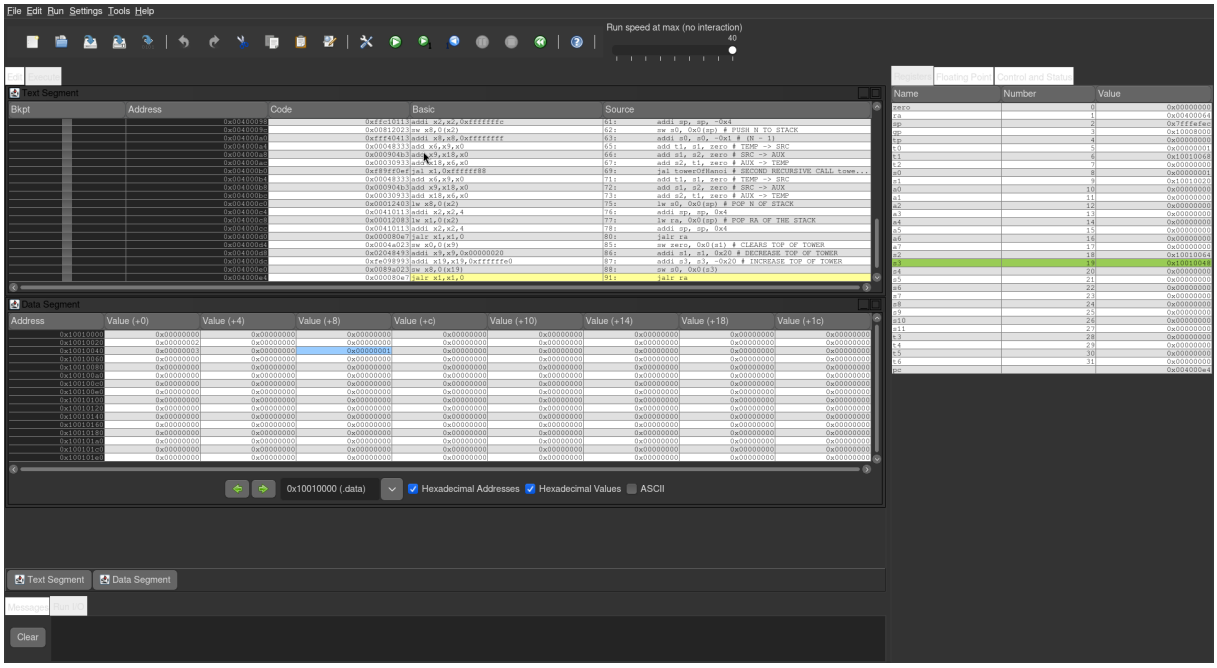

**Figure 3:** Disk 1 moved

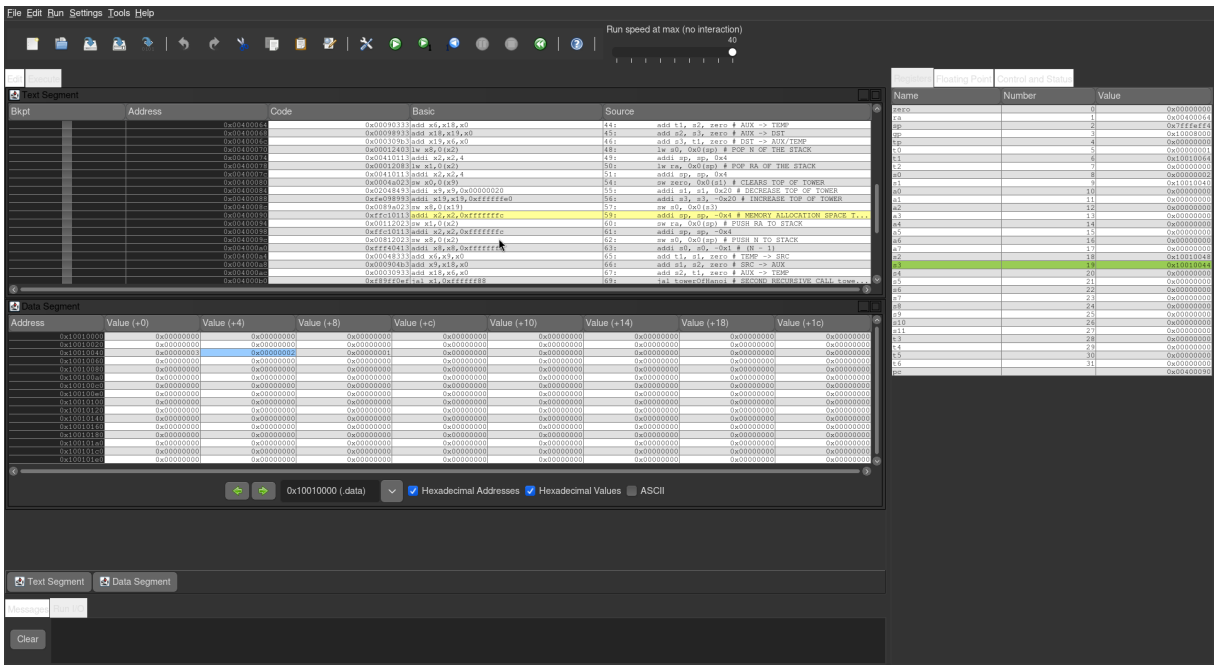

**Figure 4:** Disk 2 moved

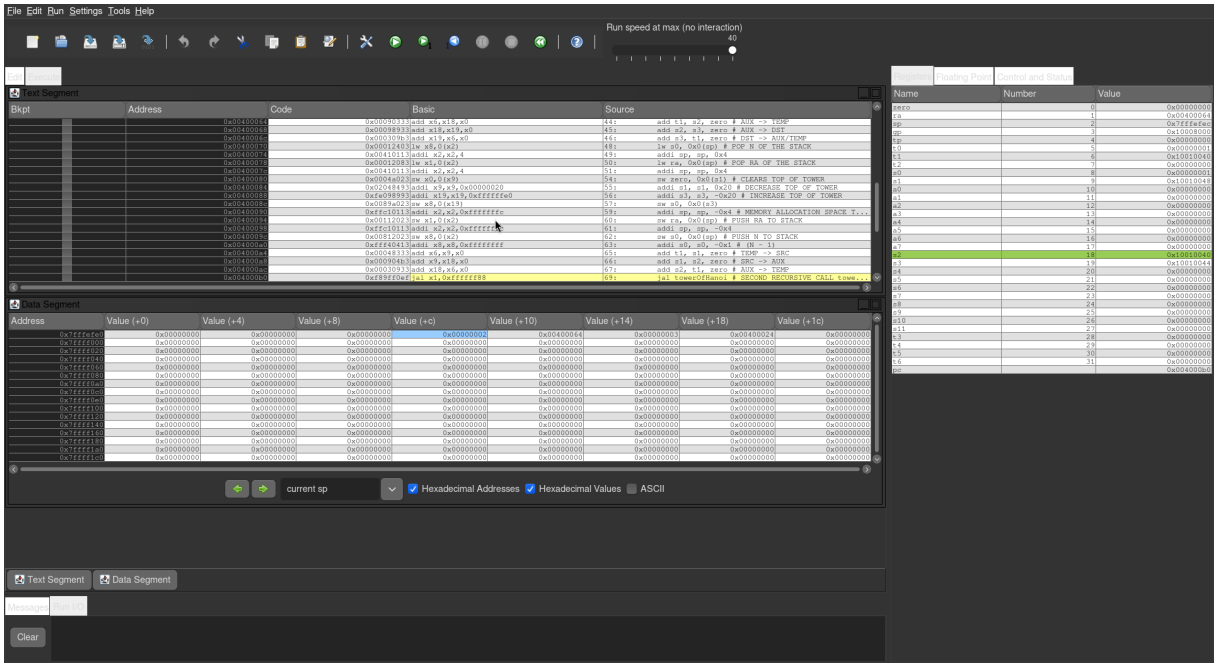

