**EGC-121 Computer Architecture**

Assignment 1

Report

By Hosdurga Rohan Vittal Kamath

Roll Number : BT2024238

1. Stein’s Algorithm to compute GCD of 2 numbers

**Objective :** The main objective of the MIPS assembly program is to compute the GCD of 2 numbers using the Stein’s Algorithm. It takes 2 positive integer inputs from the user, computes the GCD and prints the result.

**Logic:**

Stein’s Algorithm is based on bitwise operations and follows these steps:

1. If either number is 0, return the other number as GCD.
2. Remove common factors of 2 by shifting both numbers right (dividing by 2) until at least one is odd.
3. Make one number odd by repeatedly shifting it right.
4. Subtract the smaller number from the larger one and continue until one becomes zero.
5. Restore the common power of 2 removed in step 2 by left-shifting the result.

**Implementation:**

Data segment

.data

msg1: .asciiz "Enter the first number: "

msg2: .asciiz "Enter the second number: "

finalMsg: .asciiz "The GCD of the two numbers you entered is: "

newline: .asciiz "\n"

* Messages to be printed from time to time

Taking user input:

*#Taking the input*

     li $v0, 4

     la $a0, msg1

     syscall

     li $v0, 5

     syscall

     add $s0, $v0, $0

     li $v0, 4

     la $a0, msg2

     syscall

     li $v0, 5

     syscall

     add $s1, $v0, $0

The program first prints a message prompting the user to enter the first number.

It then reads the user’s input and stores it in register $s0.

The same process is repeated for the second number, which is stored in $s1.

Calling the GCD function

*#The 2 numbers entered are our arguments*

     add $a2, $s0, $0    *# $a2 = a*

     add $a3, $s1, $0    *# $a3 = b*

*#Calling the GCD function*

     jal gcd

     add $s2, $v0, $0

* The two numbers entered are stored in registers $a2 and $a3 as they are the procedure arguments.
* Then “jal gcd” calls the gcd procedure which computes the GCD of the two numbers and stores the result in $v0. The result is stored in $s2 after returning from the fucntion.

Printing the result

*#Printing the GCD of the 2 numbers*

      li $v0, 4

      la $a0, finalMsg

      syscall

      li $v0, 1

     add $a0, $s2, $0

      syscall

* Printing the result.

GCD function

gcd:

     beq $a2, $0, returnSecond      *# if(a==0) return b;*

     beq $a3, $0, returnFirst      *# if(b==0) return a;*

     li $t0, 0                  *# k=0*

* If one of the numbers is zero, then the other number is returned

for\_loop:

         andi $t1, $a2, 1

         andi $t2, $a3, 1

         or $t3, $t1, $t2

         bne $t3, $0, whileLoop1

         srl $a2, $a2, 1        *# a>>=1*

         srl $a3, $a3, 1        *# b>>=1*

         addi $t0, $t0, 1       *# k++*

         j for\_loop

* If both numbers are even, divide by 2 and increment k.

whileLoop1:

           andi $t3, $a2, 1         *# a&1*

           bne $t3, $0, whileLoop2  *# if a&1 != 0 we exit*

           srl $a2, $a2, 1          *# a>>=1*

           j whileLoop1

* Aim is to make the first number (say a) odd. So, the code keeps dividing it by 2 until it becomes odd.

whileLoop2:

andi $t3, $a3, 1 # b&1

bne $t3, $0, compare # if a&1 !=0 we exit

srl $a3, $a3, 1 # Otherwise, keep dividing b by 2

j whileLoop2

* The same is done with the second number until it beomes odd.

compare:

slt $t3, $a3, $a2 # check if b < a

beq $t3, $0, subtract # if b >= a, we just do the subtraction

# Else we do the swap

add $t4, $a2, $0

add $a2, $a3, $0

add $a3, $t4, $0

* If b<a, swap a and b so that a is always smaller

subtract:

sub $a3, $a3, $a2 # b-=a

bne $a3, $0, whileLoop2 # if b!=0, we continue the iterations

sllv $v0, $a2, $t0 # a<<k

# returning our answer

jr $ra

* Subtract a from b. Continue until b becomes 0.
* Multiply the result by 2^k to restore the common factor of 2 removed earlier and return.

returnFirst: add $v0, $a2, $0

jr $ra

returnSecond: add $v0, $a3, $0

jr $ra

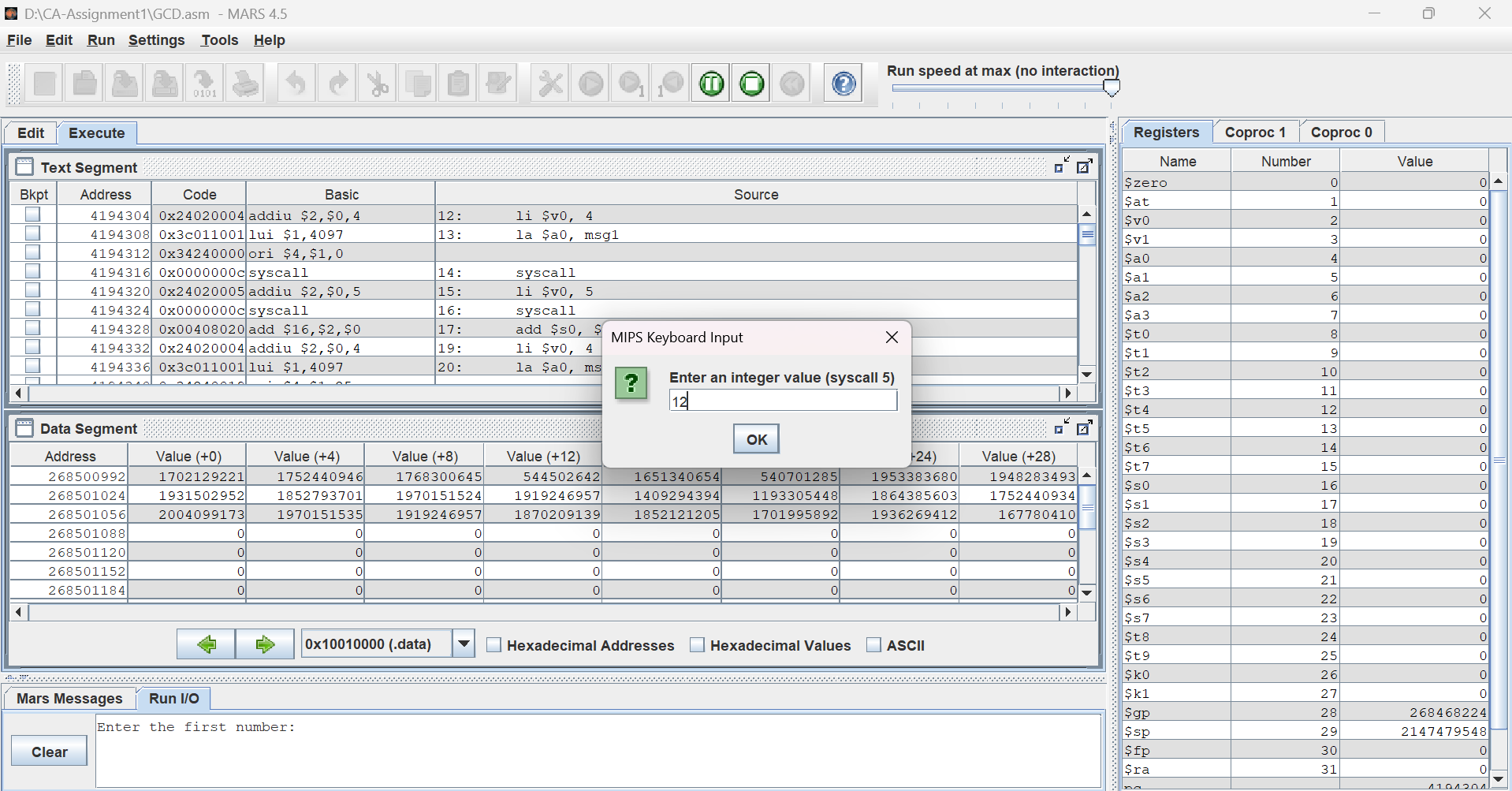
* If one number becomes zero, return the other

