Assembly Language and Computer Architecture Lab

Lecturer: Dr. Le Ba Vui

SOICT

Final Project Report

Student:
Doan Thi Thu Quyen 20226063
Duong Phuong Thao 20226001

Contents

1 Problem 1: Moving a ball in the bitmap display

1.1 Program Description

The program displays a movable round ball on a 512 x 512 pixel bitmap screen, starting at the center. The ball moves based on keyboard inputs ('w', 'a', 's', 'd') and changes speed with 'z' (slower) or 'x' (faster). When the ball touches the screen edges, it reverses direction and continues moving. The ball is erased at its old position and redrawn at the new position to simulate movement.

1.2 Method

The method of this program involves setting the program, generating circle, handling input, detecting edges, drawing ball, and a loop procedure. The following steps outline the methodology:

1.2.1 Setup

Initialize variables, screen size, and ball properties.

1.2.2 Circle Generation

Precompute the circle's shape by storing its pixels relative to the center.

1.2.3 Input Handling

Continuously monitor for keyboard input and adjust the ball's direction or speed accordingly.

1.2.4 Edge Detection

Check if the ball has reached the screen boundary and reverse its direction if necessary.

1.2.5 Drawing the Ball

Draw the ball on the screen at its new position.

1.2.6 Loop and Refresh

Maintain a constant loop to update the ball's position and redraw it.

1.3 Algorithms

The algorithms applied in the program focus on the following key points:

1.3.1 Initialization

- Set up initial parameters such as screen size, ball position (x, y), ball direction (dx, dy), and speed t (frame delay).
- Precompute the circle's shape using a symmetry algorithm.

1.3.2 Main Loop

- **Input Handling:**
  - Check for keyboard input.
  - Update the ball's direction (dx, dy) or speed (t) based on the input.
- **Edge Detection:**
  - If the ball's next position exceeds the screen boundary, reverse its direction (dx = -dx or dy = -dy).
- **Ball Movement:**
  - Erase the ball at its current position by redrawing it with the background color.
  - Update the ball's position based on its direction and speed.
  - Draw the ball at the new position.
- **Delay:**
  - Use a system call to introduce a delay ('t') to control the speed of movement.
  - Repeat the loop.

1.3.3 Circle Drawing

- For a given radius r, calculate all pixels that make up the circle using the equation $x^2 + y^2 = r^2$.
- Use symmetry to minimize computation by generating points for all eight symmetrical positions simultaneously: $(x, y), (-x, y), (-x, -y), (x, -y), (y, x), (-y, x), (-y, -x), (y, -x)$.

1.3.4 Input Handing Logic

- Adjust the ball's direction: W ($dx = 0, dy = -1$), S ($dx = 0, dy = 1$), A ($dx = -1, dy = 0$), D ($dx = 1, dy = 0$).
- Adjust speed: Z (increase delay t), X (decrease delay t).

1.3.5 Edge Detection

- Calculate the ball's boundary positions using its center (x, y) and radius r.
- If the ball's boundary exceeds the screen dimensions (0 to 511), reverse its movement direction.

1.3.6 Ball Erasing and Redrawing

- Erase the ball at its old position by redrawing it with the background color.
- Update the ball's coordinates (x, y) and redraw it at the new position with its color.

1.3.7 Program Termination

Monitor for a specific key input (e.g., "O") to terminate the program.

1.4 Source code with comments

These below pictures displays some key functions' code along with comments for clarification.

1.4.1 Set up Program

```
.eqv KEY_CODE 0xFFFF0004
.eqv KEY_READY 0xFFFF0000

.eqv SCREEN_MONITOR 0x10010000

.data
circle_end:     .word   1       # The end of the "circle" array
circle:         .word           # The pointer to the "circle" 2-dimentional array
.text
setup:
        addi    s0, zero, 255   # x = 255
        addi    s1, zero, 255   # y = 255
        addi    s2, zero, 1     # dx = 1
        addi    s3, zero, 0     # dy = 0
        addi    s4, zero, 20    # r = 20
        addi    a0, zero, 50    # t = 50ms/frame
        jal     circle_data

input:
        li      t1, KEY_READY   # Check whether there is input data
        lw      t0, 0(t1)
        li      t6, 1
        bne     t0, t6, edge_check
        jal     direction_change
```

Figure 1: Set up program

1.4.2 Check Edge

```
# Check whether the circle has touched the edge
edge_check:
right:
        li t6, 1
        bne     s2, t6, left
        j       check_right
left:
        li t6, -1
        bne     s2, t6, down
        j       check_left

down:
        li t6, 1
        bne     s3, t6, up
        j       check_down
up:
        li t6, -1
        bne     s3, t6, move_circle
        j       check_up

move_circle:
        li s5, 0        # Set color to black
        jal     draw_circle     # Erase the old circle

        add     s0, s0, s2      # Set x and y to the coordinates of the center of the new circle
        add     s1, s1, s3
        li      s5, 0xFFB6C1    # Set color to pink
        jal     draw_circle     # Draw the new circle
loop:

        li a7, 32               # Syscall value for sleep
        ecall
        j       input           # Renew the cycle
```

Figure 2: Check Ball Meet Edge

1.4.3 Circle Data Generation

```
# Procedure below
circle_data:
        addi    sp, sp, -4      # Save ra
        sw      ra, 0(sp)
        la      s5, circle      # s5 becomes the pointer of the "circle" array
        mul     a3, s4, s4      # a3 = r^2
        li s7, 0        # pixel x (px) = 0

pixel_data_loop:
        bgt     s7, s4, data_end
        mul     t0, s7, s7      # t0 = px^2
        sub     a2, a3, t0      # a2 = r^2 - px^2 = py^2
        jal     root            # a2 = py

        addi    a1, s7, 0       # a1 = px
        li      s6, 0   # After saving (px, py), (-px, py), (-px, -py), (px, -py), we :

symmetric:
        li      t6, 2
        beq     s6, t6, finish
        jal     pixel_save      # px, py >= 0
        sub     a1, zero, a1
        jal     pixel_save      # px <= 0, py >= 0
        sub     a2, zero, a2
        jal     pixel_save      # px, py <= 0
        sub     a1, zero, a1
        jal     pixel_save      # px >= 0, py <= 0

        add     t0, zero, a1    # Swap px and -py
        add     a1, zero, a2
        add     a2, zero, t0
        addi    s6, s6, 1
        j       symmetric
```

