TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

VIỆN CÔNG NGHỆ THÔNG TIN & TRUYỀN THÔNG

----- 🙠 🕮 🙢 -----

FINAL PROJECT REPORT

IT3280E – ASSEMBLY LANGUAGE AND COMPUTER ARCHITECTURE LAB

PROJECT	NAME	STUDENTID
8	Đỗ Hoàng Anh	20200012
4	Cho Su Wai	2023T070

Hà Nội, năm 2023

Project 8: RAID 5 disk simulation

A. Analysis of the execution:

- The program prints a prompt asking for input data from the user.
- The user enters a character string from the keyboard.
- The program checks whether the entered character string has a length that is a multiple of 8 by using the following method: it uses a length counter for the string, performs a right shift by 3 bits, and then performs a left shift by 3 bits. If the result matches the input data, then the number is divisible by 8.
    - If the entered string has a length that is a multiple of 8, the program continues running.
    - If not, the program prints a prompt asking the user to re-enter a different character string.
- The program use a hexadecimal function to convert binary to hexadecimal by:
    - Successively reverse the first 4 bits of the binary number to the bottom 8 times.
    - After each reverse operation of the 4 bits, perform the bitwise AND operation between the obtained binary number and 0xf. From this, calculate to determine the hexadecimal form of the binary input.
- Start the RAID5 Drive Simulation Process:
    - Parity stored on disk 3:
        - The first 4-byte data block of the string is stored on disk 1
        - The next 4-byte data block of the string is stored on disk 2.

  - Perform XOR operation between the corresponding values on each disk and store the result in the parity array.
  - Convert the obtained result from binary to hexadecimal format.
  - Save the result on disk 3.
  - Print the result on all drives.
  - Check if the end-of-string condition is met:
    - If yes, jump to the user prompt display section.
    - If not, continue the program.
- Parity stored on disk 2:
  - The next 4-byte data block of the string is stored on disk 1.
  - The next 4-byte data block of the string is stored on disk 3.
  - Perform XOR operation between the corresponding values on each disk and store the result in the parity array.
  - Convert the obtained result from binary to hexadecimal format.
  - Save the result on disk 2.
  - Print the result on all drives.
  - Check if the end-of-string condition is met:
    - If yes, jump to the user prompt display section.
    - If not, continue the program.
- Parity stored on disk 1:
  - The next 4-byte data block of the string is stored on disk 2.
  - The next 4-byte data block of the string is stored on disk 3.
  - Perform XOR operation between the corresponding values on each disk and store the result in the parity array.

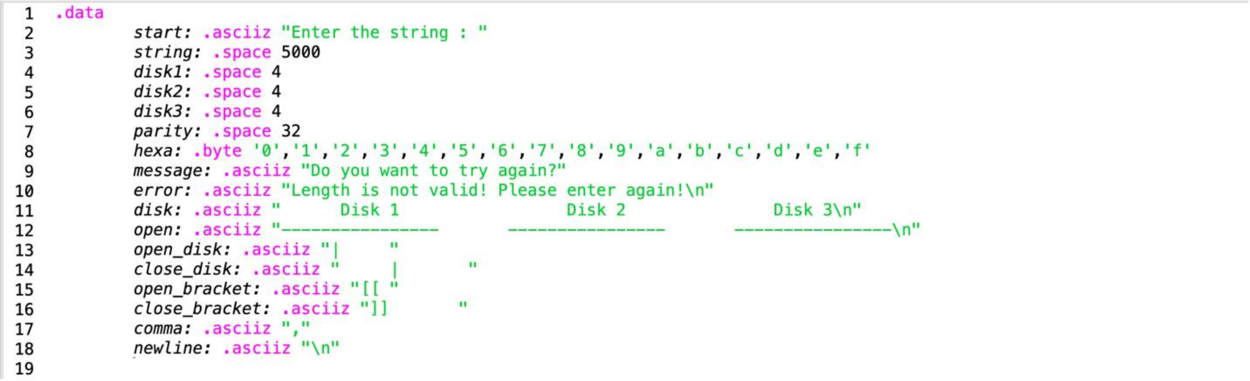
- Convert the obtained result from binary to hexadecimal format.
- Save the result on disk 1.
- Print the result on all drives.
- Check if the end-of-string condition is met:
  - If yes, jump to the user prompt display section.
  - If not, return to the parity saving step on disk 3 and start a new loop.

⇨ End the RAID5 Drive Simulation Process:

- The program displays a dialog box asking the user whether to try again with a different string.
  - If yes, the program resets the bytes in the original character string to 0 and returns to the section prompting the user to enter a character string.
  - If no, the program terminates.

B. Source code and algorithm:

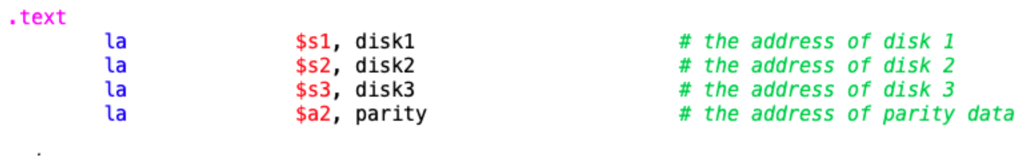
I. Initialize string and array:



```
.data
        start: .asciiz "Enter the string : "
        string: .space 5000
        disk1: .space 4
        disk2: .space 4
        disk3: .space 4
        parity: .space 32
        hexa: .byte '0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f'
        message: .asciiz "Do you want to try again?"
        error: .asciiz "Length is not valid! Please enter again!\n"
        disk: .asciiz "      Disk 1                 Disk 2               Disk 3\n"
        open: .asciiz "----------------       ----------------       ----------------\n"
        open_disk: .asciiz "|     "
        close_disk: .asciiz "     |       "
        open_bracket: .asciiz "[[ "
        close_bracket: .asciiz "]]       "
        comma: .asciiz ","
        newline: .asciiz "\n"

```

- Start: Prompt the user to enter data.
- Error: Prompt the user to re-enter when checking the invalid length of the string.
- Message: Display a dialog box asking the user if they want to retry entering the data.
- Disk, open, open_disk, close_disk, open_bracket, close_bracket, comma: Simulate RAID5 drives.
- Newline: Print a new line.
- Initialize a string array with 5000 bytes of memory to store the user-entered string.
- Initialize an array, disk1, corresponding to drive 1, with 4 bytes of memory.
- Initialize an array, disk2, corresponding to drive 2, with 4 bytes of memory.
- Initialize an array, disk3, corresponding to drive 3, with 4 bytes of memory.
- Initialize an array, parity, to store parity data with 32 bytes of memory.
- Initialize an array, hexa, to store hexadecimal values for conversion.

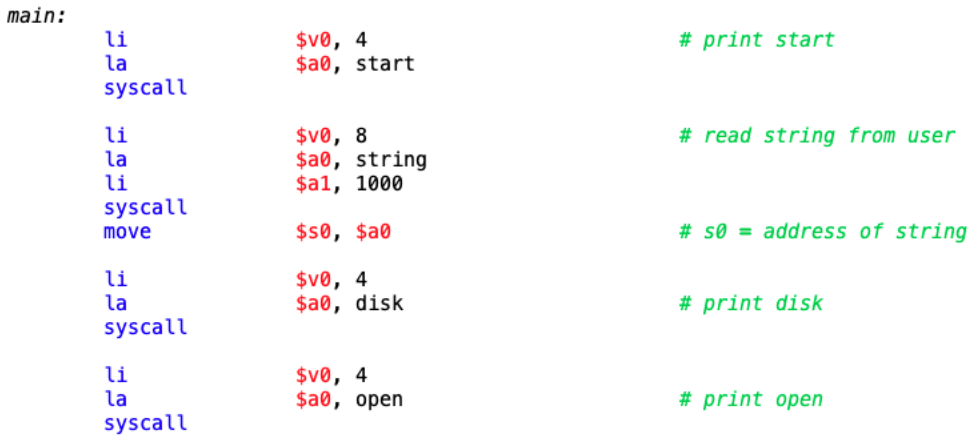
II. Load array address, print notification and entered input:



```
.text
        la              $s1, disk1                      # the address of disk 1
        la              $s2, disk2                      # the address of disk 2
        la              $s3, disk3                      # the address of disk 3
        la              $a2, parity                     # the address of parity data
```