**Observations:**

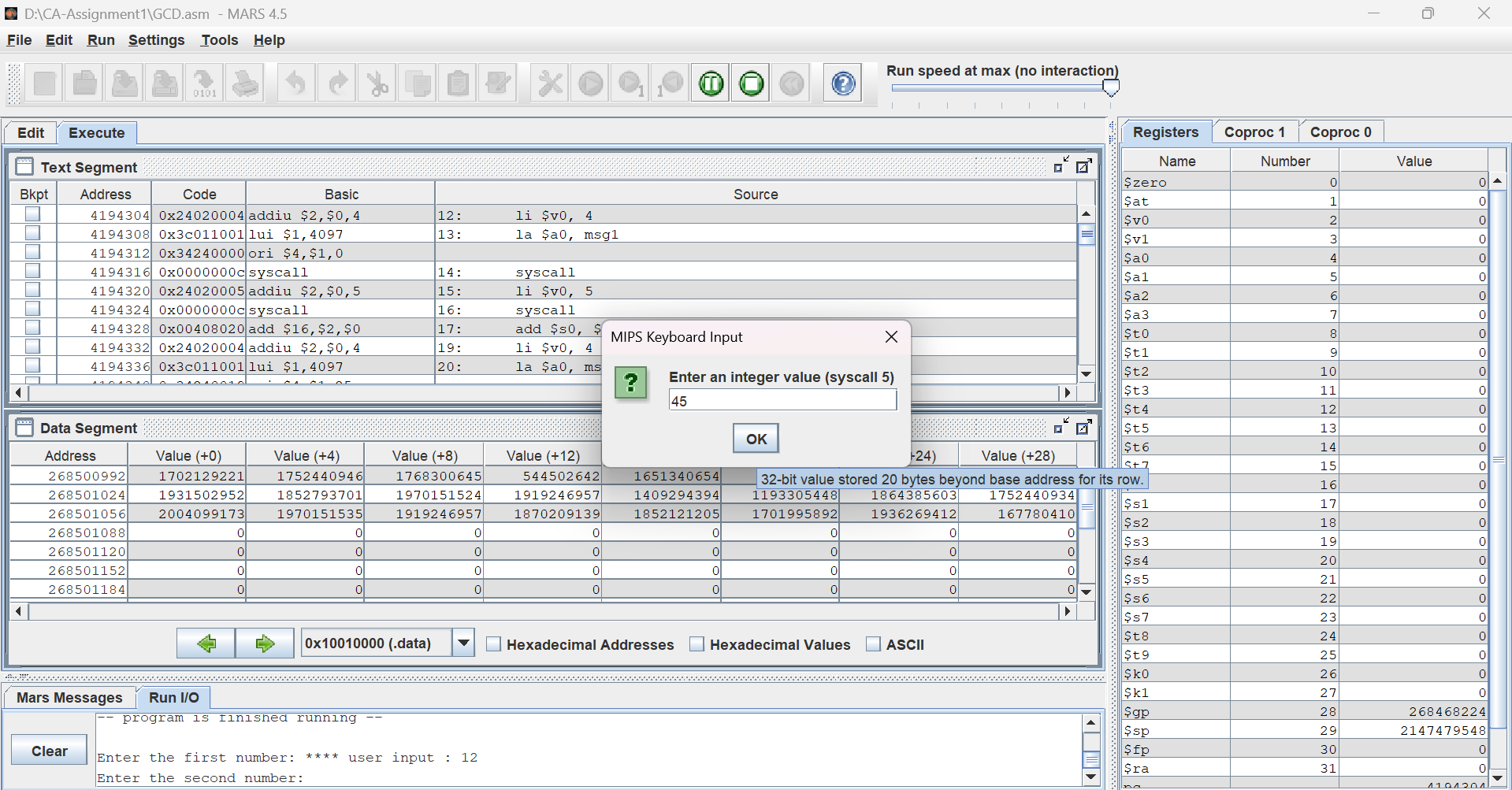
* Using bitwise shifts for division by 2 makes the algorithm fasters
* Both the numbers are ensured to be odd before performing subtarction

**Conclusion:**

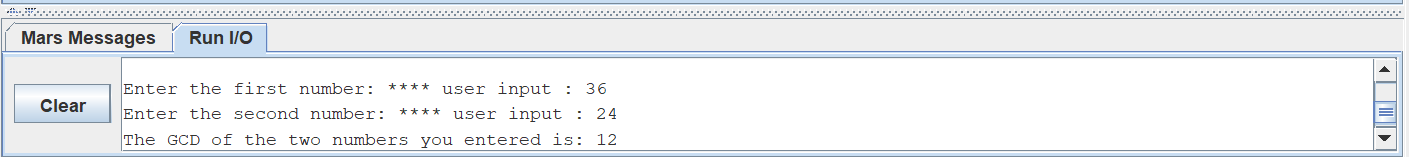
* The program correctly computes the GCD of 2 numbers entered by the user
* It follows Stein’s algorithm, making it faster than the typical Euclidean algorithm to compute the GCD of 2 numbers



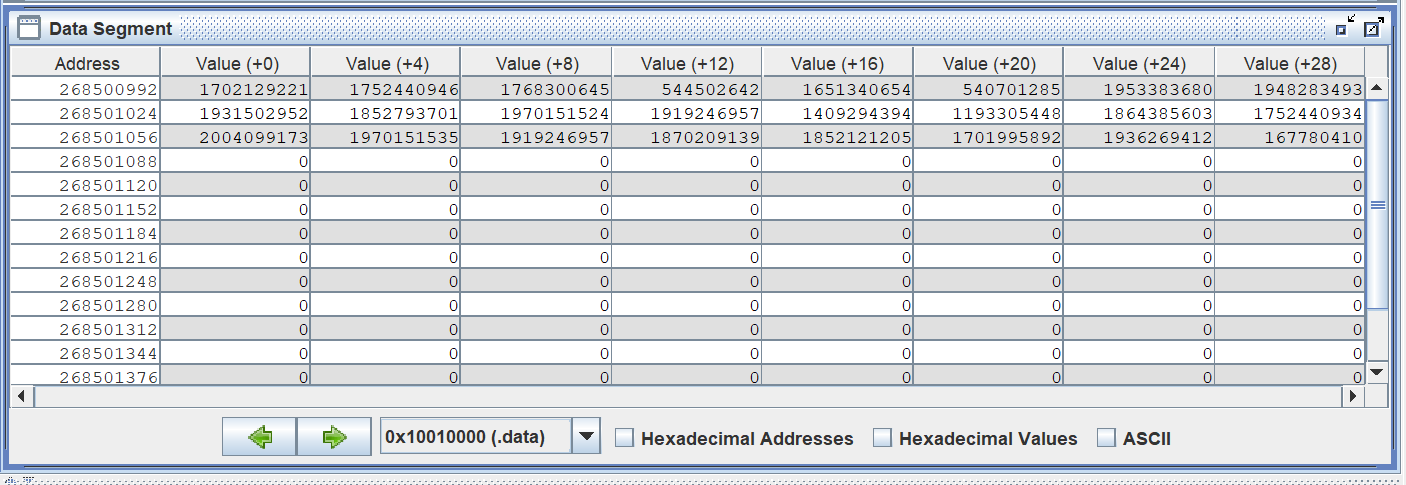
Taking user input for the first number (On MARS simulator)