Figure 3: Circle Data Generation

```
finish:
        addi    s7, s7, 1
        j       pixel_data_loop

data_end:
        la      t0, circle_end
        sw      s5, 0(t0)       # Save the end address of the "circle" array
        lw      ra, 0(sp)
        addi    sp, sp, 4
        jr      ra

root:                           # Find the square root of a2
        li      t0, 0   # Set t0 = 0
        li      t1, 0   # t1 = t0^2

root_loop:
        beq     t0, s4, root_end        # If t0 exceeds 20, 20 will be the square root
        addi    t2, t0, 1               # t2 = t0 + 1
        mul     t2, t2, t2              # t2 = (t0 + 1)^2
        sub     t3, a2, t1              # t3 = a2 - t0^2
        bgez    t3, continue            # If t3 < 0, t3 = -t3
        sub     t3, zero, t3

continue:
        sub     t4, a2, t2              # t4 = a2 - (t0 + 1)^2
        bgez    t4, compare             # If t4 < 0, t4 = -t4
        sub     t4, zero, t4

compare:
        blt     t4, t3, root_continue   # If t3 >= t4, t0 is not nearer to square root of a2 than t0 + 1
        add     a2, zero, t0            # Else t0 is the nearest number to square root of a2
        jr      ra
```

Figure 4: Circle Data Generation

```
direction_change:
        li      t5, KEY_CODE
        lw      t0, 0(t5)

case_o:
        li      t6, 111
        bne     t0, t6, case_d
        j       end_
case_d:
        li      t6, 100
        bne     t0, t6, case_a
        li      s2, 1   # dx = 1
        li      s3, 0   # dy = 0
        jr      ra

case_a:
        li      t6, 97
        bne     t0, t6, case_s
        li      s2, -1  # dx = -1
        li      s3, 0   # dy = 0
        jr      ra

case_s:
        li t6, 115
        bne     t0, t6, case_w
        li      s2, 0   # dx = 0
        li      s3, 1   # dy = 1
        jr      ra

case_w:
        li t6, 119
        bne     t0, t6, case_x
        li      s2, 0   # dx = 0
        li      s3, -1  # dy = -1
```

Figure 5: Change Direction Handling

1.4.4 Direction Handling

```
check_right:
        add     t0, s0, s4      # Set t0 to the right side of the circle
        li t6, 511
        bge     t0, t6, reverse_direction       # Reverse direction if the side has touched the edge
        j       move_circle     # Return if not

check_left:
        sub     t0, s0, s4      # Set t0 to the left side of the circle
        ble     t0, zero, reverse_direction     # Reverse direction if the side has touched the edge
        j       move_circle     # Return if not

check_down:
        add     t0, s1, s4      # Set t0 to the down side of the circle
        li t6, 511
        bge     t0, t6, reverse_direction       # Reverse direction if the side has touched the edge
        j       move_circle     # Return if not

check_up:
        sub     t0, s1, s4      # Set t0 to the up side of the circle
        ble     t0, zero, reverse_direction     # Reverse direction if the side has touched the edge
        j       move_circle     # Return if not

reverse_direction:
        sub     s2, zero, s2    # dx = -dx
        sub     s3, zero, s3    # dy = -dy
        j       move_circle
```

Figure 6: Change Direction Handling

1.4.5 Draw Circle and Pixel Draw

```
draw_circle:
        addi    sp, sp, -4      # Save ra
        sw      ra, 0(sp)
        la      s6, circle_end
        lw      s7, 0(s6)       # s7 becomes the end address of the "circle" array
        la      s6, circle      # s6 becomes the pointer to the "circle" array
draw_loop:
        beq     s6, s7, draw_end        # Stop when s6 = s7
        lw      a1, 0(s6)               # Get px
        lw      a2, 4(s6)               # Get py
        jal     pixel_draw
        addi    s6, s6, 8               # Get to the next pixel
        j       draw_loop
draw_end:
        lw      ra, 0(sp)
        addi    sp, sp, 4
        jr      ra
pixel_draw:
    li t0, SCREEN_MONITOR
    add t1, s0, a1              # fx = x + px
    add t2, s1, a2              # fy = y + py
    # Kiểm tra fx (x-coordinate)
    blt t1, zero, pixel_draw_end   # Nếu fx < 0, bỏ qua
    li t3, 511
    bgt t1, t3, pixel_draw_end     # Nếu fx > 511, bỏ qua
    # Kiểm tra fy (y-coordinate)
    blt t2, zero, pixel_draw_end   # Nếu fy < 0, bỏ qua
    bgt t2, t3, pixel_draw_end     # Nếu fy > 511, bỏ qua
    # Tính địa chỉ hợp lệ
    slli t2, t2, 9              # t2 = fy * 512
    add t2, t2, t1              # t2 = fy * 512 + fx
    slli t2, t2, 2              # t2 = (fy * 512 + fx) * 4
    add t0, t0, t2              # t0 = SCREEN_MONITOR + t2
```

Figure 7: Draw Circle and Pixel Draw

1.5 Simulation Result

Firstly, we need to connect the program with ”Bitmap Display” and ”Keyboard and Display MMIO Simulator”