- Entered address of array Disk 1 into register $s1
- Entered address of array Disk 2 into register $s2
- Entered address of array Disk 3 into register $s3

- Entered address of array parity into register $a2



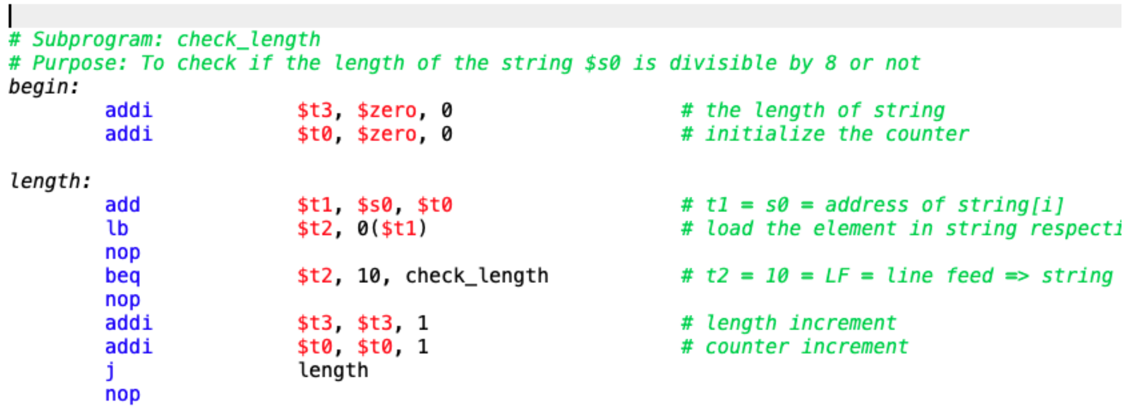
```
main:
        li              $v0, 4                          # print start
        la              $a0, start
        syscall

        li              $v0, 8                          # read string from user
        la              $a0, string
        li              $a1, 1000
        syscall
        move            $s0, $a0                        # s0 = address of string

        li              $v0, 4
        la              $a0, disk                       # print disk
        syscall

        li              $v0, 4
        la              $a0, open                       # print open
        syscall
```

Main function:

- Entered “start” string’s address into $a0 and print the notification.
- Entered the string array address into register $a0 and set the value of register $a1 to 1000, corresponding to the maximum number of characters the program will read.
- Use syscall 8 to read a string from the user.
- Save the address of the entered string into register $s0.
- Load the address of the string for disk output into register $a0 and print it
- Load the address of the string for open output into register $a0 and print it.

III. Determine the string length:

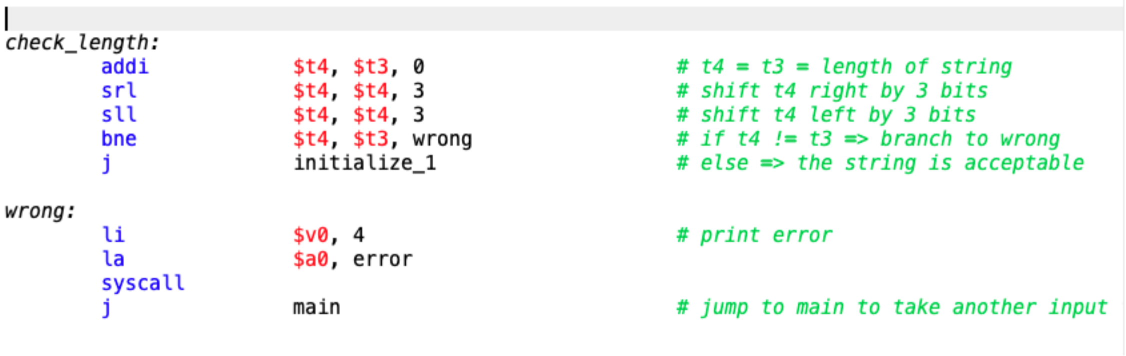
1. Get the length of string:



```
# Subprogram: check_length
# Purpose: To check if the length of the string $s0 is divisible by 8 or not
begin:
        addi            $t3, $zero, 0                   # the length of string
        addi            $t0, $zero, 0                   # initialize the counter

length:
        add             $t1, $s0, $t0                   # t1 = s0 = address of string[i]
        lb              $t2, 0($t1)                     # load the element in string respecti
        nop
        beq             $t2, 10, check_length           # t2 = 10 = LF = line feed => string
        nop
        addi            $t3, $t3, 1                     # length increment
        addi            $t0, $t0, 1                     # counter increment
        j               length
        nop
```

- Begin function:
    - Initialize the length of string equal 0 and store it in register $t3.
    - Initialize counter equal 0 and store it in register $t0**.**
- Length function:
    - Set the value of register $t1 to the address of the string stored in register $s0 incremented by the loop counter.
    - Load the content at the address in $t1 into register $t2.
    - Increase the length of the string by 1.
    - Increment the loop counter by 1.
    - Repeat the above steps until the value in $t2 equals 10, corresponding to LF (Line Feed) or "\n" in ASCII. This marks the end of the string.
    - Jump to the check_length section.

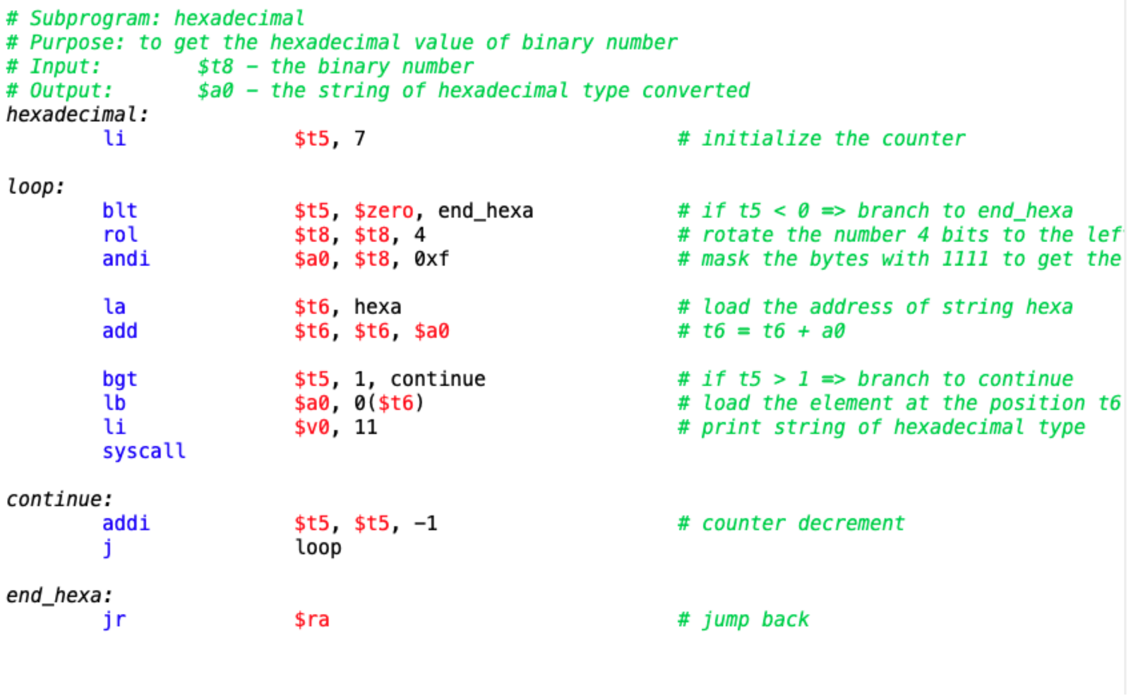
2. Check the conditional:



```
check_length:
        addi        $t4, $t3, 0                 # t4 = t3 = length of string
        srl         $t4, $t4, 3                 # shift t4 right by 3 bits
        sll         $t4, $t4, 3                 # shift t4 left by 3 bits
        bne         $t4, $t3, wrong             # if t4 != t3 => branch to wrong
        j           initialize_1                # else => the string is acceptable

wrong:
        li          $v0, 4                      # print error
        la          $a0, error
        syscall
        j           main                        # jump to main to take another input
```