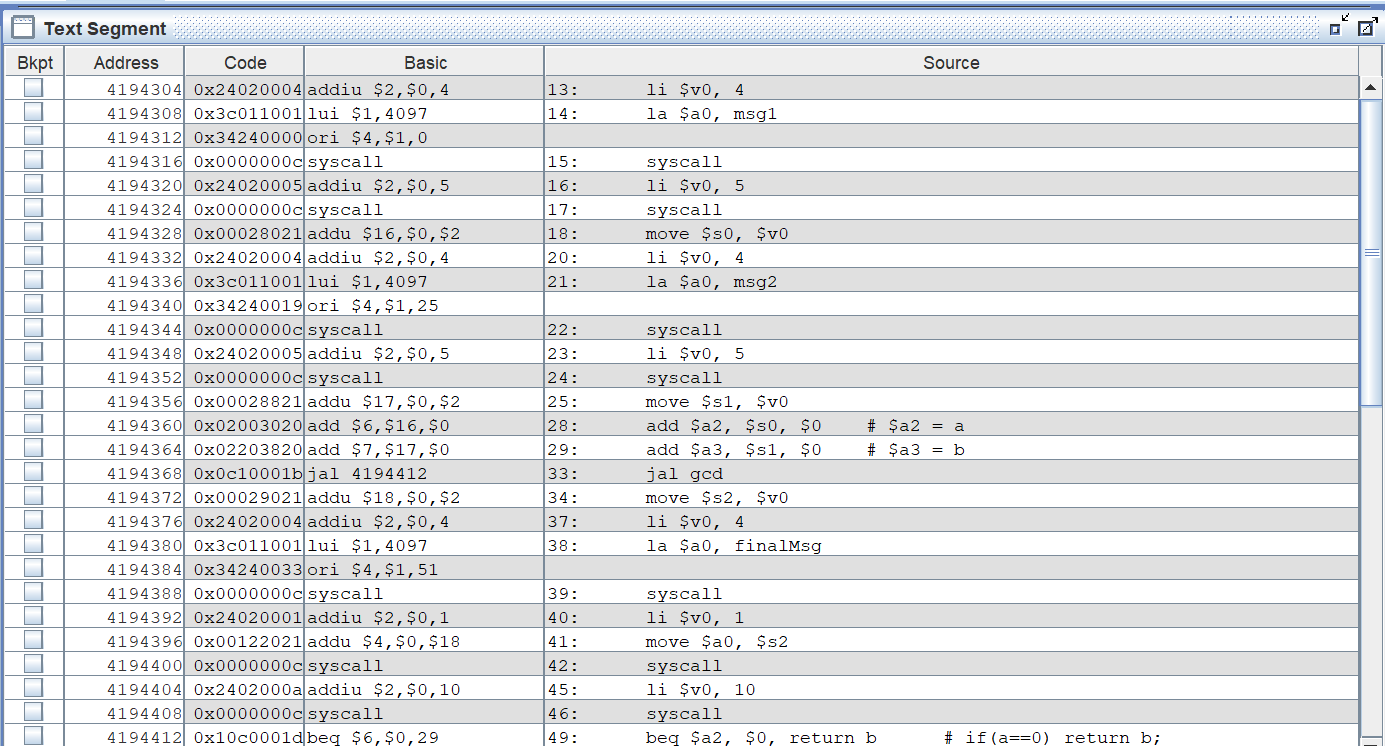
Taking user input for the second number

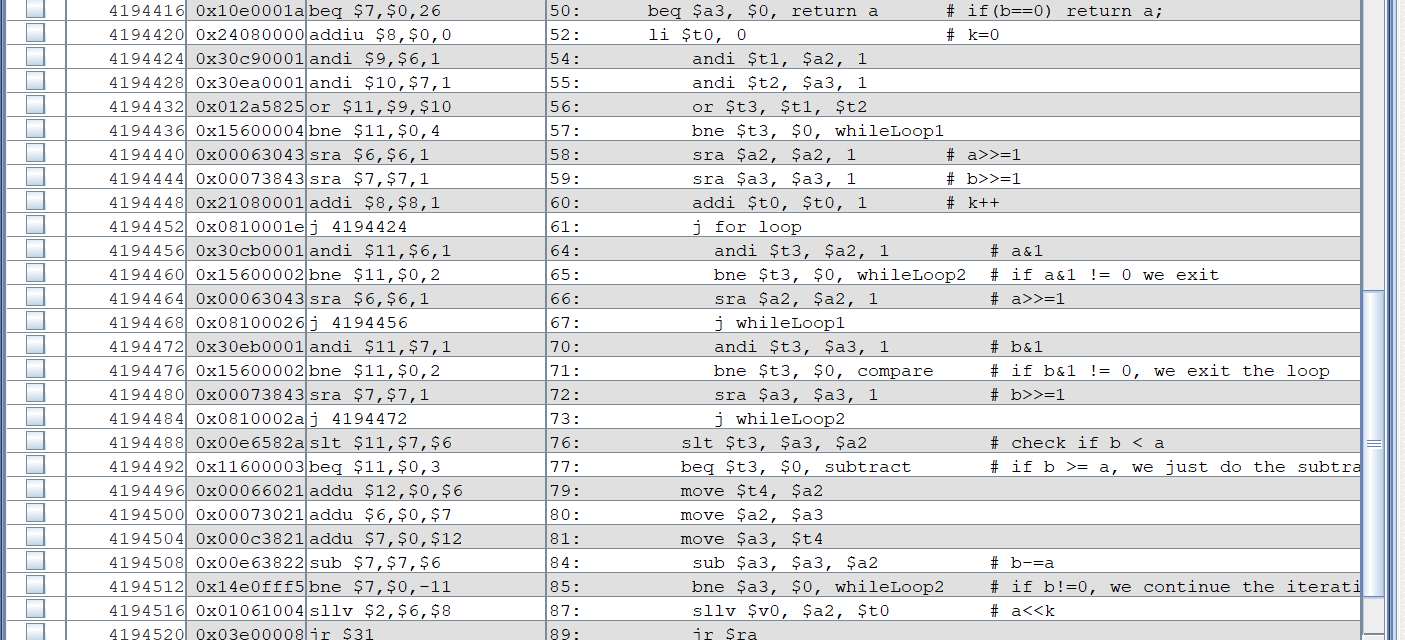


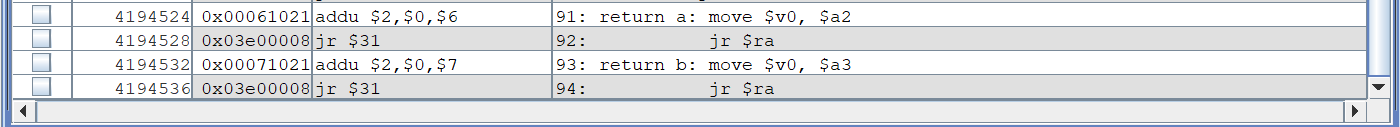
Displaying the result of the algorithm



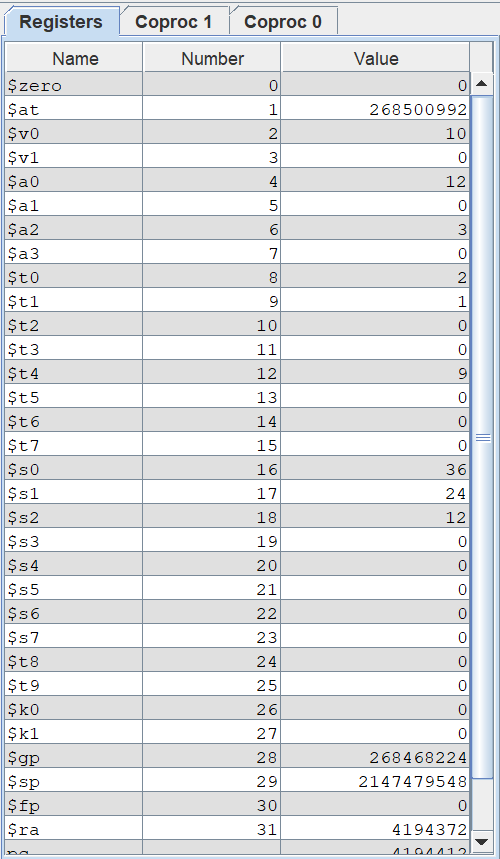
Data segment after execution of the program

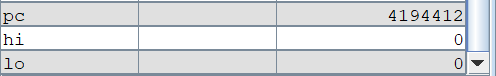






Text segment after execution of the program





Register File

1. Iterative Binary Search Algorithm

**Objective:** The MIPS assembly code implements the iterative binary search algorithm, which efficiently searches for an element within a given sequence of numbers. It takes a sorted sequence of integers as input from the user, along with the element to be searched for. The program then prints the index (0-based) where the element is located. If the element is not found, it outputs a message stating that the element is not present.

**Logic:**

1. Set low = 0, high = n - 1.
2. Compute mid = (low + high) / 2.
3. Compare arr[mid] with the target:
4. If arr[mid] == target → Found, return mid.
5. If arr[mid] < target → Search right (low = mid + 1).
6. If arr[mid] > target → Search left (high = mid - 1).
7. Repeat until low > high. If not found, return -1.