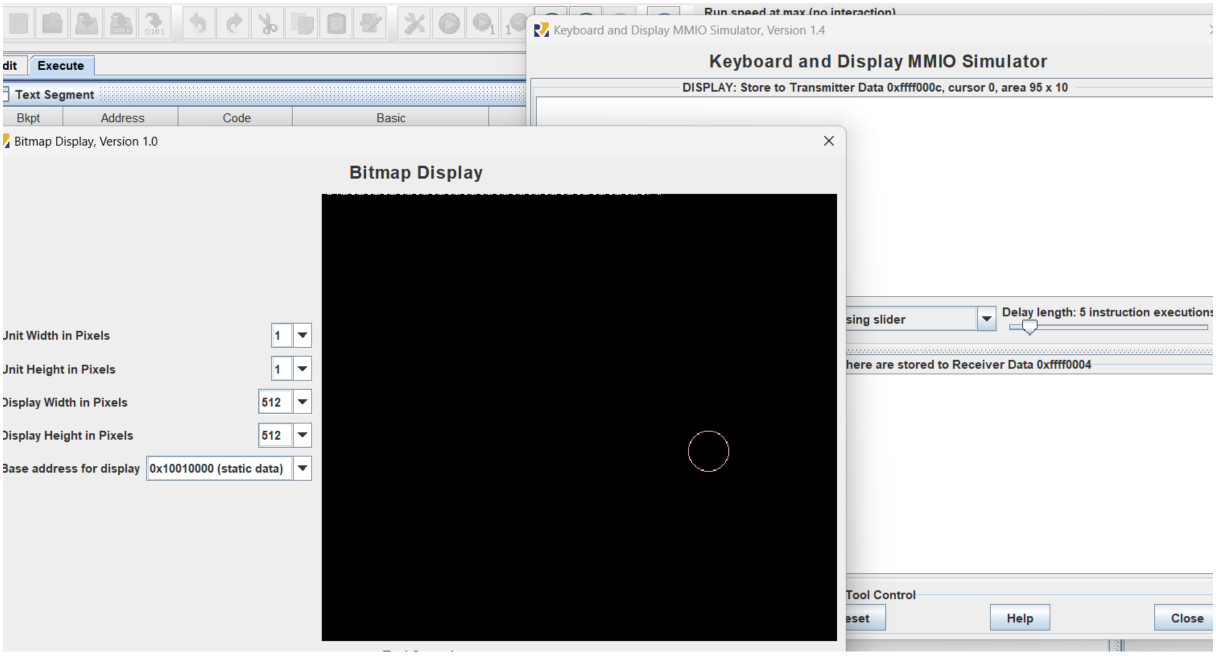

Figure 8: Connect with program

Secondly, we try to enter 'w','s','a','d' to change the ball's movement. In other cases, when 'z' is pressed, the ball speeds up, and pressing 'x' decreases the speed. These changes will be demonstrated in the presentation, as the images cannot display the real-time movement.