**Figure 5:** Stack values when reaching the second recursive call

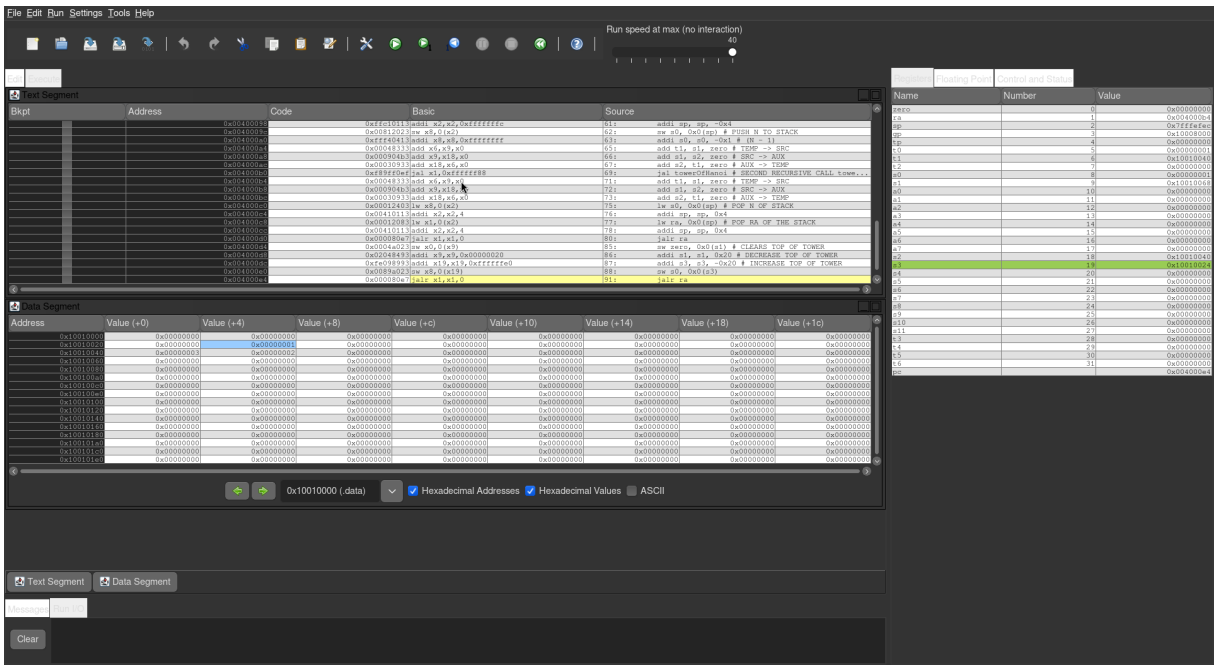

**Figure 6:** Disk 1 placed on top of disk 2

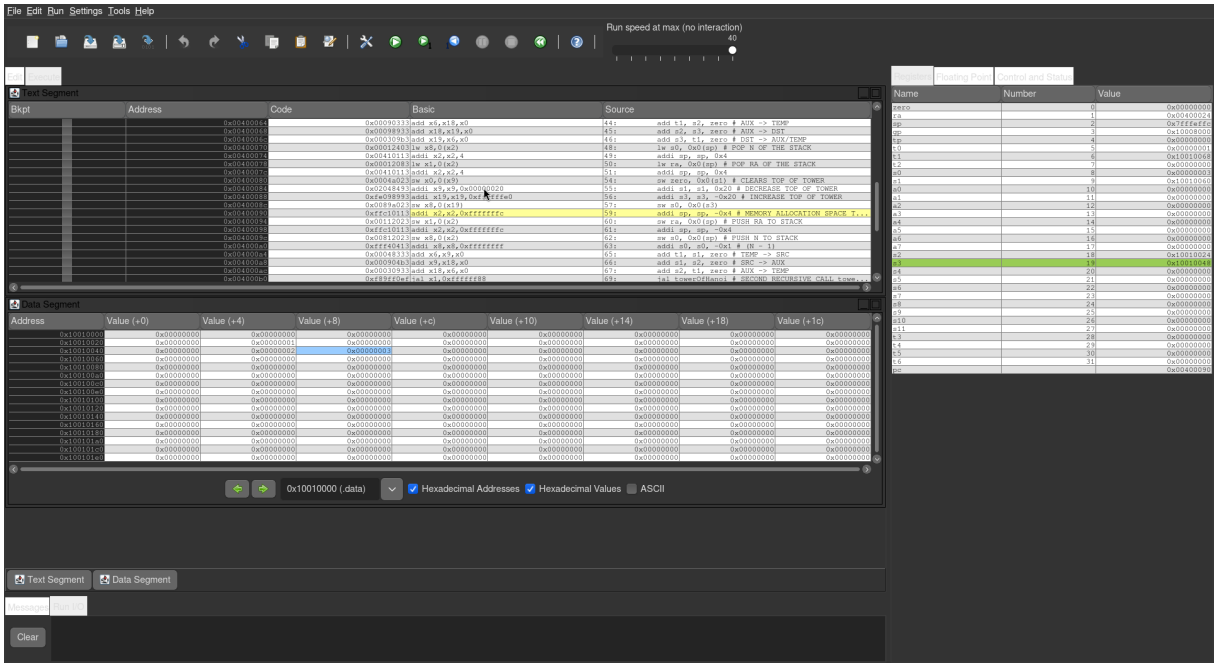