**Implementation:**

Data segment

.data

msg1: .asciiz "Number of elements you want to enter: \n"

msg2: .asciiz "Enter numbers in ascending order:\n"

msg3: .asciiz "Enter the value to search: \n"

msg4: .asciiz "Element not found\n"

msg5: .asciiz "Element has been found at index: "

* The messages that need to be printed from time to time

Taking user input

**.align** 2

nums: **.space** 4000  *# Reserving space for 1000 integers(4 bytes)*

**.text**

main: *# Taking the inputs*

      li $v0, 4

      la $a0, msg1

      syscall

      li $v0, 5

      syscall

      add $a1, $v0, $0

      li $v0, 4    *# Printing message before taking numbers*

      la $a0, msg2

      syscall

*# Filling the array with numbers*

      la $s1, nums                        *# Storing the base address of the array nums*

      li $t0, 0                   *# int i=0*

read:

     slt $t1, $t0, $a1

     beq $t1, $0, read\_done

     li $v0, 5

     syscall

     sll $t2, $t0, 2                      *# i\*4(offset)*

     add $t3, $s1, $t2                    *# Address of nums[i]*

     sw $v0, 0($t3)                       *# Store the number in the location*

     addi $t0, $t0, 1                     *# i++*

     j read

* Dynamically allocating memory for 1000 integers
* Asking the user to enter a sorted sequence of integers
* Initialising the iterator to 0. In each iteration, the code first computes the address where the number needs to be stored and then stores it in that location.
* Storing the number of elements entered in $a1 as its one of the arguments.

read\_done:

*# Taking input for the value to look for*

          li $v0, 4

          la $a0, msg3

          syscall

          li $v0, 5

          syscall

          add $a2, $v0, $0

          add $a3, $s1, $0

* Asking the user to enter the number to look for in the sequence.
* Storing the value to look for and the base address of the array in $a2 and $a3 respectively as they are the arguments as well for the procedure.

Calling the binary search procedure

*# Calling the binary search function*

          jal binarySearch

          add $s2, $v0, $0

* Calling the binary search procedure and storing the final result into $s2

Printing the result

*# Printing the final result*

          li $t0, -1

          beq $v0, $t0, notFound

          li $v0, 4

          la $a0, msg5

          syscall

          li $v0, 1

          add $a0, $s2, $0

          syscall

* Printing “not found” if the procedure returns -1. Else, printing the index(0-based) where the element was found

Binary Search Procedure

binarySearch:

             li $s0, 0              *# int low = 0*

             addi $s1, $a1, -1

             add $s1, $s1, $0       *# int high = n-1*

* Initialising low to 0 and high to (size of array-1).

whileLoop:

slt $t6, $s1, $s0

          bne $t6, $0, endLoop

          add $s2, $s0, $s1

          srl $s2, $s2, 1           *# mid = (low + high)/2*

          sll $s3, $s2, 2           *# 4\*mid*

          add $t2, $a3, $s3         *# base address + 4\*mid*

          lw $t3, 0($t2)            *# getting the nums[mid]*

          beq $t3, $a2, returnIdx   *# if nums[mid]==val we have to return mid*

          slt $t4, $t3, $a2

          beq $t4, $0, searchLeft   *# if nums[mid]>val*

          j searchRight             *# if nums[mid]<val*

* Until low<high, the program computes the middle index of the sequence.
* Then, the program loads the number stored in the corresponding address and checks if the number is equal to the number the user is looking for. If it is equal, then the the procedure immediately returns the value of the mid.
* If the number is greater than the number being searched, then the left side of mid is searched and vice-versa

searchRight:

            addi $s0, $s2, 1        *# low = mid+1*

            j whileLoop

searchLeft:

            addi $s1, $s2, -1       *# high = mid-1*

            j whileLoop

* Searching the left of mid : The code sets high = mid-1
* Searching the right of mid : The code sets low = mid+1

returnIdx: add $v0, $s2, $0

           jr $ra

* Returning the index where the number is found

endLoop:

        li $v0, -1

        jr $ra

* Returning -1 is the number isn’t found in the sequence

notFound:

         li $v0, 4

         la $a0, msg4

         syscall

*# Exit*

         li $v0, 10

         syscall

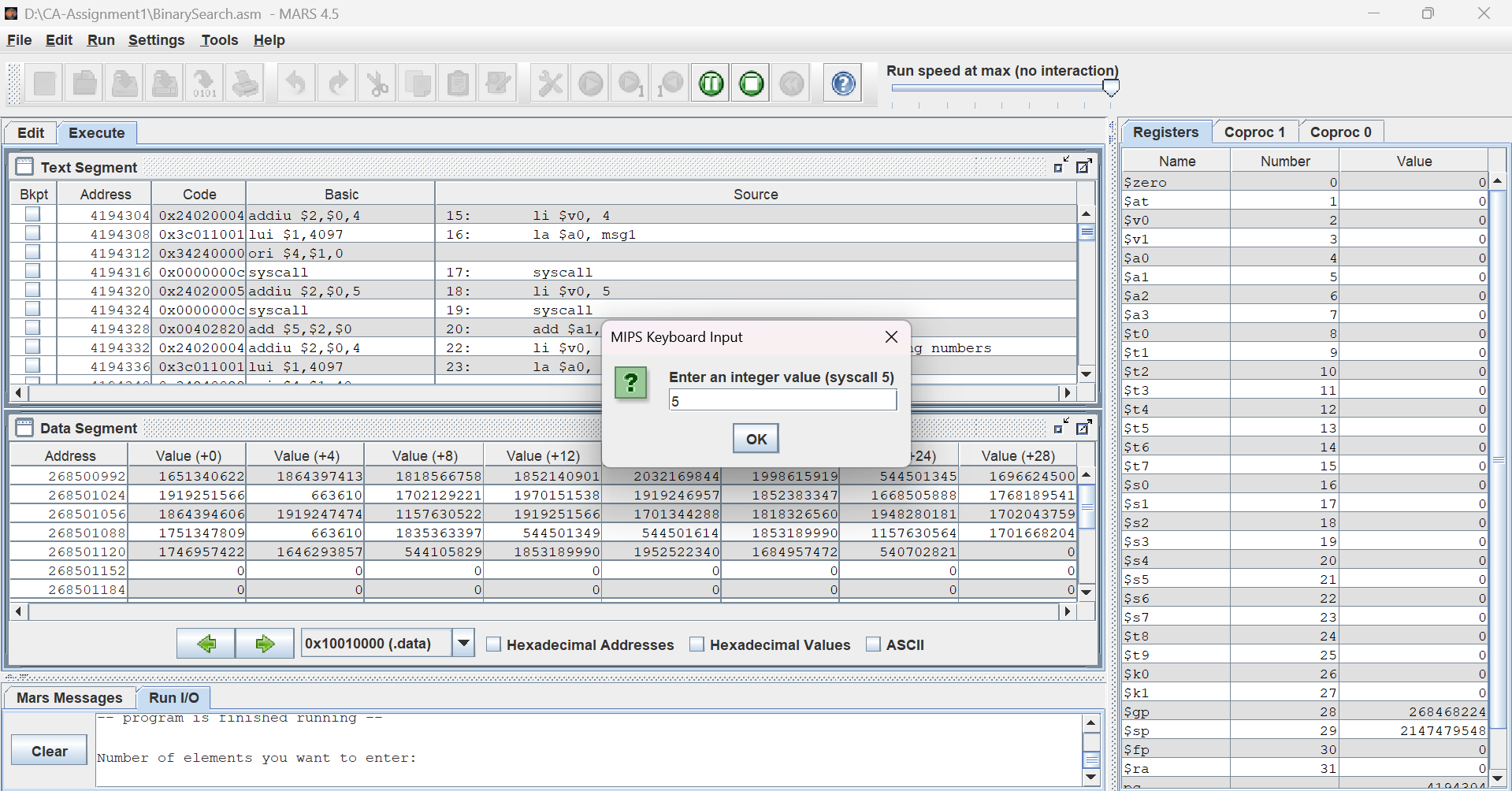
* Printing the “Not found” message and exiting from the program