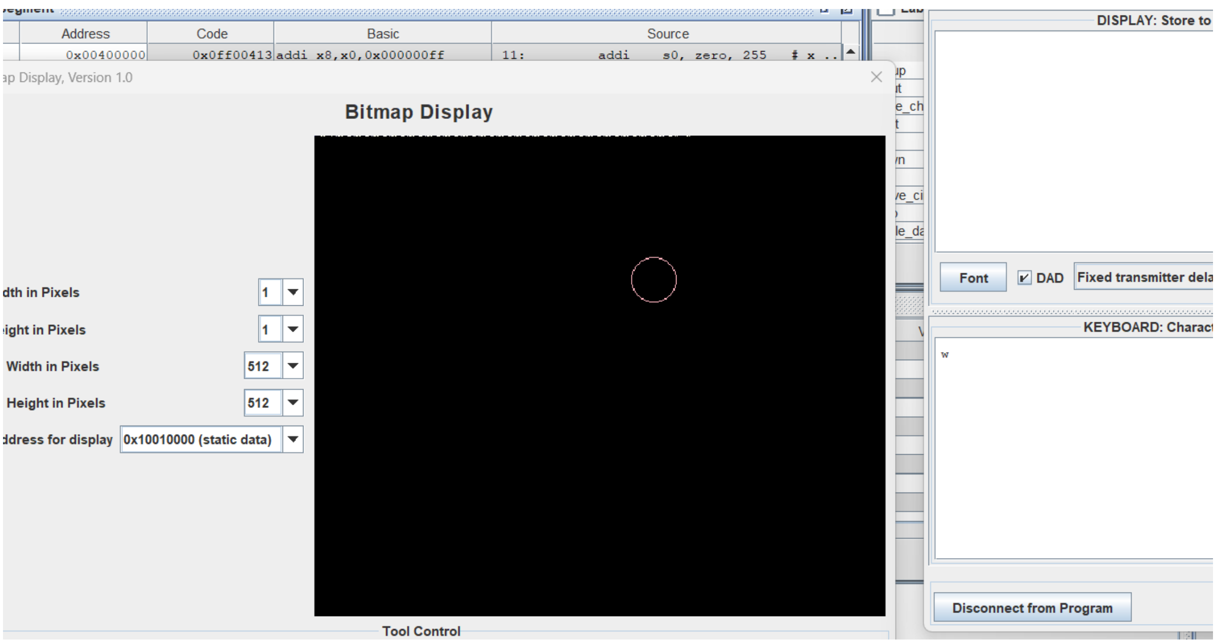

Figure 9: Enter 'w', move up

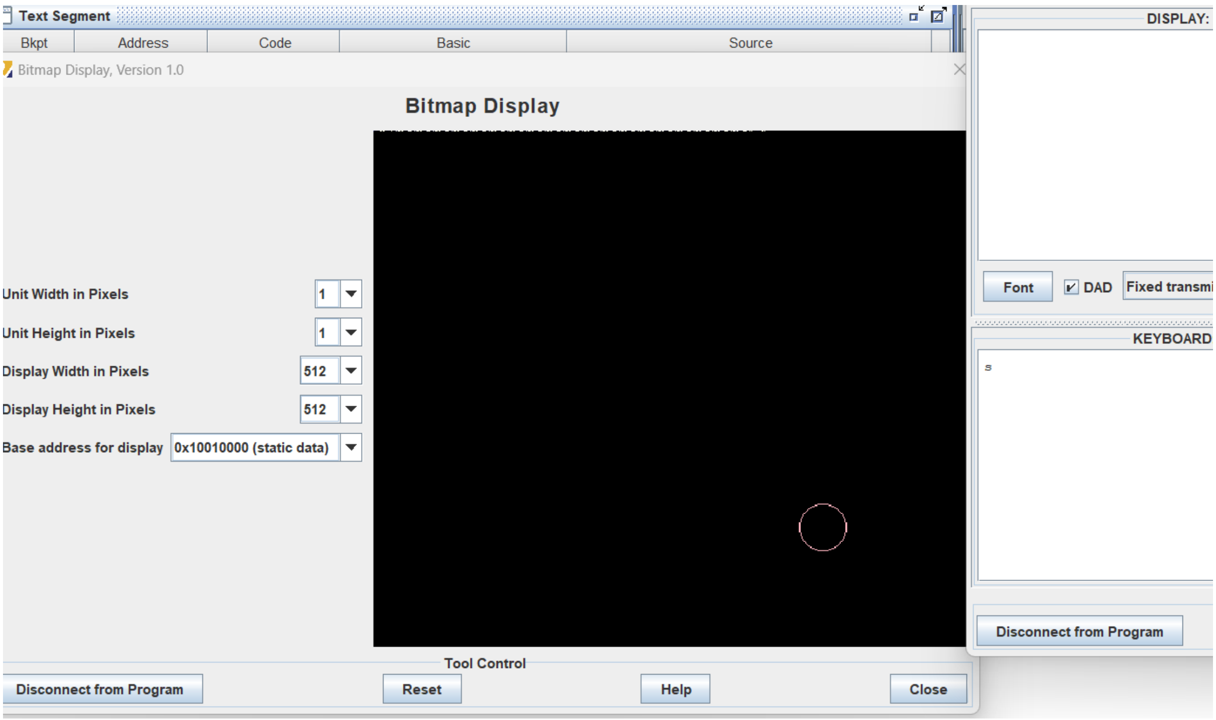

Figure 10: Enter ’s’, move down

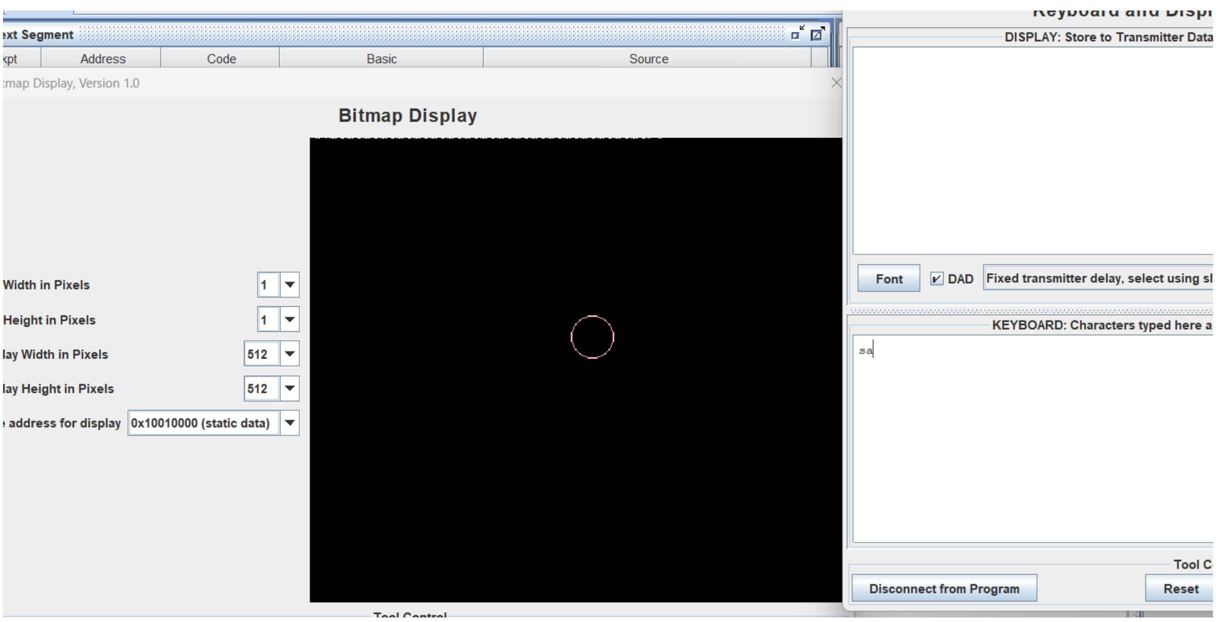

Figure 11: Enter ’a’, turn left

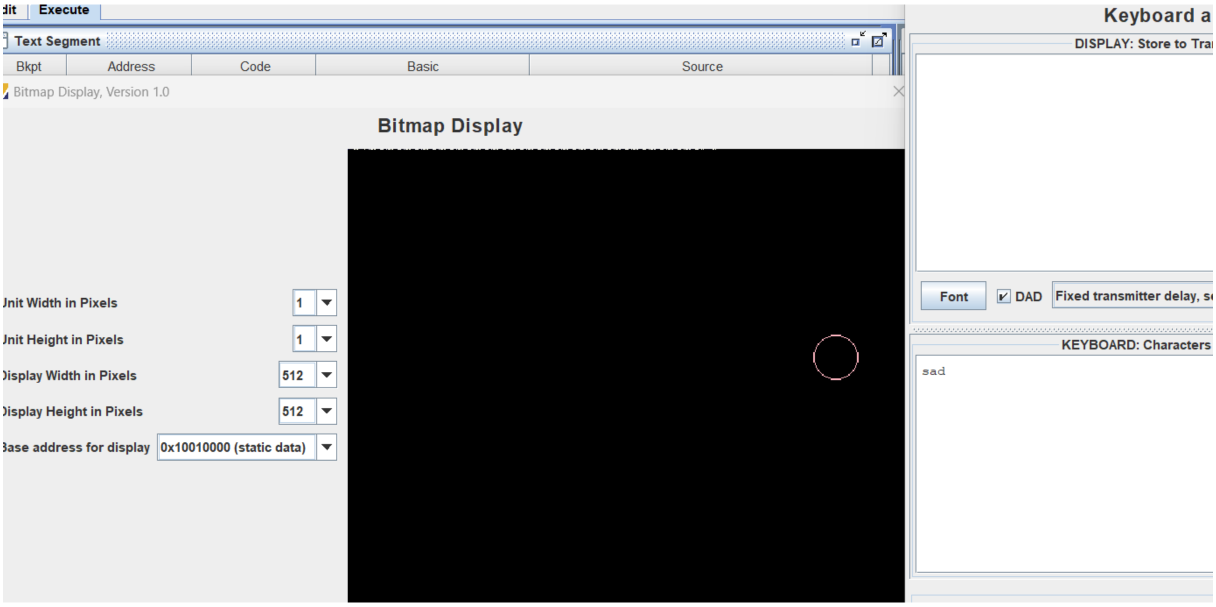

Figure 12: Enter 'd', turn right

Finally, we verify that the ball changes direction when it hits the edge, as expected.