**Figure 7:** Disk 3 moved

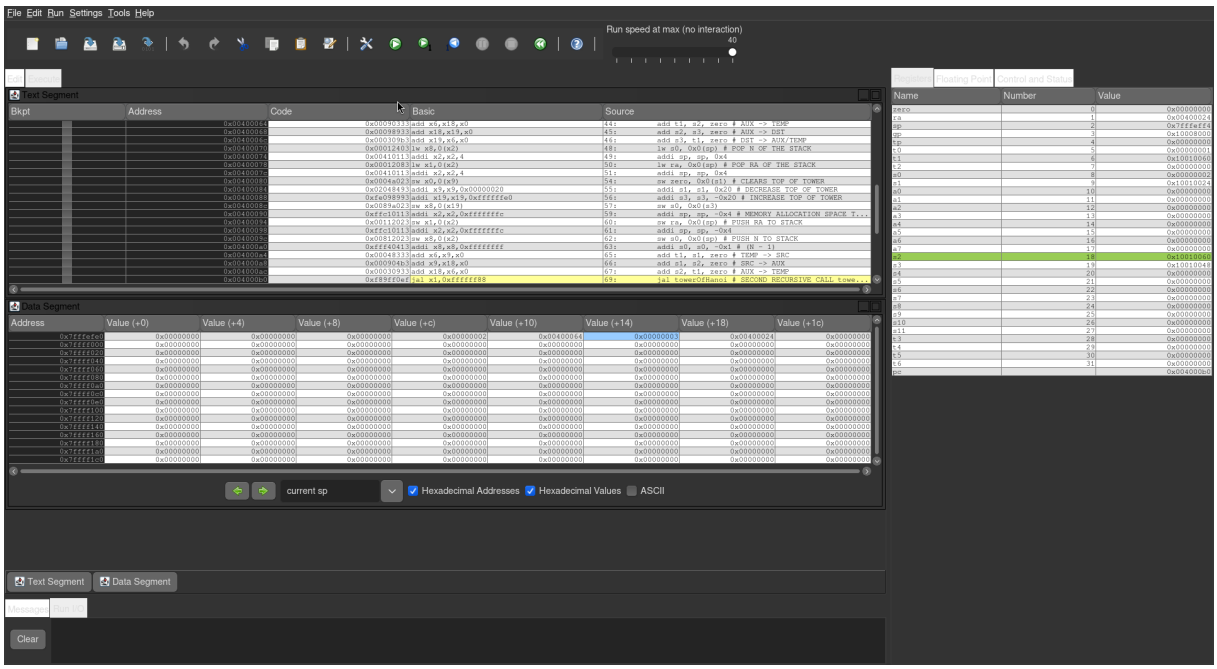

**Figure 8:** Stack values when reaching the second recursive call

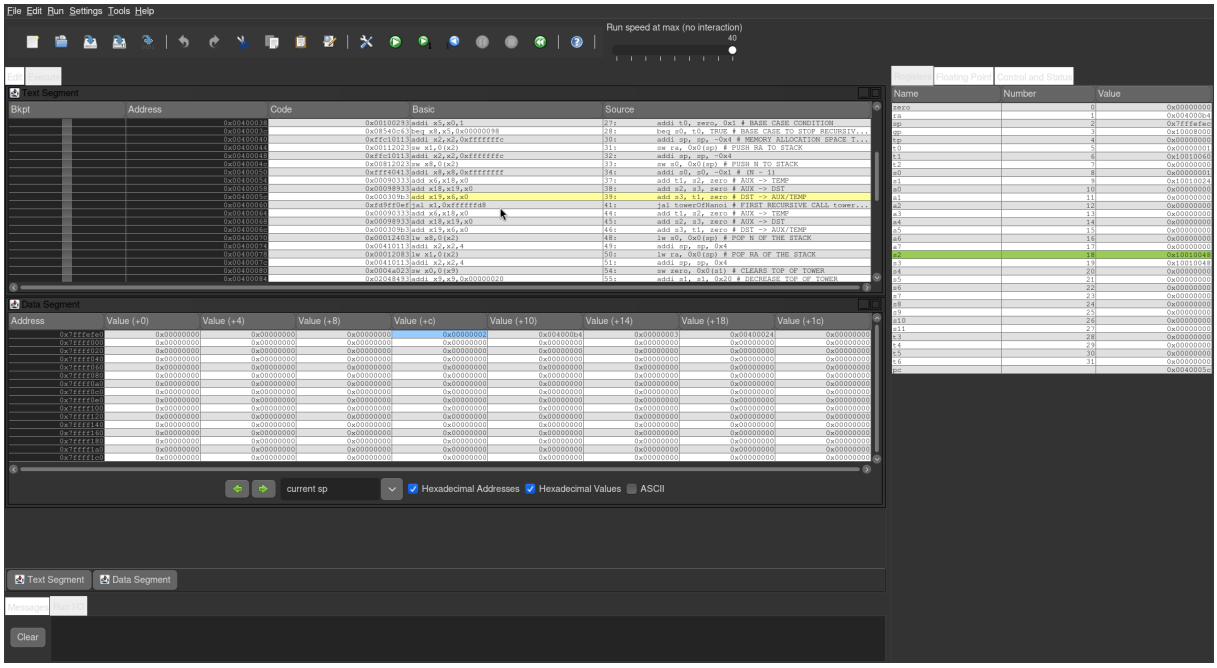

**Figure 9:** Stack values after the