- Check_length function:
  - Store the length of the string into register $t4 from $t3.
  - Perform a right shift by 3 bits with the value in register $t4.
  - Perform a left shift by 3 bits with the obtained value.
  - If the resulting value is equal to the length of the string stored in register $t3, then the length of the string is a multiple of 8. This condition is satisfied, and jump to the initialize_1 function.
  - If not equal, the string does not satisfy the condition. Jump to the wrong function.
- Function wrong:
  - Load the address of the error string into register $a0 and print the user prompt to re-enter the string.
  - Return to the main function to start over

IV. Conversion function from binary to hexadecimal



```
# Subprogram: hexadecimal
# Purpose: to get the hexadecimal value of binary number
# Input:        $t8 - the binary number
# Output:       $a0 - the string of hexadecimal type converted
hexadecimal:
        li              $t5, 7                          # initialize the counter

loop:
        blt             $t5, $zero, end_hexa            # if t5 < 0 => branch to end_hexa
        rol             $t8, $t8, 4                     # rotate the number 4 bits to the lef
        andi            $a0, $t8, 0xf                   # mask the bytes with 1111 to get the

        la              $t6, hexa                       # load the address of string hexa
        add             $t6, $t6, $a0                   # t6 = t6 + a0

        bgt             $t5, 1, continue                # if t5 > 1 => branch to continue
        lb              $a0, 0($t6)                     # load the element at the position t6
        li              $v0, 11                         # print string of hexadecimal type
        syscall

continue:
        addi            $t5, $t5, -1                    # counter decrement
        j               loop

end_hexa:
        jr              $ra                             # jump back
```

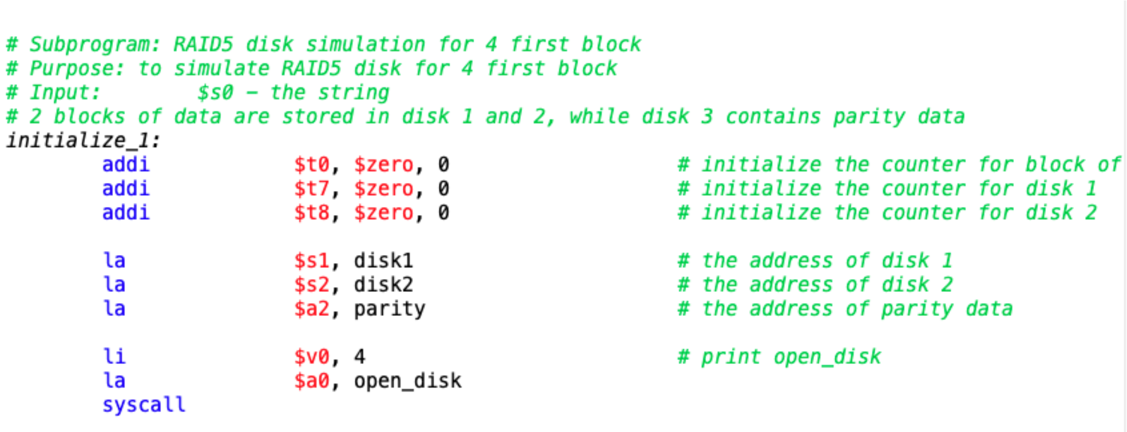
- Initialize the counter variable to 7 in register $t5
- Register $t8 contains the binary value to be converted
- Perform a left circular shift by 4 bits with the value stored in register $t8
- Perform an AND operation with 0xf (1111) to obtain the last 4 bits of the above sequence and store it in register $a0 => register $a0 then has a value from 0 to 15
- Load the address of the hex array into register $t6
- Add the value in register $a0 to $t6 => register $t6 contains the address of the $a0-th element in the hex array
- If the value of the counter in $t5 is greater than 1, jump to the continue function:
    - Subtract 1 from the counter value
    - Return to the loop and perform the above steps
- When the counter value is 1 or 0, load the value of the $a0-th element in the hex array into register $a0 and print it

- The loop ends when the counter value stored in register $t5 is less than 0. At that point, the program returns to the previous address.

V.

1. Disk 1 and 2 contain data of 4-byte block, disk 3 contain parity data

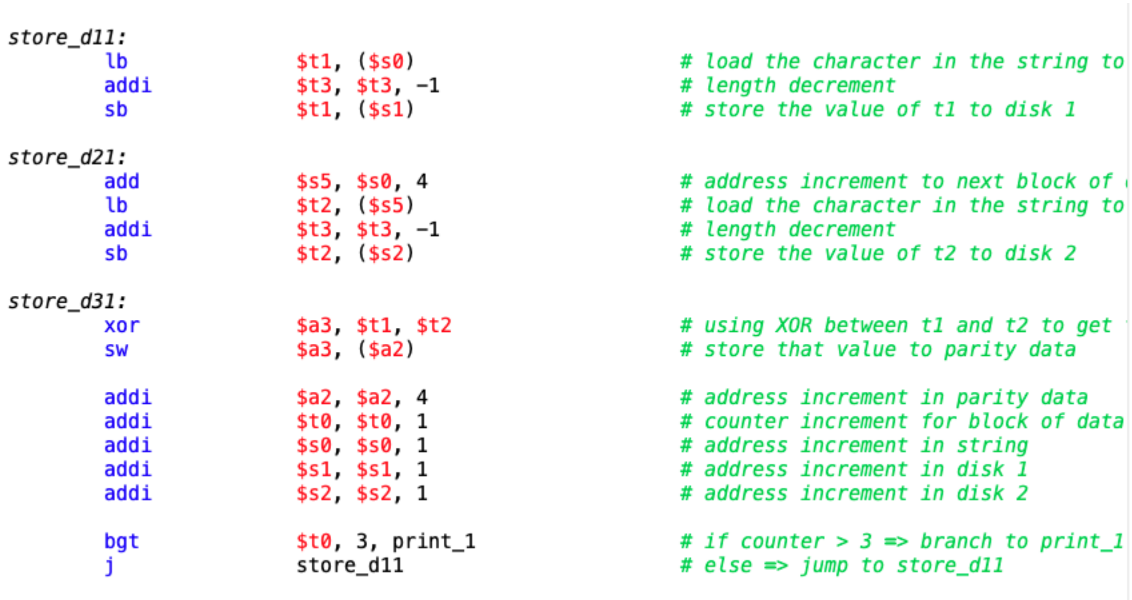
a. Initialize data:



```
# Subprogram: RAID5 disk simulation for 4 first block
# Purpose: to simulate RAID5 disk for 4 first block
# Input:        $s0 - the string
# 2 blocks of data are stored in disk 1 and 2, while disk 3 contains parity data
initialize_1:
        addi            $t0, $zero, 0               # initialize the counter for block of
        addi            $t7, $zero, 0               # initialize the counter for disk 1
        addi            $t8, $zero, 0               # initialize the counter for disk 2

        la              $s1, disk1                  # the address of disk 1
        la              $s2, disk2                  # the address of disk 2
        la              $a2, parity                 # the address of parity data

        li              $v0, 4                      # print open_disk
        la              $a0, open_disk
        syscall
```

- Initialize the data block counter to 0 in register $t0
- Initialize the block counter for disk 1 to 0 in register $t7
- Initialize the block counter for disk 2 to 0 in register $t8
- Load the address of the disk1 array into register $s1
- Load the address of the disk2 array into register $s2
- Load the address of the parity array into register $a2
- Load the address of the open_disk string and print it to the interface

b. Load and store data:

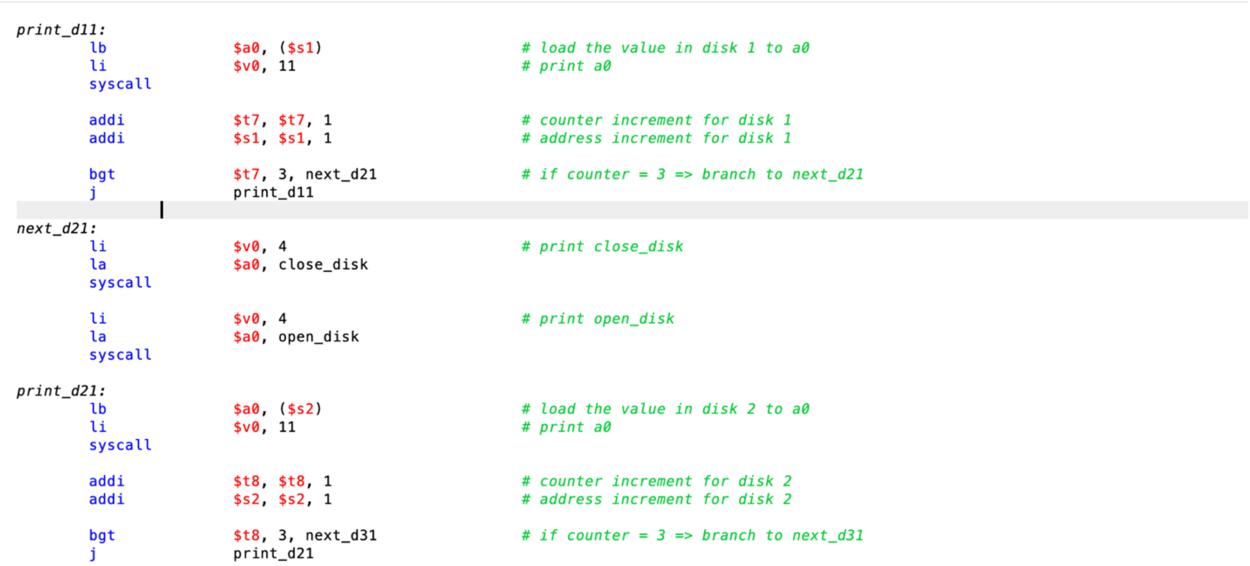


```
store_d11:
        lb              $t1, ($s0)                      # load the character in the string to
        addi            $t3, $t3, -1                    # length decrement
        sb              $t1, ($s1)                      # store the value of t1 to disk 1

store_d21:
        add             $s5, $s0, 4                     # address increment to next block of
        lb              $t2, ($s5)                      # load the character in the string to
        addi            $t3, $t3, -1                    # length decrement
        sb              $t2, ($s2)                      # store the value of t2 to disk 2

store_d31:
        xor             $a3, $t1, $t2                   # using XOR between t1 and t2 to get
        sw              $a3, ($a2)                      # store that value to parity data

        addi            $a2, $a2, 4                     # address increment in parity data
        addi            $t0, $t0, 1                     # counter increment for block of data
        addi            $s0, $s0, 1                     # address increment in string
        addi            $s1, $s1, 1                     # address increment in disk 1
        addi            $s2, $s2, 1                     # address increment in disk 2

        bgt             $t0, 3, print_1                 # if counter > 3 => branch to print_1
        j               store_d11                       # else => jump to store_d11
```

- Function store_d11:
  - Load the character at the address stored in $s0 in the input string into register $t1
  - Subtract 1 from the character count
  - Store the value in register $t1 into the array $s1 corresponding to disk 1
- Function store_d21:
  - Set the value of register $s5 to $s0 + 4 corresponding to the next data block in the string
  - Load the character at the address stored in $s5 in the input string into register $t2
  - Subtract 1 from the character count
  - Store the value in register $t2 into the array $s2 corresponding to disk 2
- Function store_d31:

- Use the XOR operator between the values in registers $t1 and $t2 and store the result in register $a3
- Store the value in register $a3 into the array $a2 corresponding to parity data
- Add 4 to the value at the address in the parity array stored in register $a2
- Add 1 to the data block counter stored in register $t0
- Add 1 to the address in the character string stored in register $s0
- Add 1 to the address in disk 1 stored in register $s1
- Add 1 to the address in disk 2 stored in register $s2
- Repeat the above steps until the counter is equal to 3 => end of the loop and jump to the print_1 function

c. Calculate and print data for each disk



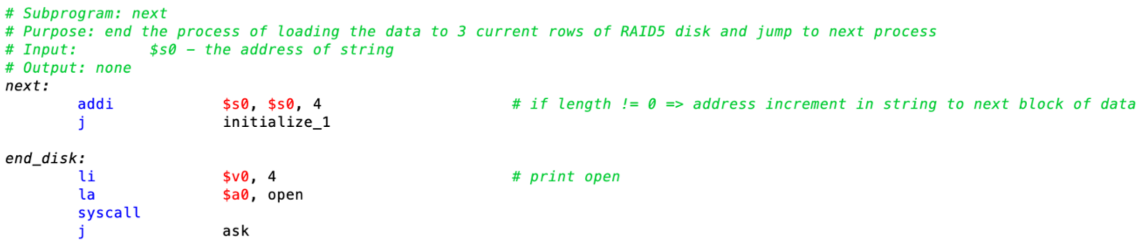
```
print_d11:
        lb              $a0, ($s1)                      # load the value in disk 1 to a0
        li              $v0, 11                         # print a0
        syscall

        addi            $t7, $t7, 1                     # counter increment for disk 1
        addi            $s1, $s1, 1                     # address increment for disk 1

        bgt             $t7, 3, next_d21                # if counter = 3 => branch to next_d21
        j               print_d11

next_d21:
        li              $v0, 4                          # print close_disk
        la              $a0, close_disk
        syscall

        li              $v0, 4                          # print open_disk
        la              $a0, open_disk
        syscall

print_d21:
        lb              $a0, ($s2)                      # load the value in disk 2 to a0
        li              $v0, 11                         # print a0
        syscall

        addi            $t8, $t8, 1                     # counter increment for disk 2
        addi            $s2, $s2, 1                     # address increment for disk 2

        bgt             $t8, 3, next_d31                # if counter = 3 => branch to next_d31
        j               print_d21
```

- Function print_1:
    - Load the address of array disk1 into register $s1
    - Load the address of array disk2 into register $s2

- Function print_d11:
    - Load a character from the address in disk1 at $s1 into register $a0
    - Use syscall 11 to print the character
    - Increment the counter stored in register $t7 by 1
    - Increment the address value in disk1 at $s1 by 1
    - Repeat until counter = 3 -> End loop and jump to function next_d21
- Function next_d21:
    - Load the address of the close_disk string into register $a0 and print it
    - Load the address of the open_disk string into register $a0 and print it
- Repeat again until function end_1
    - Load the character at the address in the parity array at $a2 into register $t8
    - Jump to the hexadecimal function to convert the value to hexadecimal
    - After conversion, load the address of close_bracket into register $a0 and print it
    - Load the address of newline into register $a0 and print a new line
    - Check conditions:
        - If the length of the character string is 0, jump to end_disk
        - If the string is not terminated, perform similar steps to store data blocks into disk 1, 3, and store parity data into disk 2

⇨ Do the same with disk 1 and disk 2 contain parity data

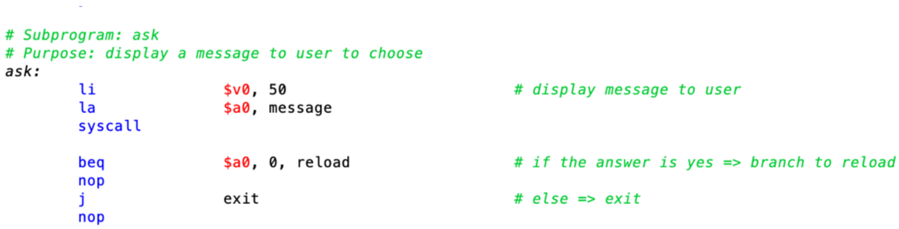
VI. Check String Conditions Source Code:



```
# Subprogram: next
# Purpose: end the process of loading the data to 3 current rows of RAID5 disk and jump to next process
# Input:        $s0 - the address of string
# Output: none
next:
        addi            $s0, $s0, 4                     # if length != 0 => address increment in string to next block of data
        j               initialize_1

end_disk:
        li              $v0, 4                          # print open
        la              $a0, open
        syscall
        j               ask
```

- Function next:
  - ○ If the string is not terminated, increment the address in the character string by 4 to point to the next data block.
  - ○ Return to the initialize_1 function to start a new loop.
- Function end_disk:
  - ○ This function is used when the string has ended.
  - ○ Load the address of the open string into register $a0 and print it.
  - ○ Jump to the ask function.