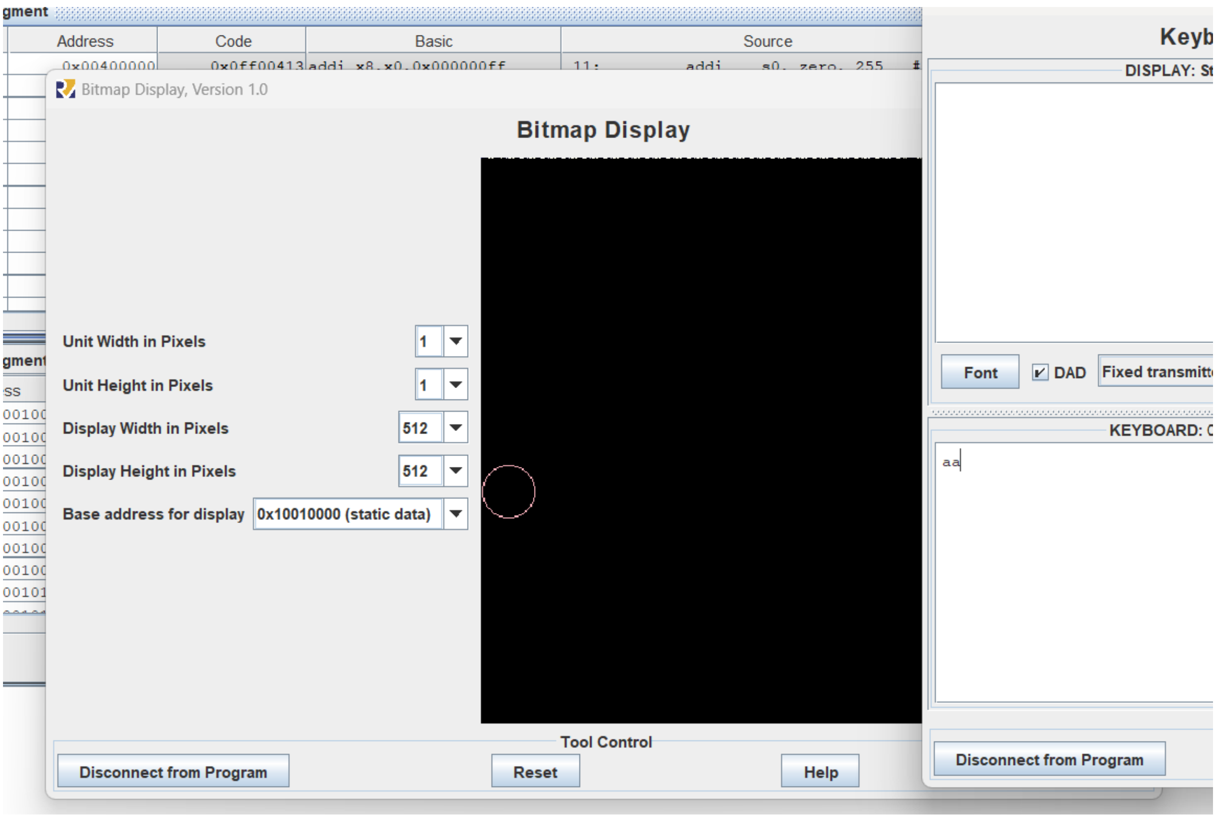

Figure 13: Ball hits the edge

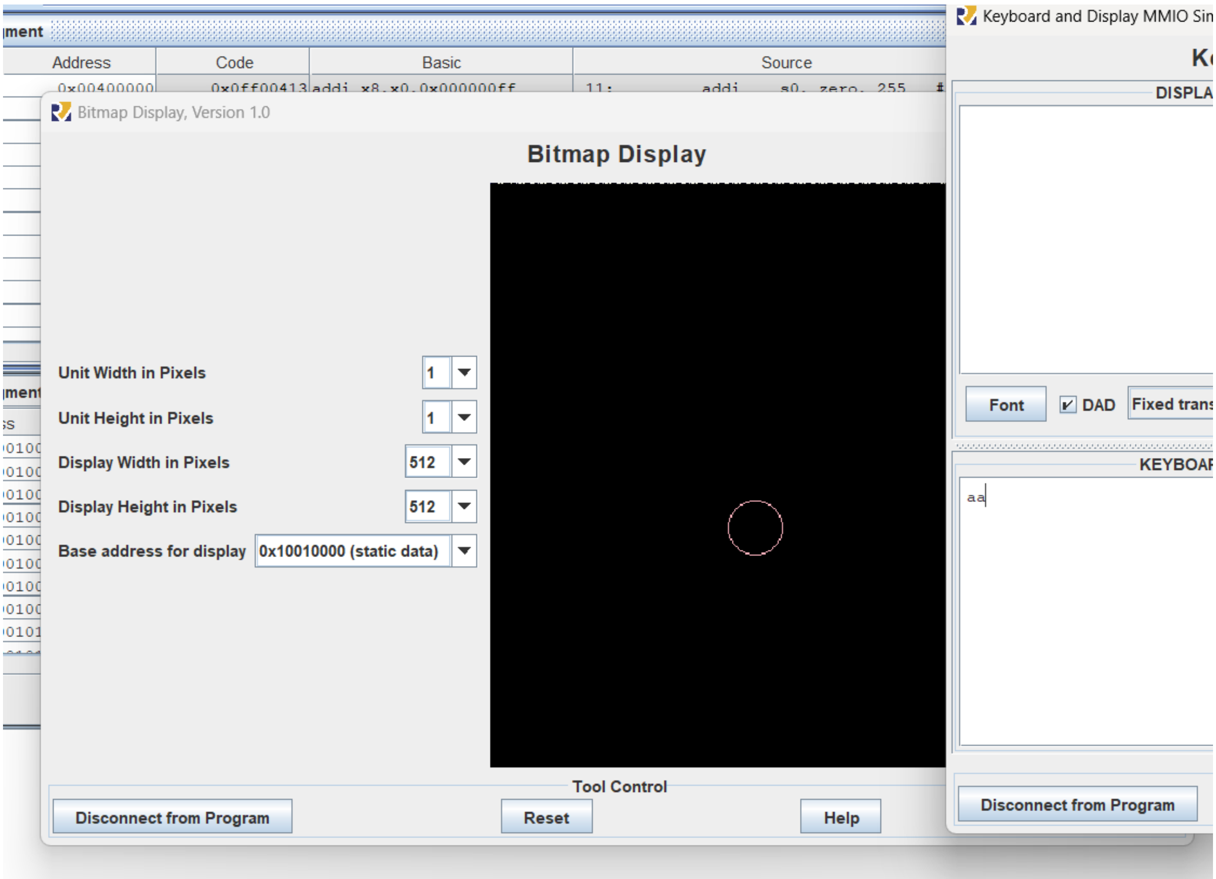

Figure 14: Ball changes direction

2 Problem 15: Number Memory Game

2.1 Program Description

The program is designed to test the user's memory. In each round, it generates 4 random integers between 0 and 1000 and gives the user five seconds to memorize them. After the time is up, the numbers are deleted, and the program displays a 4x4 grid containing 16 numbers. Among these, 4 are the original numbers, and the other 12 are random decoys.

The user needs to identify the 4 correct numbers and enter them, separated by spaces, followed by pressing Enter. If all 4 numbers are correct, the program starts a new round with a different set of numbers. If any number is incorrect or the user entered the wrong syntax, the program ends.

2.2 Method

The method of this program involves generating random numbers, displaying them to the user, and evaluating the user's ability to recall and input these numbers correctly. The following steps outline the methodology:

2.2.1 Initialization

The program begins by setting up the required registers and memory locations to handle input and output operations.

Key memory-mapped locations, such as ‘KEY_CODE‘, ‘KEY_READY‘, ‘DISPLAY_CODE‘, and ‘DISPLAY_READY‘, are assigned for handling keyboard input and display output.

2.2.2 Random Number Generation

Four random integers within a predefined range are generated using system calls (‘ecall‘) and stored in memory for later comparison.
Each generated number is processed and saved in the stack as individual digits separated by a space (ASCII ‘0x20‘).

2.2.3 Display Phase

The program constructs a 4x4 grid of numbers, mixing the four true numbers with randomly generated decoys. These are displayed sequentially, one digit at a time, with proper spacing and formatting.

2.2.4 Timer Implementation

A countdown timer is displayed on the screen, allowing the user a limited time to memorize the grid before it is cleared.

2.2.5 User Input

The user is prompted to input four numbers separated by spaces and terminated by pressing Enter.
Input is validated in real-time, ensuring that only numeric characters are accepted, and spaces are correctly handled as delimiters.