second recursive call

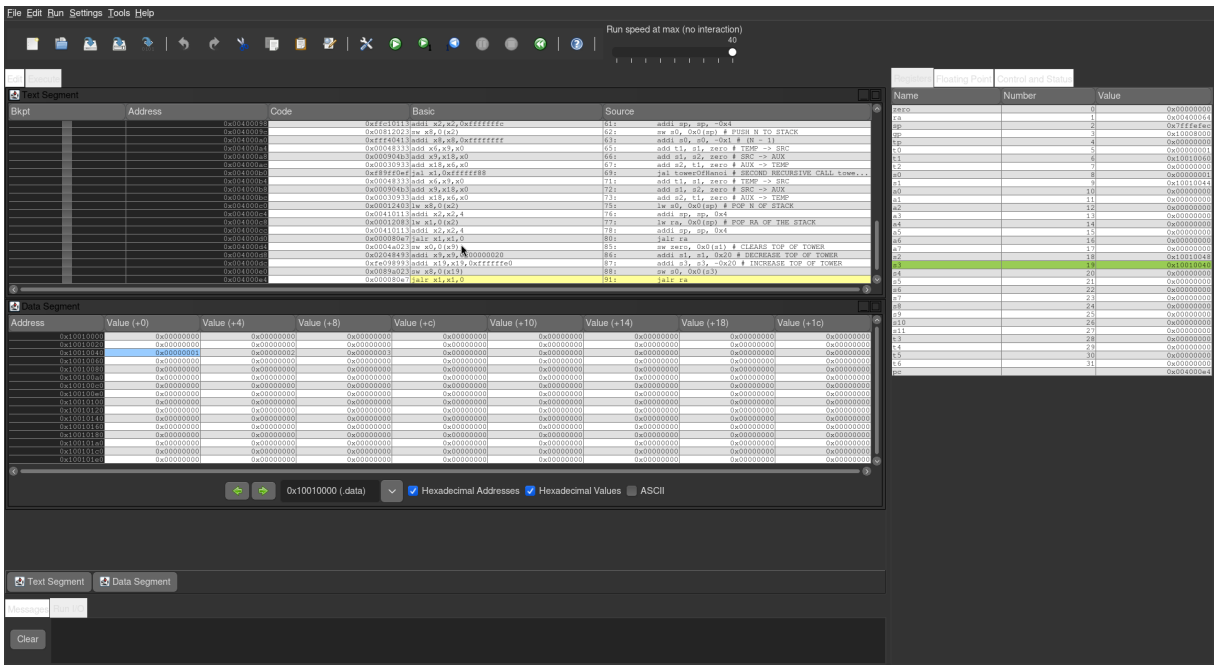

**Figure 10:** Disk 1 moved

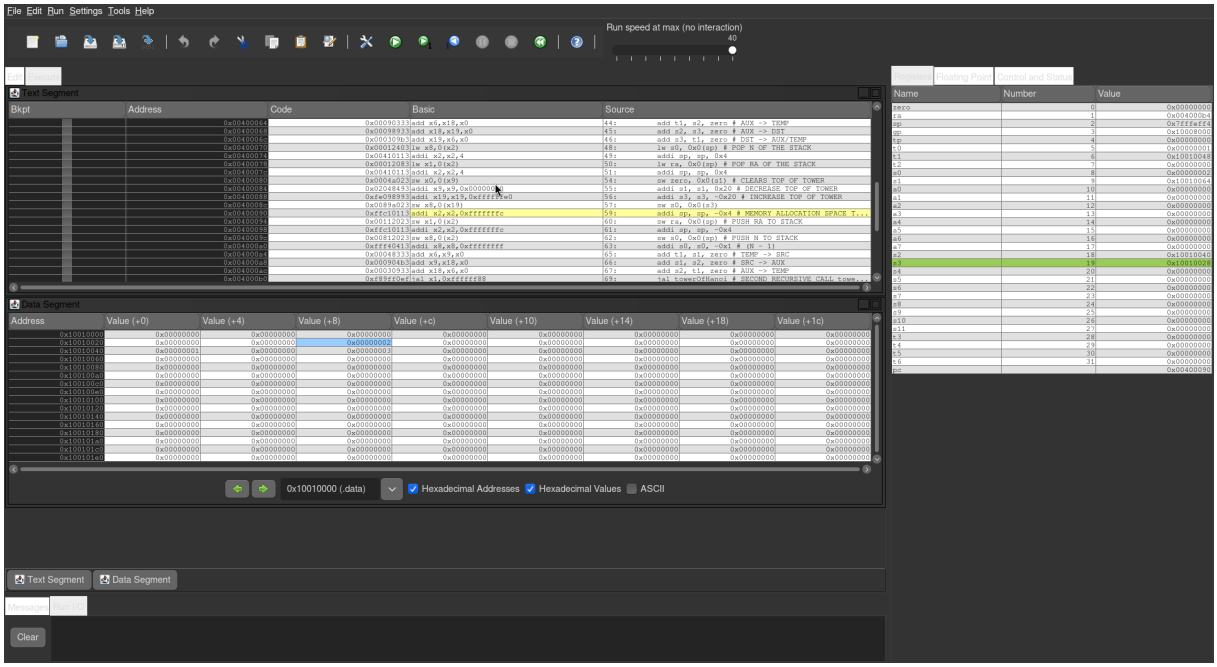

**Figure 11:** Disk 2 placed on top of disk 3

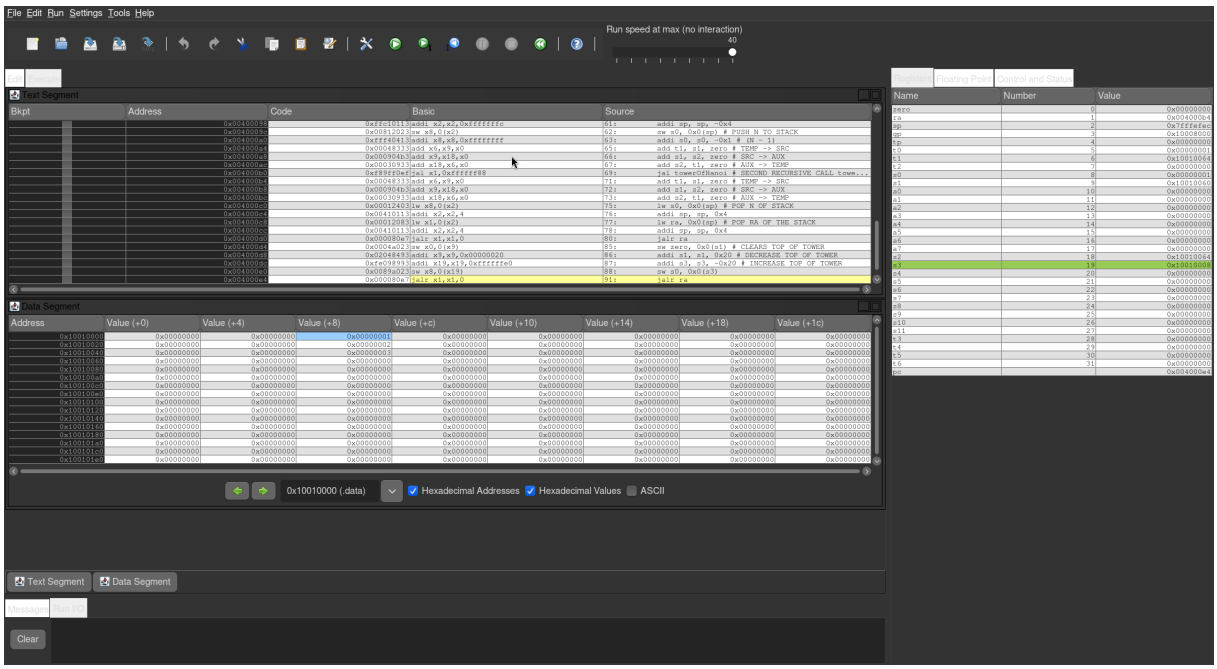