**Observations:**

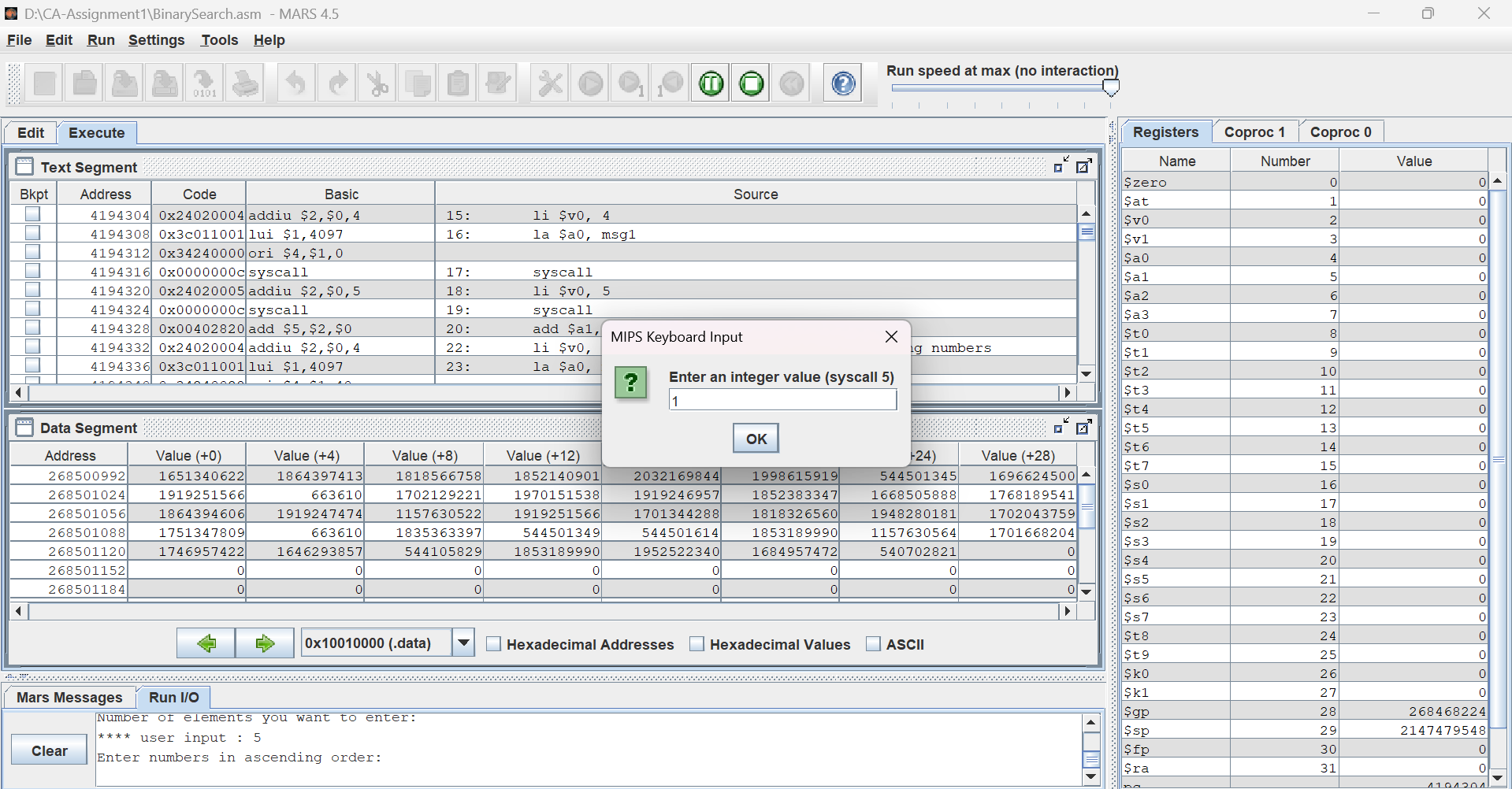
1. The program correctly reads user input and stores it in memory.
2. The binary search algorithm successfully finds elements in O(log n) time.
3. If the element is not found, the program displays "Element not found".
4. The program works efficiently with sorted input and correctly calculates array addresses.
5. The program does not crash and handles different inputs properly**.**