2.2.6 Validation

The user's input is checked against the stored numbers. If all four numbers match, the program proceeds to the next round. Otherwise, the program terminates.

2.2.7 Repetition or Termination

Upon successful validation, the program resets and starts a new round with a new set of random numbers.
If the user fails, the program ends execution.

2.3 Algorithms

The algorithms applied in the program focus on the following key points:

2.3.1 Random Number Generation and Storage

Use system call (‘ecall‘) to generate a random number within the range [0, 1000].
Store the number in memory (‘A‘ array) and break it down into individual digits.
Save each digit onto the stack, separated by a space character.

2.3.2 Display Construction

Iterate over the stored stack values to retrieve true numbers.
Use additional random numbers as decoys until the grid is filled.
Place numbers in a 4x4 grid, adjusting the cursor position after every fourth number.

2.3.3 Countdown Timer

Display the time remaining in seconds and tenths of seconds.
Use a loop to decrement the timer values and update the display accordingly.
Ensure synchronization with the display hardware by polling the 'DISPLAY_READY' register.

2.3.4 Input Handling

Poll the 'KEY_READY' register to detect keypresses
Read ASCII codes from 'KEY_CODE' and validate them.
Convert valid numeric input from ASCII to integers and store them in temporary variables.
Separate numbers using the space key and terminate input upon detecting the Enter key.

2.3.5 Validation of User Input

Compare each user-entered number with the stored true numbers in the 'A' array.
Mark matched numbers in the array to prevent duplicate matching.
If all four numbers are correctly identified, proceed to the next round; otherwise, terminate.

2.3.6 Cursor Management

Calculate cursor positions for displaying text and numbers using row and column indices.
Update the cursor position by modifying the 'DISPLAY_CODE' register.
Use the 'move_cursor' function to manage positioning for multi-line output.

2.4 Source code with comments

These below pictures contains source code with explanatory comments highlighting key functions.