**Figure 12:** All three disks are accommodated on the DST tower

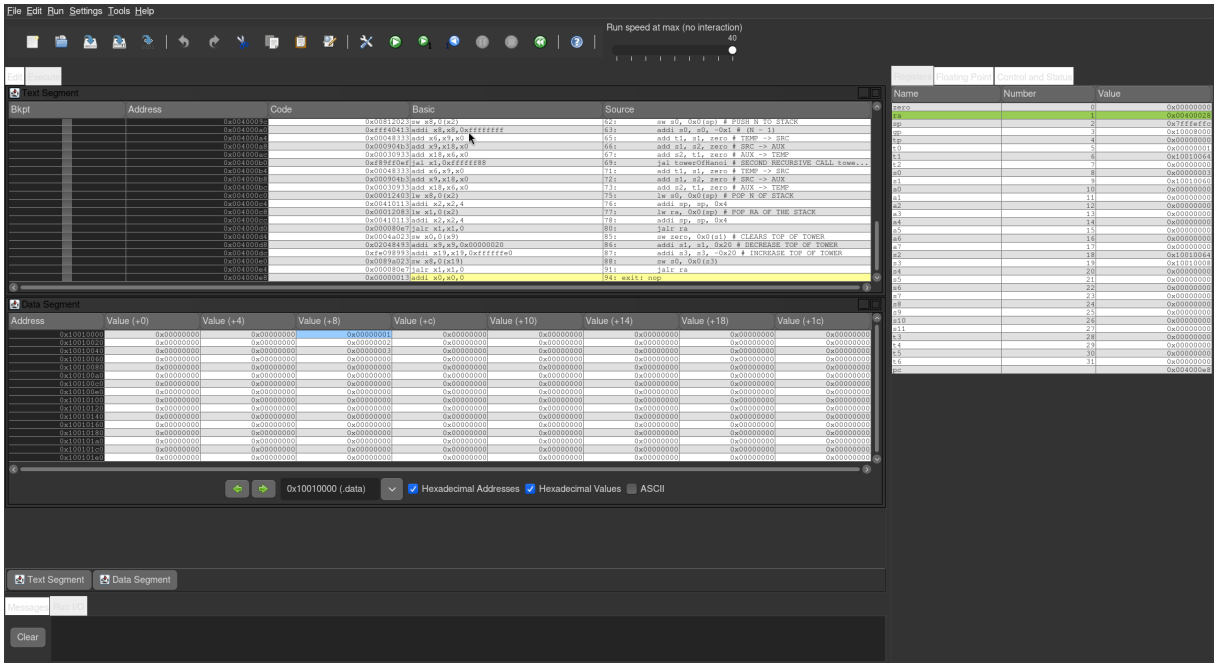

**Figure 13:** Program ends correctly

**N = 8**

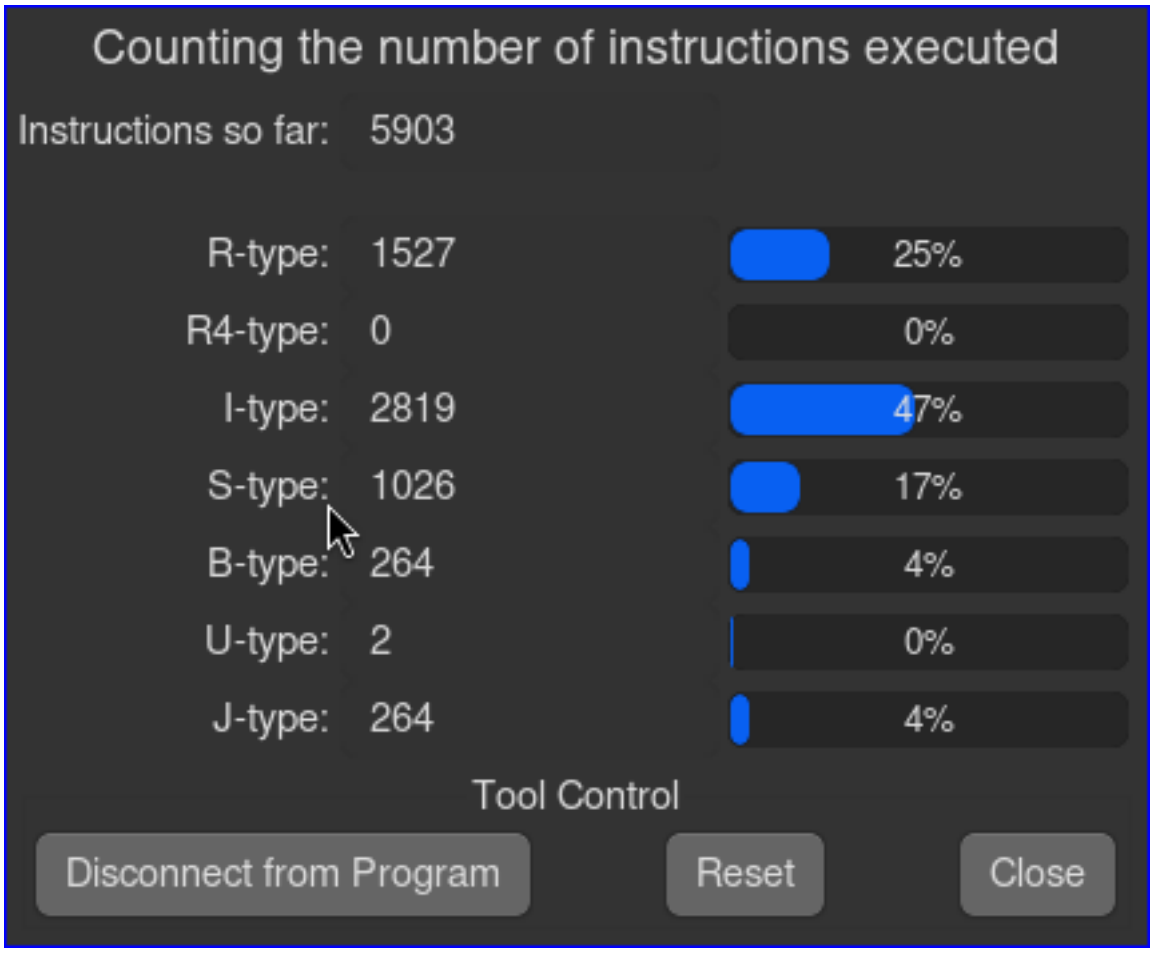