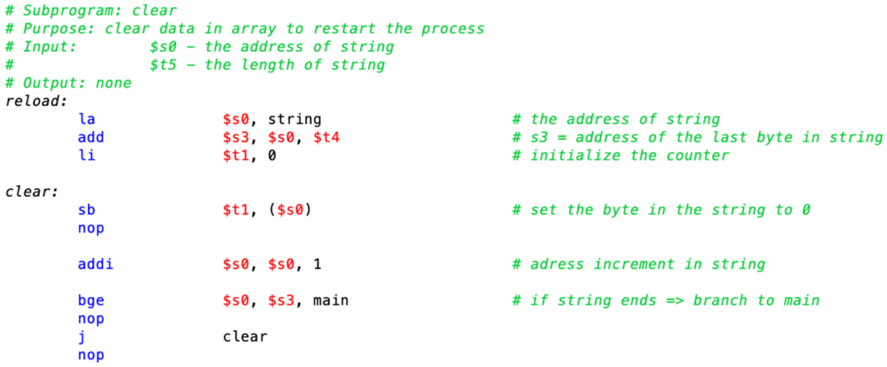
VII. Display User Prompt Dialog Source Code:



```
# Subprogram: ask
# Purpose: display a message to user to choose
ask:
        li              $v0, 50                         # display message to user
        la              $a0, message
        syscall

        beq             $a0, 0, reload                  # if the answer is yes => branch to reload
        nop
        j               exit                            # else => exit
        nop
```

- Function ask:
- Load the address of the message string into register $a0.
- Use syscall 50 to display a dialog asking the user if they want to try again.
  - ○ If yes, jump to the reload function.
  - ○ If no, jump to the exit function.

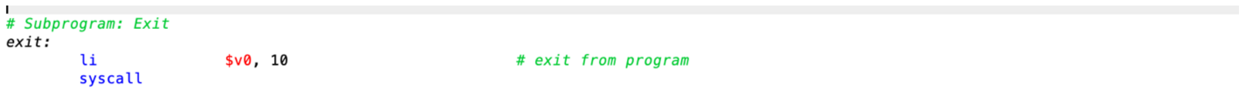
VIII. Reset String Data Source Code:



```
# Subprogram: clear
# Purpose: clear data in array to restart the process
# Input:        $s0 - the address of string
#               $t5 - the length of string
# Output: none
reload:
        la              $s0, string                     # the address of string
        add             $s3, $s0, $t4                   # s3 = address of the last byte in string
        li              $t1, 0                          # initialize the counter

clear:
        sb              $t1, ($s0)                      # set the byte in the string to 0
        nop

        addi            $s0, $s0, 1                     # adress increment in string

        bge             $s0, $s3, main                  # if string ends => branch to main
        nop
        j               clear
        nop
```

- Function reload:
  - Load the initial character string address into register $s0.
  - Set the value of register $s3 = $s0 + $t4 => $s3 contains the address of the last character in the string.
  - Initialize the counter variable to 0 in register $t1.
- Function clear:
  - Store the value 0 at the address in $s0 in the character string.
  - Increment the address value in the character string at $s0 by 1.
  - Repeat the above steps until the value of $s0 is equal to $s3.End the loop and return to the main function.

IX. End of Program Source Code:

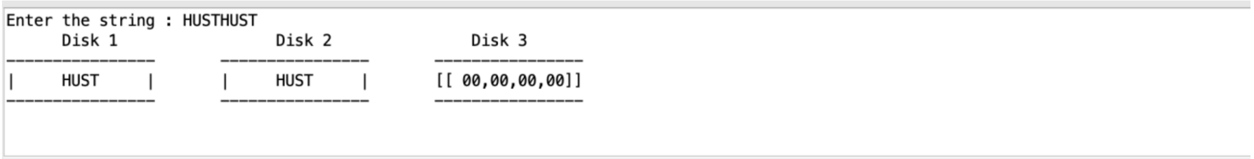


```
# Subprogram: Exit
exit:
        li              $v0, 10                         # exit from program
        syscall
```

Function end_program:

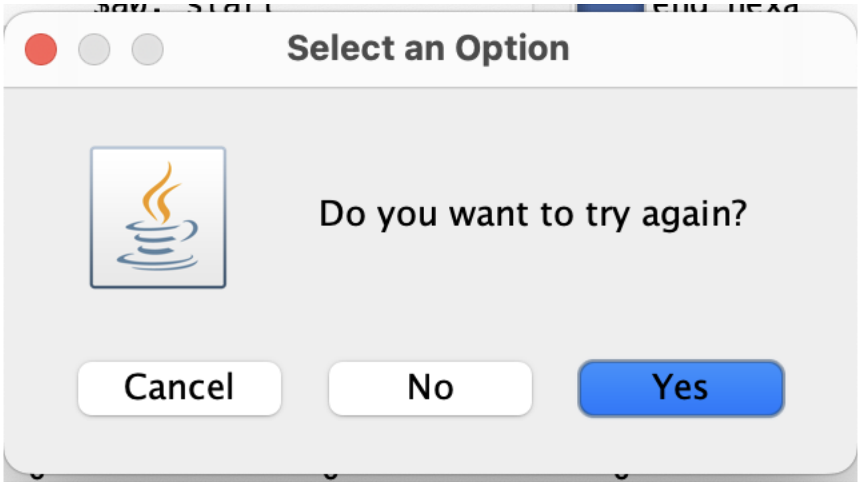
C. Simulation Results

- Input with string 8:



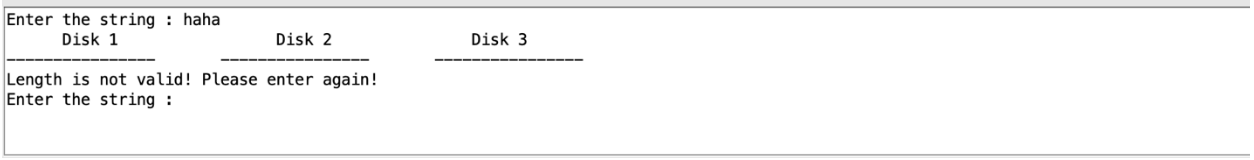
```
Enter the string : HUSTHUST
      Disk 1                Disk 2               Disk 3
----------------       ----------------     ----------------
|     HUST      |      |     HUST      |     [[ 00,00,00,00]]
----------------       ----------------     ----------------
```

- Success dialog



Do you want to try again?