2.4.1 Set up program

```
# 2. Assemble
# 3. Reset
	.eqv KEY_CODE 0xFFFF0004 # ASCII code from keyboard, 1 byte
	.eqv KEY_READY 0xFFFF0000 # =1 if has a new keycode ?
	# Auto clear after lw
	.eqv DISPLAY_CODE 0xFFFF000C # ASCII code to show, 1 byte
	.eqv DISPLAY_READY 0xFFFF0008 # =1 if the display has already to do
	# Auto clear after sw
.data:
	A: .word 1,2,3,4
	mes: .asciz "Timer: "
	# used register val: t0, a0, a1, s0, s1, s8, s9, s10, s11
.text
set_up_program:
	li a0, KEY_CODE
	li a1, KEY_READY
	li s0, DISPLAY_CODE
	li s1, DISPLAY_READY
	li s9, 12
	jal clear_display
	li s10, 0
	li s11, 0
	jal move_cursor # clear screen first
	addi t0, sp, 0 # pointer to saved correct number
	li t1, 0 # number of gened int counted
	li t2, 4
	la s2, A
```

Figure 15: Set up program

2.4.2 Generate Random Numbers

```
###------------------###
# generate four random number, save each number by each digit in stack, separate by ascii char " "
gen_random_num:
	beq t1, t2, print_to_display # if generate all 4 numbers, print them to display
	li a7, 42
	li a0, 0
	li a1, 1000 # bound of gened number
	ecall
	li a7, 1
	sw a0, 0(s2)
	addi s2, s2, 4
	# ecall # print to screen, s? xoá sau
	addi t4, t0, 0 # t4 is parameter pointer to save number to stakc
	addi a0, a0, 0 # a0 hold number to save
	jal save_num # save number to stack
	addi t0, t4, 0 # return value of new pointer in stack, set value to t0
	addi t1, t1, 1
	li a7, 11
	li a0, 32
	ecall
	j gen_random_num # generate new number
	# end function to generate number
```

Figure 16: Generate Random Numbers

2.4.3 Display Random Numbers

```
# function to print 4 gened number to display
print_to_display:
        # t0 is pointer to a position in stack where t0 + 1 -> sp hold int to guess
        li a0, KEY_CODE
        li a1, KEY_READY
        li s0, DISPLAY_CODE
        li s1, DISPLAY_READY
        addi t1, t0, 0
        li s10, 0       # set start column
        li s11, 0       # set start row
        jal move_cursor
loop_through_stack:
        beq t1, sp, end_display
        lw t2, 0(t1) # get int char from stack
wait_for_dis:
        lw t3, 0(s1) # t2 = [s1] = DISPLAY_READY
        beq t3, zero, wait_for_dis # if t2 == 0 then polling
show_key:
        addi t2, t2, 48 # since t2 is int value, we set t2 += 48 <=> t2 += '0'
        sw t2, 0(s0)
        addi t1, t1, 4
        j loop_through_stack
end_display:
```

Figure 17: Display Random Numbers

2.4.4 Print Timer

```
## ### ------------- ### ##
print_timer:
        li s10, 0 # set columns start value for timer
        li s11, 1 # set row start value for timer
        jal move_cursor
set_up_timer_para:
        la t5, mes
print_mes:
        lb t4, 0(t5)
        beq t4, zero, end_print_mes
wait_print_mes:
        lw t6, 0(s1)
        beq t6, zero, wait_print_mes
        sw t4, 0(s0)
        addi t5, t5, 1
        j print_mes
end_print_mes:
```

Figure 18: Print Timer

2.4.5 Count Down

```
count_down:
        li t5, 0        # second to countdown, in tenth
        li t6, 5        # second to countdown, in digit
set_cursor_print_time:
        li s10, 7       # set column value to print time
        li s11, 1       # set row value to print time
        jal move_cursor
print_second_cd:
print_tenth:
        lw t1, 0(s1)            # Check if display is ready
        beq t1, zero, print_tenth # Wait until the display is ready
        addi t5, t5, 48
        sw t5, 0(s0)
        addi t5, t5, -48
print_digit:
        lw t1, 0(s1)            # Check if display is ready
        beq t1, zero, print_digit # Wait until the display is ready
        addi t6, t6, 48
        sw t6, 0(s0)
        addi t6, t6, -48
        li a7, 32
        li a0, 1000
        ecall
decrement_both:
        blt zero, t6, decrement_and_repeat
        addi t6, t6, 9
        addi t5, t5, -1
        blt t5, zero, end_timer
        li s10, 7
        li s11, 1
        jal move_cursor
        j print_tenth
decrement_and_repeat:
        addi t6, t6, -1
        li s10, 8
```

Figure 19: Count Down

2.4.6 Print Number Matrix

```
## ========= ========= ========= ##
clear_screen:
        li s9, 12
        jal clear_display
        li s10, 0
        li s11, 0
        jal move_cursor
print_number_fct:
        # t0 hold pointer to stack where correct value are held
        li t1, 16 # max number of int to print
        addi t3, t0, 0 # pointer in stack that hold true number
        li t2, 4 # number of true value left needed to print to screen
        li t4, 0 # number of fake value printed to screen
        li a6, 0 # total number of int printed to screen
print_num:
        li a7, 41
        li a0, 0
        ecall # get random integer
        beq t3, sp, print_mock_int # if print all true int, only print mock int
        li a6, 0
        add a6, t2, t4
        bge a6, t1, print_true_int # if still has true value left, print true value
        blt a0, zero, print_mock_int # if random int < 0, print fake int
        j print_true_int
# >>>>>>>>><<<<<<<<<< #
```

Figure 20: Print Number Matrix

```
# >>>>>>>>><<<<<<<<<< #
print_true_int:
        lw s7, 0(s1)
        beq s7, zero, print_true_int
        lw a4, 0(t3) # get int value in stack
        addi t3, t3, 4
        addi a4, a4, 48
        sw a4, 0(s0)
        addi a4, a4, -32 # if a4 is ascii code for space char, then a4 - 32 = 0
        beq a4, zero, post_print_true
        j print_true_int
print_mock_int:
        li a7, 42
        li a0, 0
        li a1, 1000
        ecall # get random int in [0, 1000]
        li t5, 10
print_digit_loop:
        rem t6, a0, t5 # get last digit
another_wait:
        lw s7, 0(s1)          # Check if display is ready
        beq s7, zero, another_wait # Wait until the display is ready
        addi t6, t6, 48
        sw t6, 0(s0)
        addi t6, t6, -48
        div a0, a0, t5 # divide by 10
        beq a0, zero, post_print_mock # if finish printing fake int, go to post print
        j print_digit_loop
# >>>>>>>>><<<<<<<<<< #
#       |       |       #
```