**Figure 14:** Percentage of each instruction when N = 8

**4 <= N <= 15**

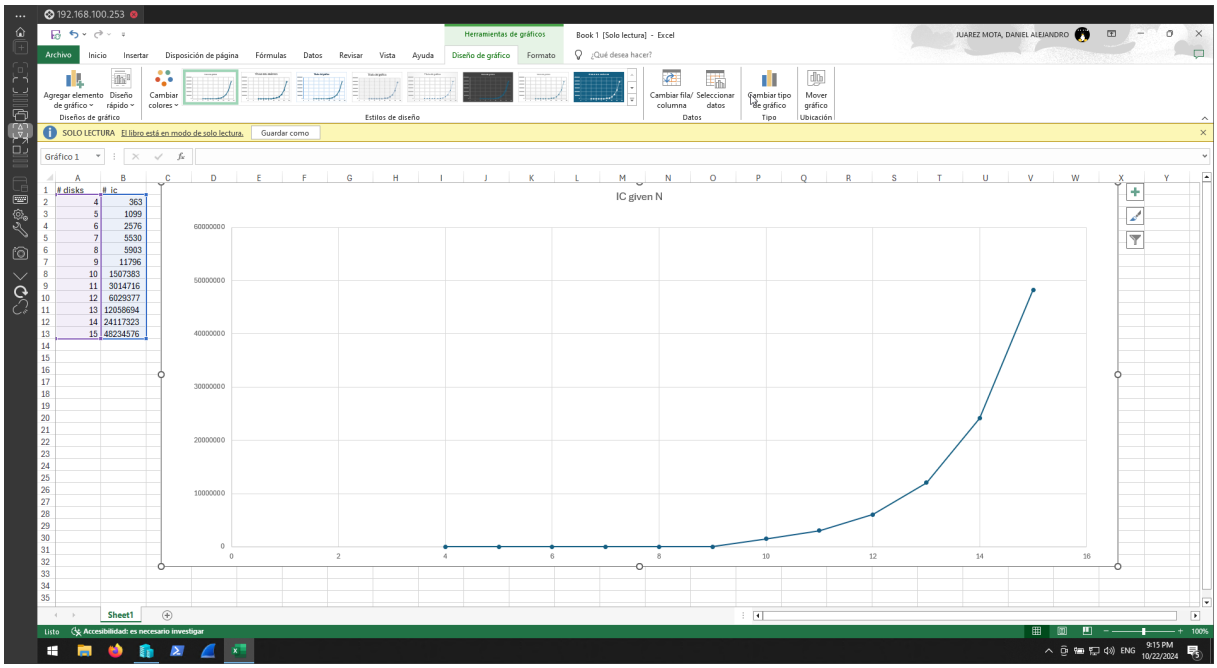


**Figure 15:** IC increment given N

# CONCLUSIONS

In conclusion, the Tower of Hanoi problem could successfully be applied in RISC-V assembly using recursivity, operations with the stack and manipulations of the **.Data** memory section to illustrate disks movements, nonetheless the IC graph demonstrates the implemented algorithm **is not** an optimal solution in terms of time and efficiency given **N > 10**, since **IC** will grow exponentially.