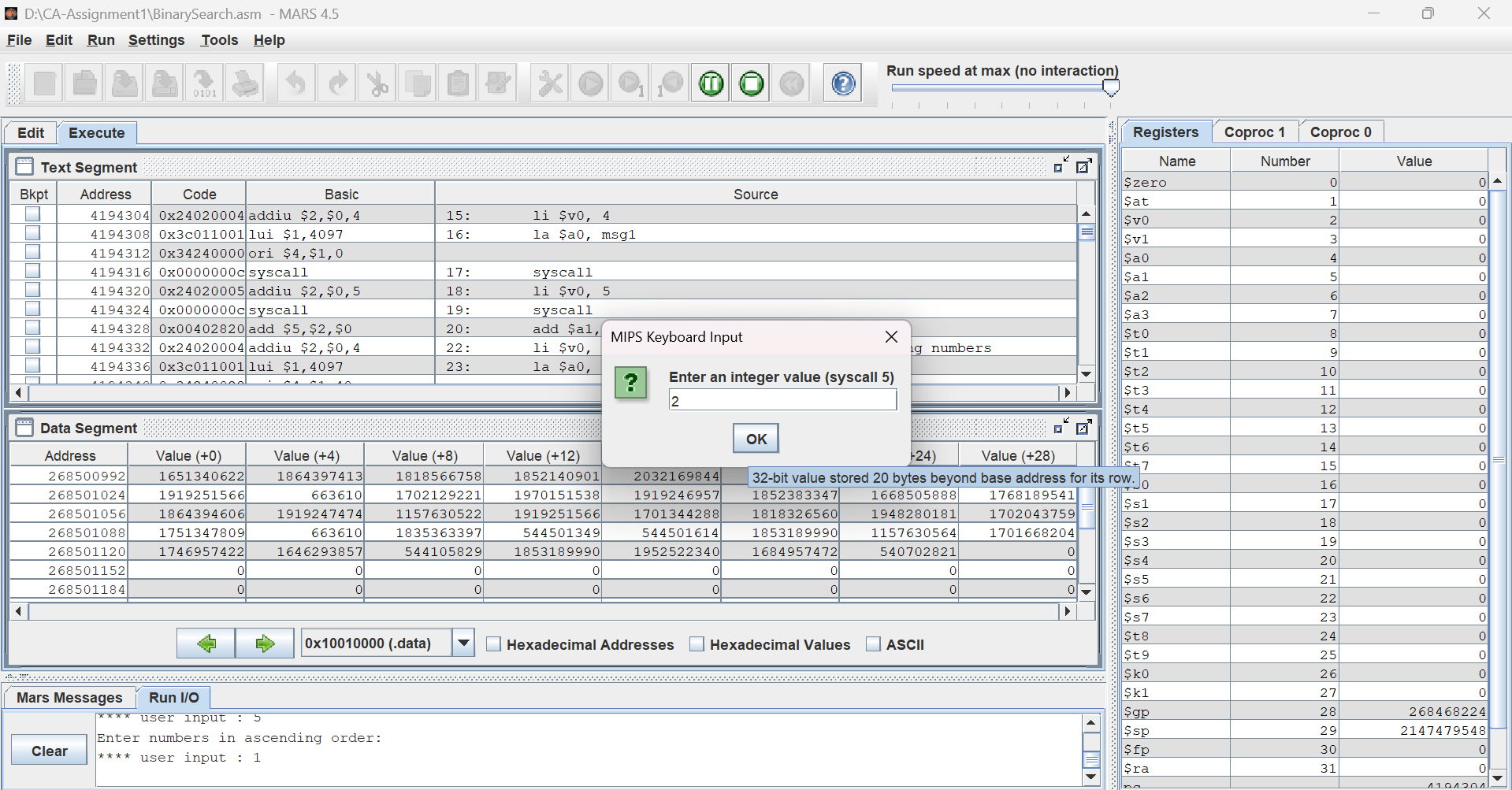
**Conclusion:**

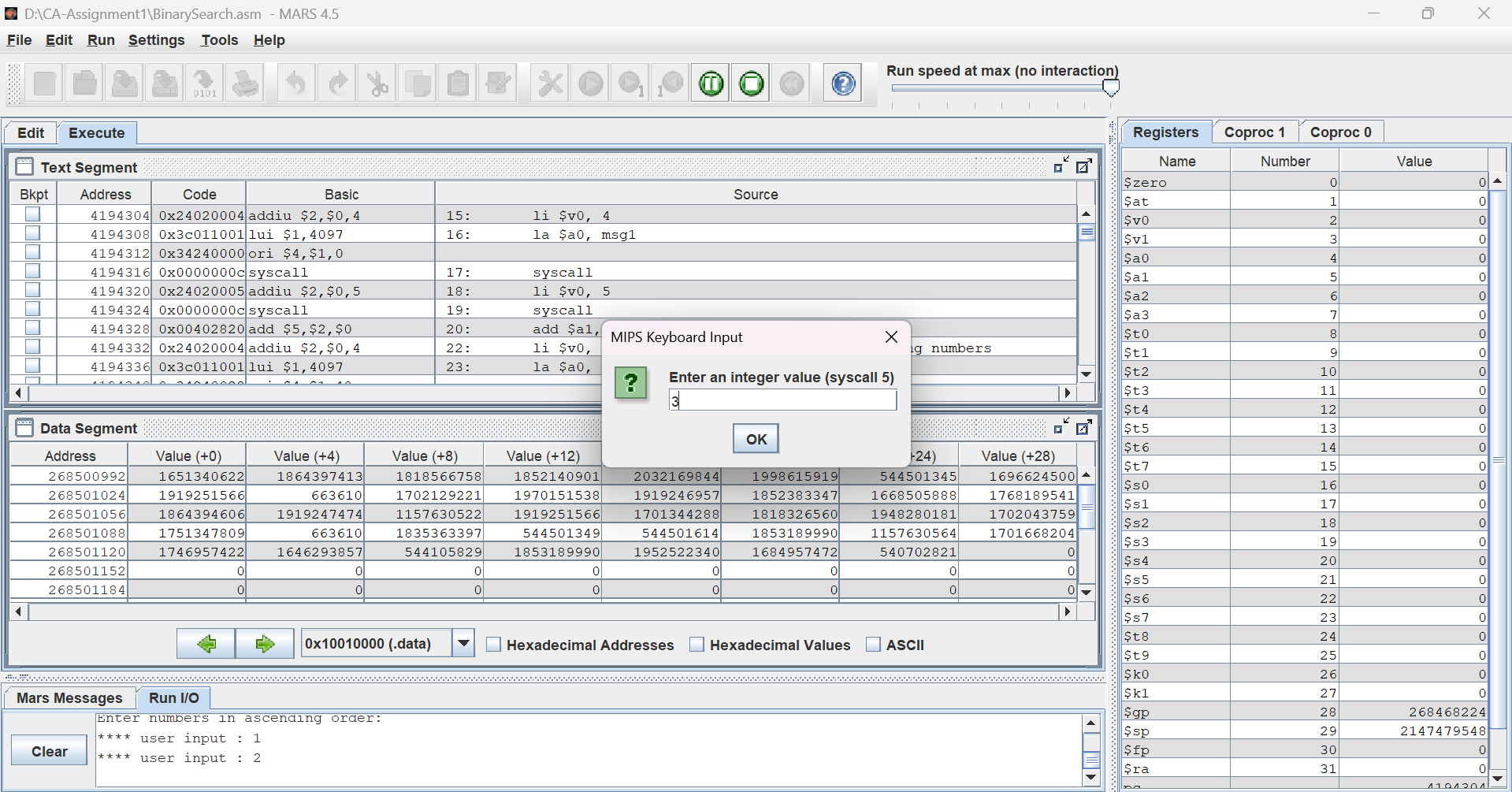
1. The iterative binary search algorithm was successfully implemented in MIPS assembly.
2. It efficiently searches for an element in a sorted array, reducing search time to O(log n).
3. The program correctly prints whether the element is found or not.
4. This project enhanced the understanding of assembly language, memory allocation, and logical operations.

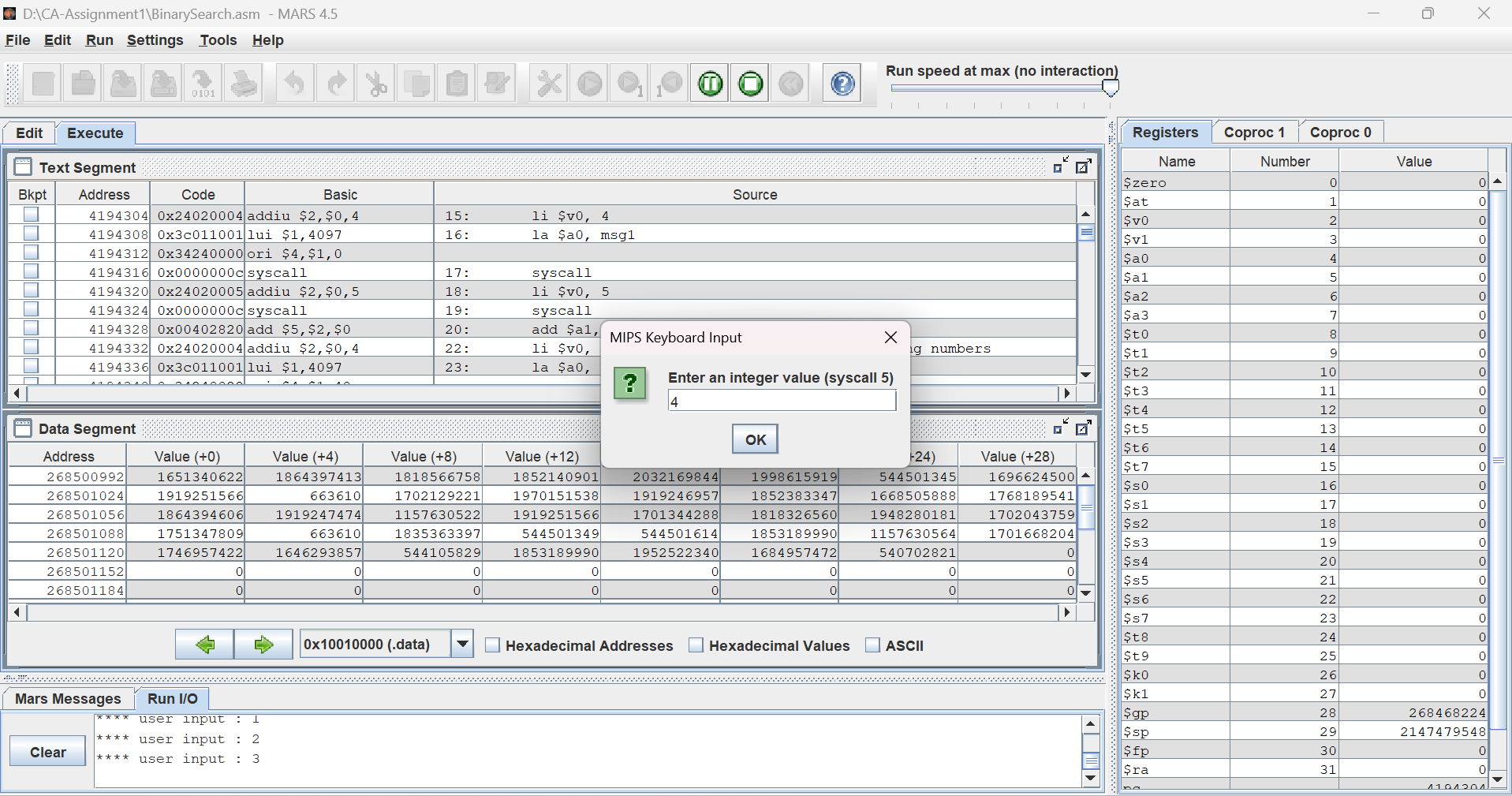


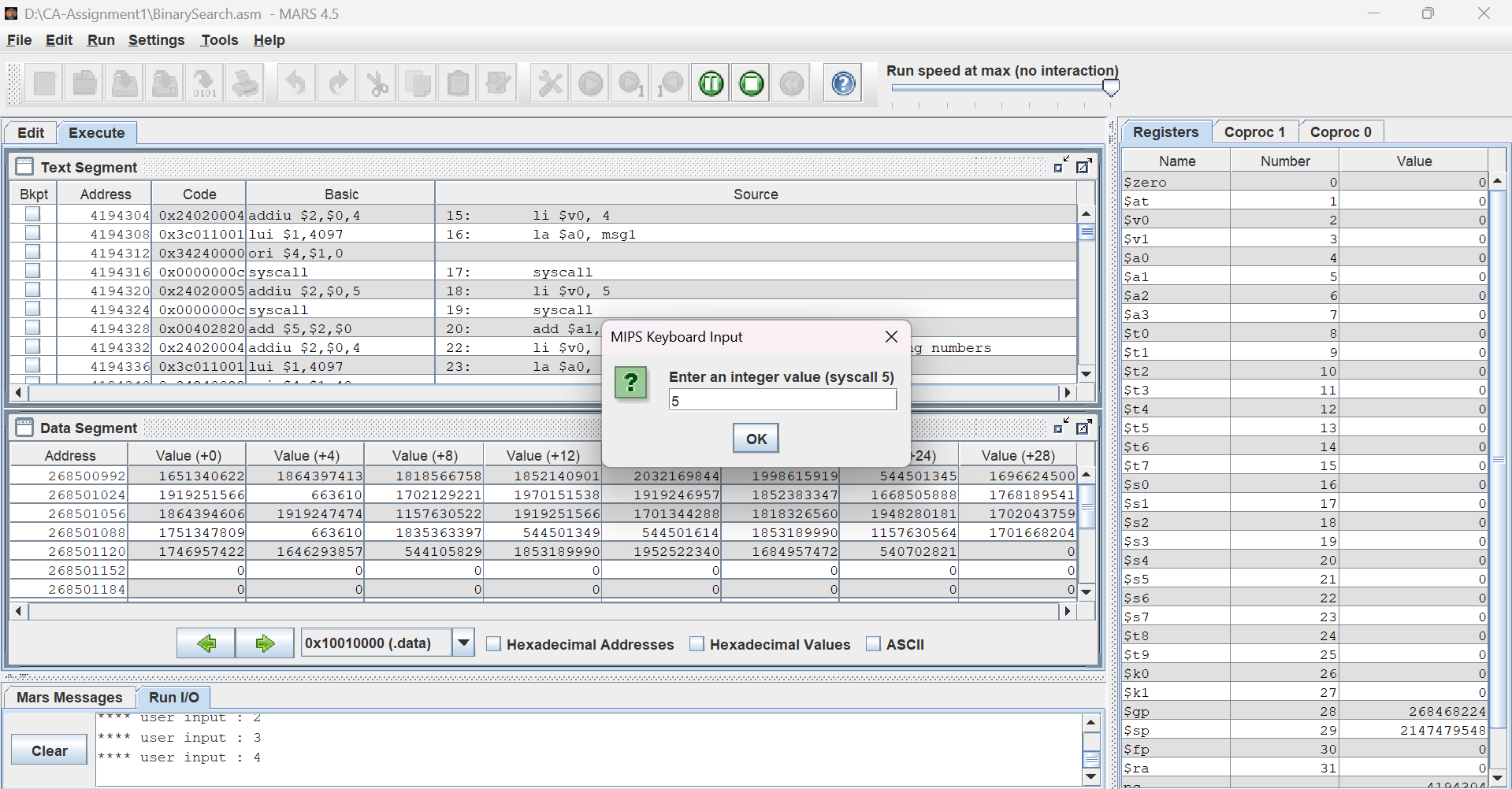
User input for number of integers in the sequence