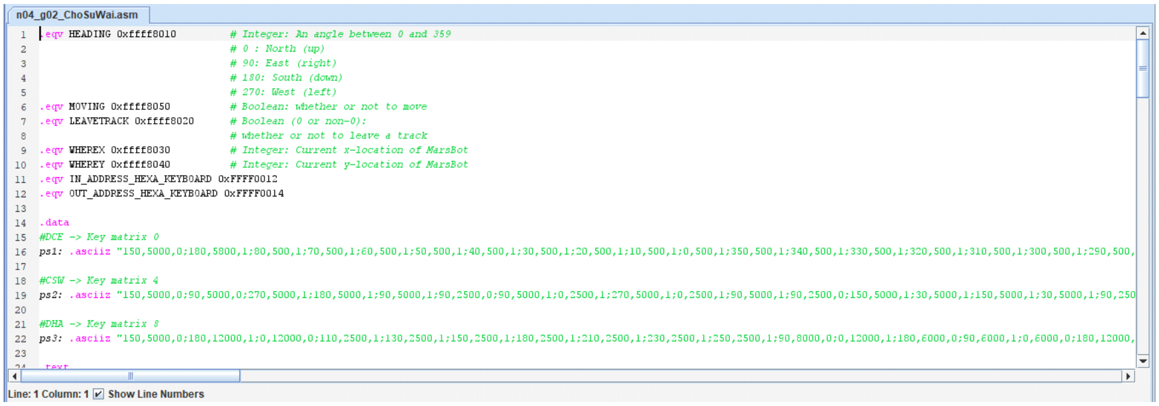
- Input with string 4:



```
Enter the string : haha
      Disk 1                Disk 2               Disk 3
----------------       ----------------     ----------------
Length is not valid! Please enter again!
Enter the string :
```

Project 4: Postscript CNC Marsbot

You can run this program with Mars Bot and Digital Lab Sim.



```
.eqv HEADING 0xffff8010          # Integer: An angle between 0 and 359
                                 # 0 : North (up)
                                 # 90: East (right)
                                 # 180: South (down)
                                 # 270: West (left)
.eqv MOVING 0xffff8050           # Boolean: whether or not to move
.eqv LEAVETRACK 0xffff8020       # Boolean (0 or non-0):
                                 # whether or not to leave a track
.eqv WHEREX 0xffff8030           # Integer: Current x-location of MarsBot
.eqv WHEREY 0xffff8040           # Integer: Current y-location of MarsBot
.eqv IN_ADDRESS_HEXA_KEYBOARD 0xFFFF0012
.eqv OUT_ADDRESS_HEXA_KEYBOARD 0xFFFF0014

.data
#DCE -> Key matrix 0
ps1: .asciiz "150,5000,0;180,5800,1;80,500,1;70,500,1;60,500,1;50,500,1;40,500,1;30,500,1;20,500,1;10,500,1;0,500,1;350,500,1;340,500,1;330,500,1;320,500,1;310,500,1;300,500,1;290,500,

#CSW -> Key matrix 4
ps2: .asciiz "150,5000,0;90,5000,0;270,5000,1;180,5000,1;90,5000,1;90,2500,0;90,5000,1;0,2500,1;270,5000,1;0,2500,1;90,5000,1;90,2500,0;150,5000,1;30,5000,1;150,5000,1;30,5000,1;90,250

#DHA -> Key matrix 8
ps3: .asciiz "150,5000,0;180,12000,1;0,12000,0;110,2500,1;130,2500,1;150,2500,1;180,2500,1;210,2500,1;230,2500,1;250,2500,1;90,8000,0;0,12000,1;180,6000,0;90,6000,1;0,6000,0;180,12000,

.text
```

This project includes 3 postscripts : DCE, CSW, DHA.

#DCE -> Key matrix 0

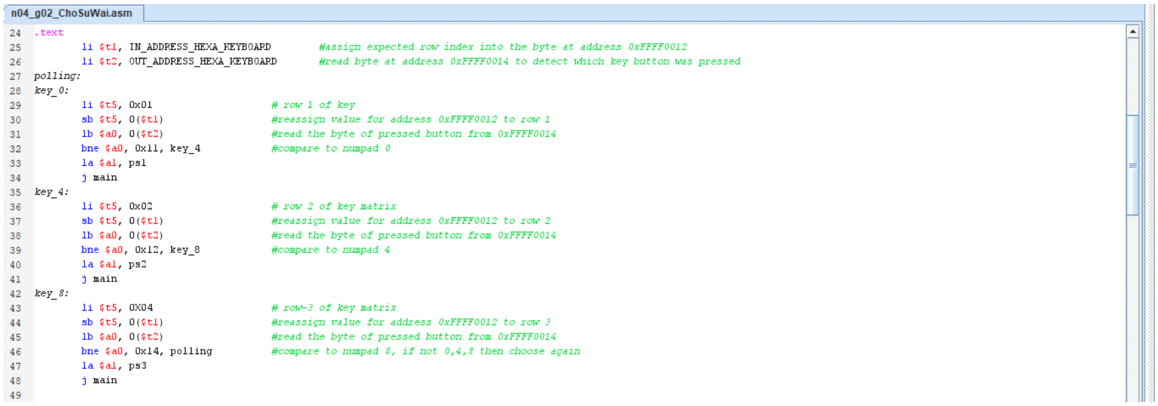
ps1: .asciiz
"150,5000,0;180,5800,1;80,500,1;70,500,1;60,500,1;50,500,1;40,500,1;30,500,1;20,500,1;10,500,1;0,500,1;350,500,1;340,500,1;330,500,1;320,500,1;310,500,1;300,500,1;290,500,1;280,490,1;90,9000,0;180,1000,0;310,400,1;300,400,1;290,400,1;280,400,1;270,800,1;260,400,1;250,400,1;240,400,1;230,400,1;220,400,1;210,400,1;200,400,1;190,400,1;180,800,1;170,400,1;160,400,1;150,400,1;140,400,1;130,400,1;120,400,1;110,400,1;100,400,1;90,800,1;80,400,1;70,400,1;60,400,1;50,400,1;120,3000,0;0,5800,1;90,2000,1;180,2900,0;270,2000,1;180,2900,0;90,2000,1;;90,3000,0;"

#CSW -> Key matrix 4

ps2: .asciiz
"150,5000,0;90,5000,0;270,5000,1;180,5000,1;90,5000,1;90,2500,0;90,5000,1;0,2500,1;270,5000,1;0,2500,1;90,5000,1;90,2500,0;150,5000,1;30,5000,1;150,5000,1;30,5000,1;90,2500,0;"

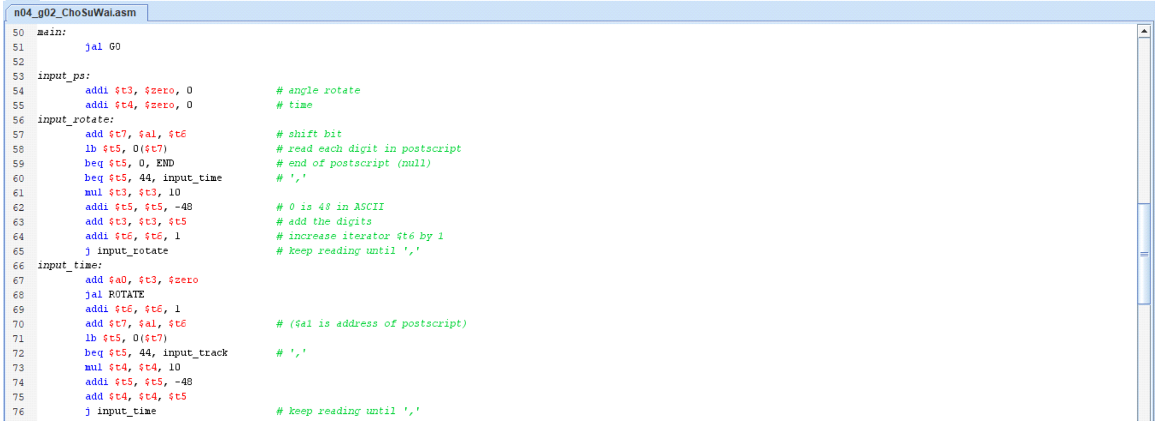
#DHA -> Key matrix 8

ps3: .asciiz
"150,5000,0;180,12000,1;0,12000,0;110,2500,1;130,2500,1;150,2500,1;180,2500,1;210,2500,1;230,2500,1;250,2500,1;90,8000,0;0,12000,1;180,6000,0;90,6000,1;0,6000,0;180,12000,1;90,2000,0;30,12000,1;150,12000,1;330,6000,0;270,6000,1;90,10000,0;"