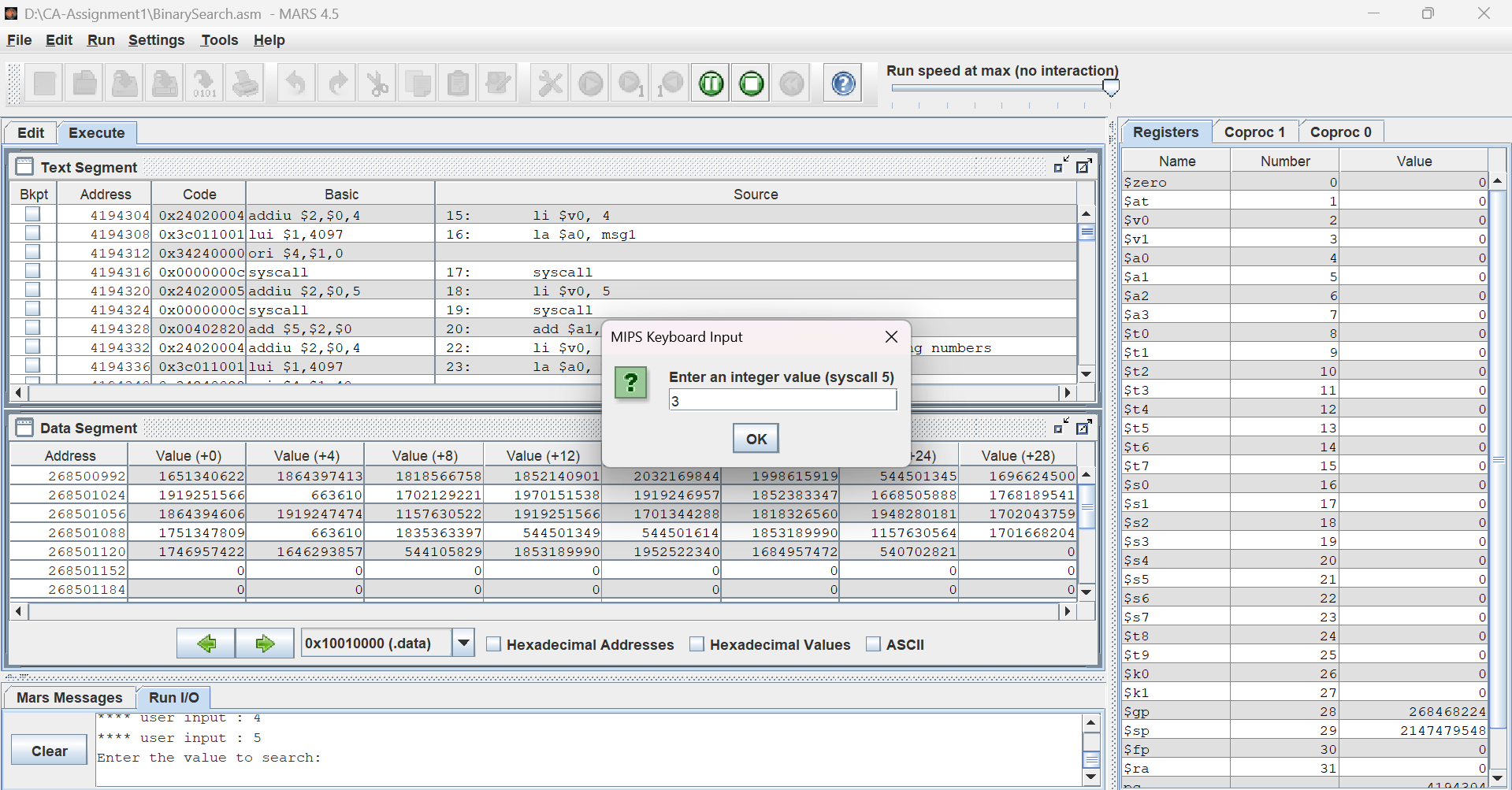




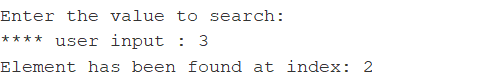




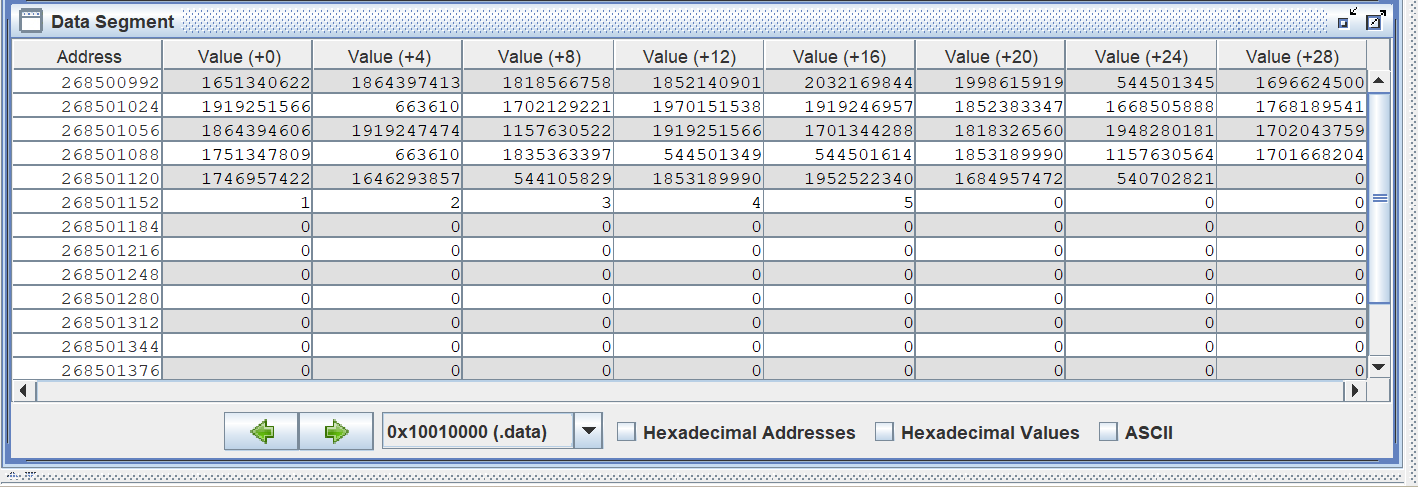
Entering 5 numbers of the integer sequence