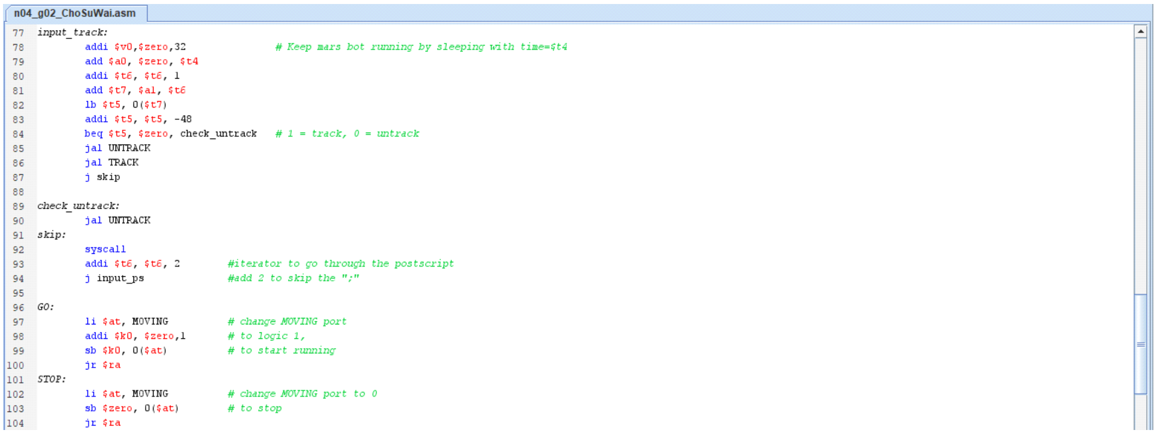
```
.text
        li $t1, IN_ADDRESS_HEXA_KEYBOARD        #assign expected row index into the byte at address 0xFFFF0012
        li $t2, OUT_ADDRESS_HEXA_KEYBOARD       #read byte at address 0xFFFF0014 to detect which key button was pressed
polling:
key_0:
        li $t5, 0x01                    # row 1 of key
        sb $t5, 0($t1)                  #reassign value for address 0xFFFF0012 to row 1
        lb $a0, 0($t2)                  #read the byte of pressed button from 0xFFFF0014
        bne $a0, 0x11, key_4            #compare to numpad 0
        la $a1, ps1
        j main
key_4:
        li $t5, 0x02                    # row 2 of key matrix
        sb $t5, 0($t1)                  #reassign value for address 0xFFFF0012 to row 2
        lb $a0, 0($t2)                  #read the byte of pressed button from 0xFFFF0014
        bne $a0, 0x12, key_8            #compare to numpad 4
        la $a1, ps2
        j main
key_8:
        li $t5, 0X04                    # row-3 of key matrix
        sb $t5, 0($t1)                  #reassign value for address 0xFFFF0012 to row 3
        lb $a0, 0($t2)                  #read the byte of pressed button from 0xFFFF0014
        bne $a0, 0x14, polling          #compare to numpad 8, if not 0,4,8 then choose again
        la $a1, ps3
        j main

```

In this one, this program checks which key user clicked and go to main.



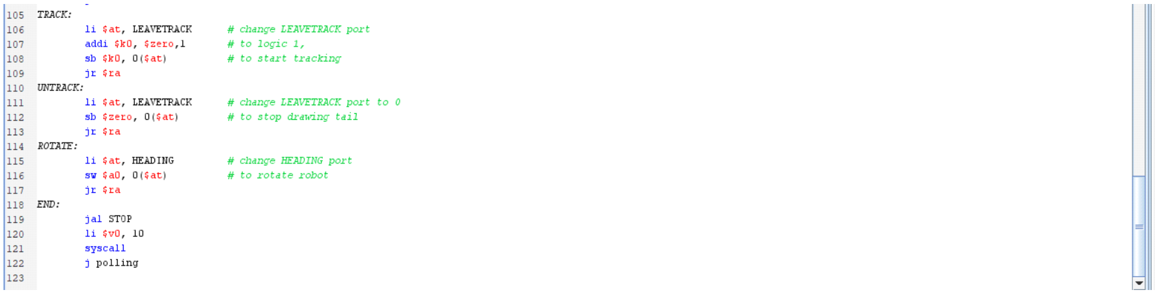
```
main:
        jal GO

input_ps:
        addi $t3, $zero, 0              # angle rotate
        addi $t4, $zero, 0              # time
input_rotate:
        add $t7, $a1, $t6               # shift bit
        lb $t5, 0($t7)                  # read each digit in postscript
        beq $t5, 0, END                 # end of postscript (null)
        beq $t5, 44, input_time         # ','
        mul $t3, $t3, 10
        addi $t5, $t5, -48              # 0 is 48 in ASCII
        add $t3, $t3, $t5               # add the digits
        addi $t6, $t6, 1                # increase iterator $t6 by 1
        j input_rotate                  # keep reading until ','
input_time:
        add $a0, $t3, $zero
        jal ROTATE
        addi $t6, $t6, 1
        add $t7, $a1, $t6               # ($a1 is address of postscript)
        lb $t5, 0($t7)
        beq $t5, 44, input_track        # ','
        mul $t4, $t4, 10
        addi $t5, $t5, -48
        add $t4, $t4, $t5
        j input_time                    # keep reading until ','
```

In this one, iterates the postscript including angle, duration, cut/uncut.



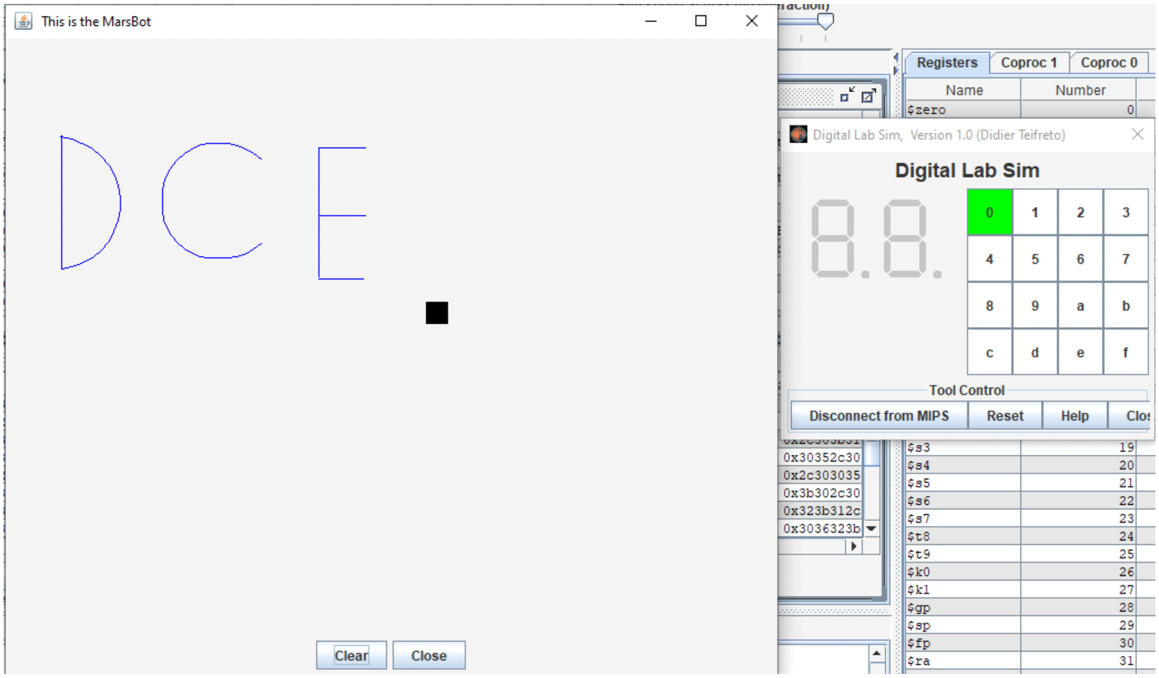
```
input_track:
        addi $v0,$zero,32               # Keep mars bot running by sleeping with time=$t4
        add $a0, $zero, $t4
        addi $t6, $t6, 1
        add $t7, $a1, $t6
        lb $t5, 0($t7)
        addi $t5, $t5, -48
        beq $t5, $zero, check_untrack   # 1 = track, 0 = untrack
        jal UNTRACK
        jal TRACK
        j skip

check_untrack:
        jal UNTRACK
skip:
        syscall
        addi $t6, $t6, 2        #iterator to go through the postscript
        j input_ps              #add 2 to skip the ";"

GO:
        li $at, MOVING          # change MOVING port
        addi $k0, $zero,1       # to logic 1,
        sb $k0, 0($at)          # to start running
        jr $ra
STOP:
        li $at, MOVING          # change MOVING port to 0
        sb $zero, 0($at)        # to stop
        jr $ra
```

In skip, add 2 to skip ";". And in GO and STOP labels, 1 is moving and 0 is to stop.



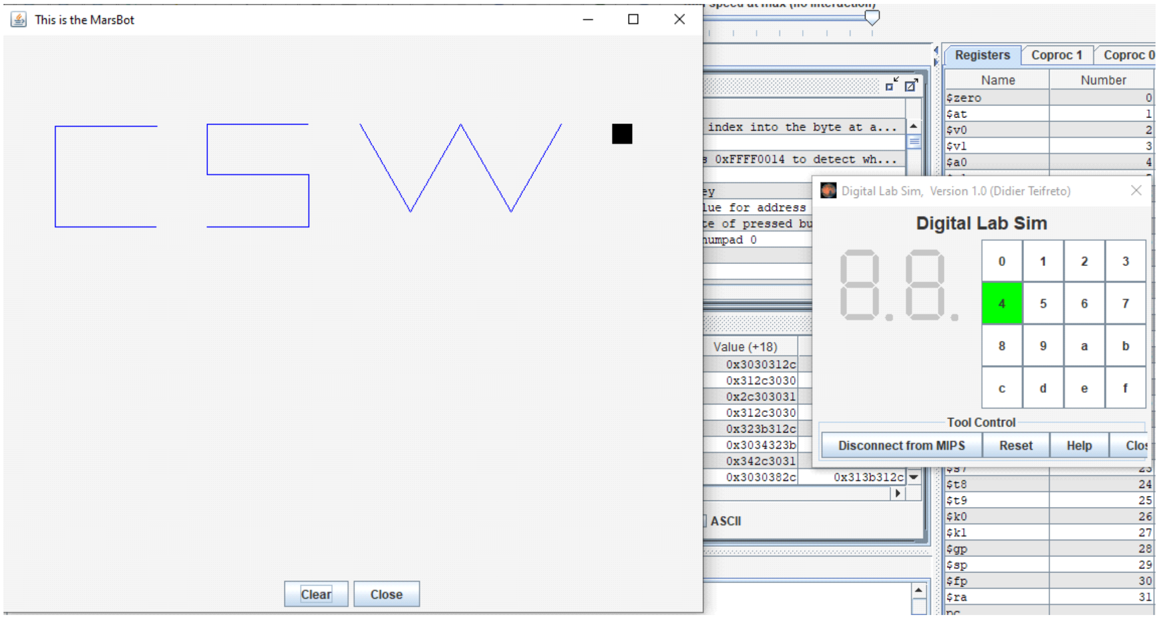
```
TRACK:
        li $at, LEAVETRACK      # change LEAVETRACK port
        addi $k0, $zero,1       # to logic 1,
        sb $k0, 0($at)          # to start tracking
        jr $ra
UNTRACK:
        li $at, LEAVETRACK      # change LEAVETRACK port to 0
        sb $zero, 0($at)        # to stop drawing tail
        jr $ra
ROTATE:
        li $at, HEADING         # change HEADING port
        sw $a0, 0($at)          # to rotate robot
        jr $ra
END:
        jal STOP
        li $v0, 10
        syscall
        j polling

```

Here's the simulations.



```
0x30352c30
0x2c303035
0x3b302c30
0x323b312c
0x3036323b
```

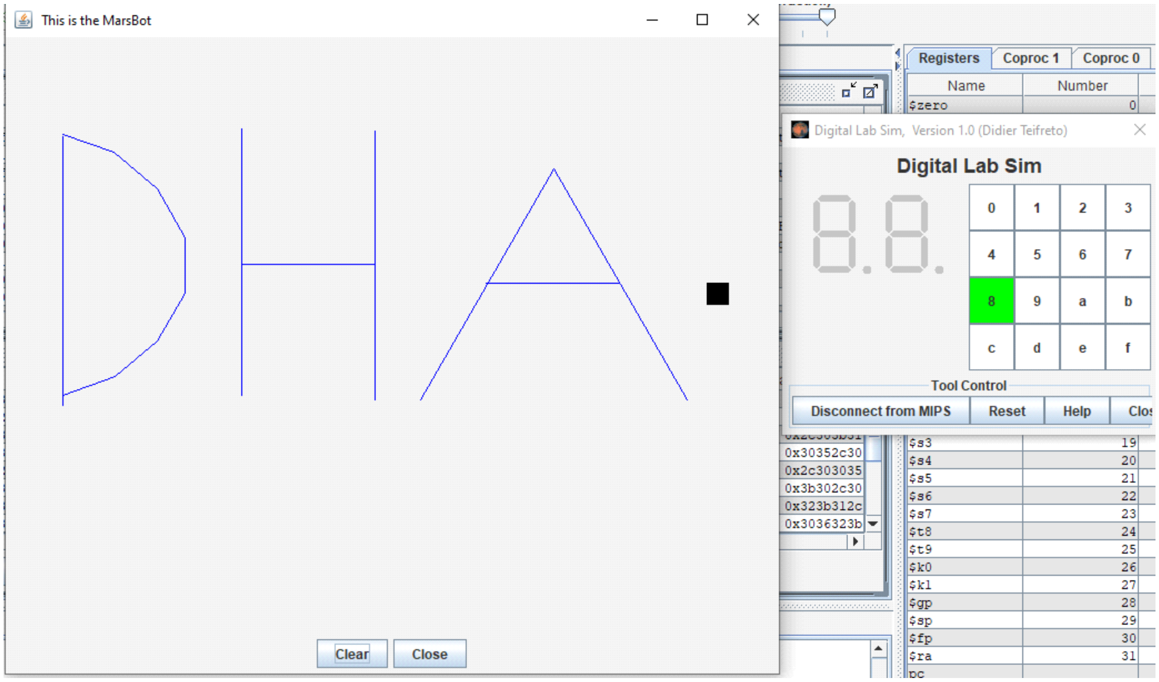
Name	Number
$zero	0
$s3	19
$s4	20
$s5	21
$s6	22
$s7	23
$t8	24
$t9	25
$k0	26
$k1	27
$gp	28
$sp	29
$fp	30
$ra	31



```
index into the byte at a...
s 0xFFFF0014 to detect wh...
ey
lue for address
te of pressed bu
numpad 0
```

Value (+18)	
0x3030312c	
0x312c3030	
0x2c303031	
0x312c3030	
0x323b312c	
0x3034323b	
0x342c3031	
0x3030382c	0x313b312c

Name	Number
$zero	0
$at	1
$v0	2
$v1	3
$a0	4
$t8	24
$t9	25
$k0	26
$k1	27
$gp	28
$sp	29
$fp	30
$ra	31
pc	



```
0x30352c30
0x2c303035
0x3b302c30
0x323b312c
0x3036323b
```

Name	Number
$zero	0
$s3	19
$s4	20
$s5	21
$s6	22
$s7	23
$t8	24
$t9	25
$k0	26
$k1	27
$gp	28
$sp	29
$fp	30
$ra	31
pc	