Entering the value to look for

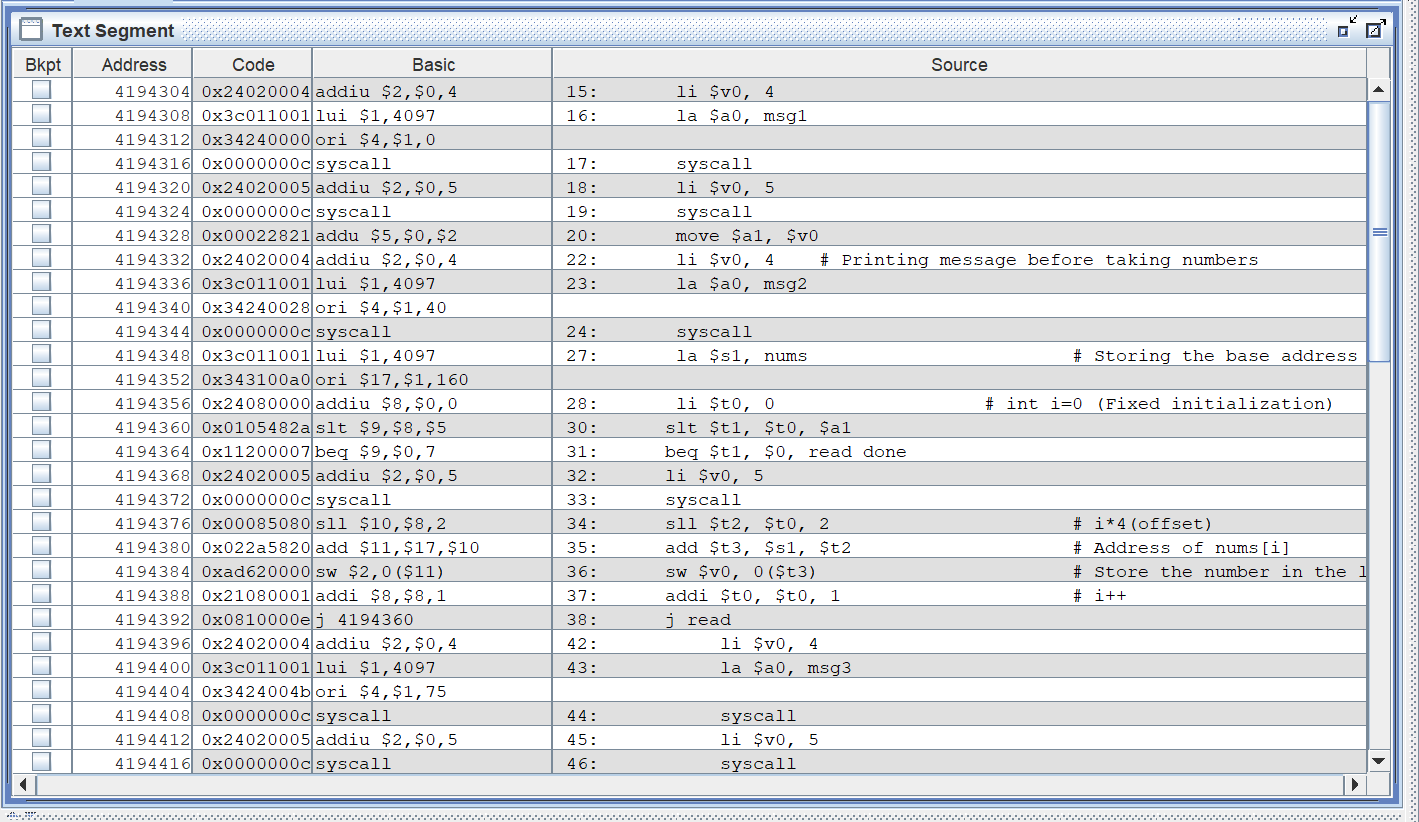


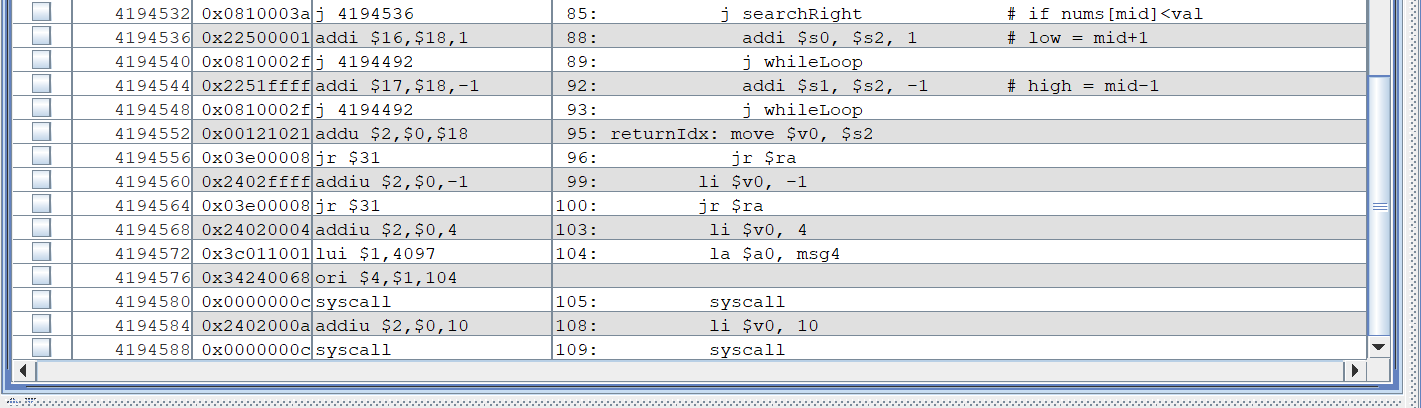
Execution results



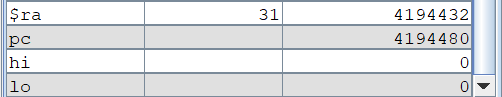
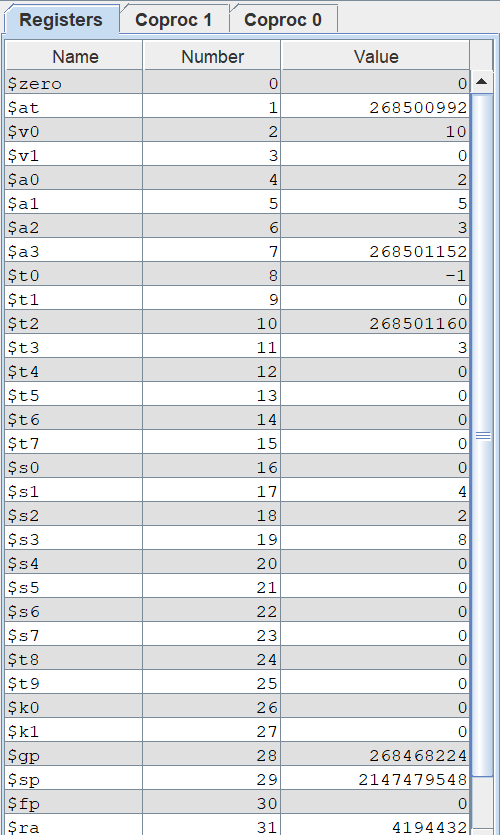


Data segment after execution of the program





Text segment after execution of the program



Register File

1. Python Assembler

The code is an assembler for MIPS that translates assembly language into machine code. It starts by converting MIPS register names (e.g., `$t0`, `$s1`) into their 5-bit binary equivalents using the `Register` class. The conversion is necessary because MIPS instructions represent registers as 5-bit values in machine code. Also, the `Instruction` class manages pseudo-instructions such as `ble` (branch if less than or equal) and `la` (load address) by specifying how they are expanded into multiple machine instructions. For instance, `ble` is expanded into `slt` (set less than) and `beq` (branch if equal), and `la` is expanded into `lui` (load upper immediate) and `ori` (OR immediate). These expansions are used to ensure that pseudo-instructions are translated into valid machine code correctly.

The heart of the assembler is the `Encoder` class, which offers methods to encode instructions into their corresponding binary forms: R-format for register operations (e.g., `add`, `sub`), I-format for immediate operations (e.g., `addi`, `lw`), and J-format for jump instructions (e.g., `j`, `jal`). The `Assembler` class then reads the assembly file, separating the `.data` and `.text` sections. It handles labels by storing their addresses in dictionaries (`data\_labels` for data segment labels and `text\_labels` for text segment labels) and instructions by expanding pseudo-instructions and encoding them into machine code. For example, the `li` (load immediate) instruction is encoded as an I-format instruction, whereas `ble` is expanded to `slt` and `beq`, each of which is encoded separately.

The `main` function finally manages the whole process. It reads in the assembly file, instantiates the `Assembler` class, and invokes methods to parse the file and encode the instructions. The generated machine code, stored as binary strings, is saved into an output file (`MachineCode.txt`). The assembler recognizes many instructions, ranging from arithmetic instructions (`add`, `sub`), memory-related instructions (`lw`, `sw`), control flow instructions (`j`, `beq`), and pseudo-instructions (`li`, `la`). By parsing systematically, expanding, and encoding instructions, the code successfully converts human-readable assembly language into binary machine code that can be executed on a MIPS processor.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*