Figure 21: Print Number Matrix

```
# >>>>>>>>><<<<<<<<< #
#       |       |       #
post_print_mock:
print_space:
        lw s7, 0(s1)
        beq s7, zero, print_space
        li t6, 32
        sw t6, 0(s0) # print space to screen
        addi t4, t4, 1 # increase fake number print to screen
        j newline_decider
post_print_true:
        addi t2, t2, -1 # increase true number print to screen
newline_decider:
        # after each four number, move cursor down one row, set column to 0
        li t5, 4
        li a6, 0
        sub a6, a6, t2
        addi a6, a6, 4
        add a6, a6, t4
        beq a6, t1, end_print_num
        rem t5, a6, t5 # remainder of num_of_int_print % 4
        beq t5, zero, new_line
        j old_line
new_line:
        li s10, 0
        srli s11, s11, 8 # since in move_cursor, s11 *= 2^8, we have to divide by 2^8 first
        addi s11, s11, 1 # since above we set initial s11 = 0, we increment them after each four num
        jal move_cursor
old_line:
        j print_num
end_print_num:
#       |       |       #
###/////////////////////###
```

Figure 22: Print Number Matrix

2.4.7 Get User Answer

```
###/////////////////////###
get_user_ans:
        li s10, 0 # number of int that user has entered
        li s11, 3 # number of int that user supposed to enter by the time 'entered'
set_up:
        li a0, KEY_CODE
        li a1, KEY_READY
        li s0, DISPLAY_CODE
        li s1, DISPLAY_READY
        li t1, 0 # number getted
        li t2, 10 # multiplication value
        li t5, 48 # 0 in ascii
        li t6, 57 # 9 in ascii
        li s3, 32 # space in ascii
        li s4, 10 # newline in ascii
loop:
wait_for_key:
        lw t3, 0(a1)
        beq t3, zero, wait_for_key
read_key:
        lw t4, 0(a0)
        beq t4, s3, space
        beq t4, s4, newline
        blt t4, t5, end_program
        blt t6, t4, end_program
        addi t4, t4, -48
        mul t1, t1, t2
        add t1, t1, t4
        j wait_for_key
```

Figure 23: Get User Answer

```
space:
        addi s10, s10, 1
        blt s11, s10, end_program
        la s2, A
        li s5, 0 # number checked in A
        li s7, 4 # max number in A
check_int:
        lw s6, 0(s2)
        beq s5, s7, end_program # if we go through all number in A and get no match, end program
        beq t1, s6, mark_int_in_A # if we find a match, mark them with -1
        addi s5, s5, 1 # increase number of A we go through
        addi s2, s2, 4 # increase address pointer
        j check_int
mark_int_in_A:
        li s6, -1
        sw s6, 0(s2)
        j set_up # after verify number in A, return to getting user input
newline:
        blt s10, s11, end_program
        j next_round
```

Figure 24: Get User Answer

2.5 Simulation Result

Firstly, we need to connect the program with the "Keyboard and and Display MMIO Simulator" tool.

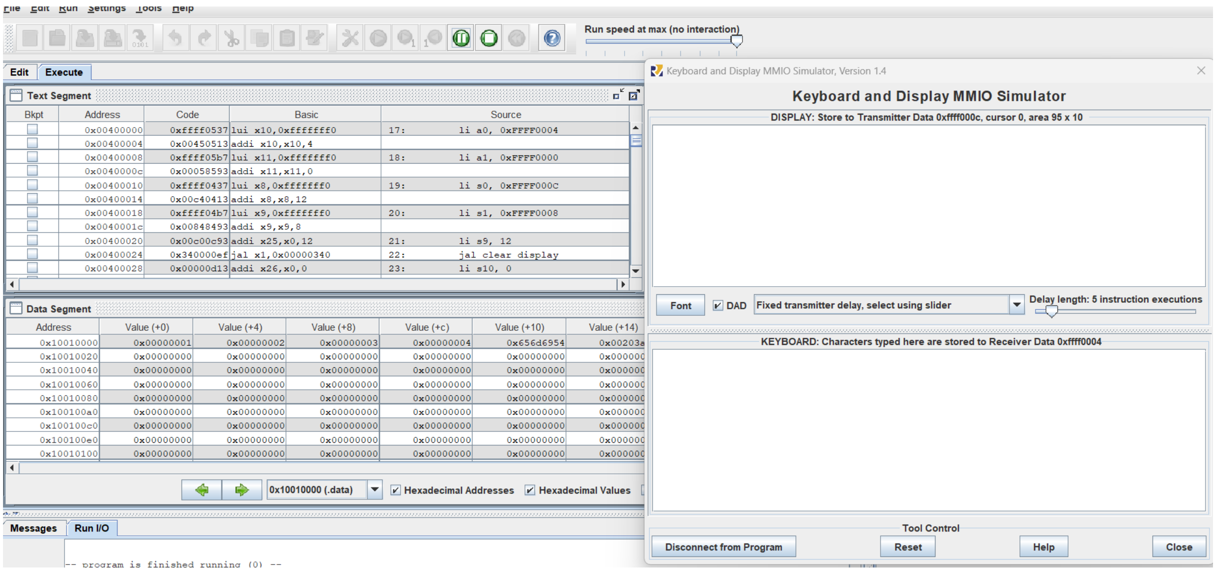

Figure 25: Connect tool with program

Secondly, we press the "Reset" button to generate 4 numbers.

Figure 26: Generate four numbers

Thirdly, after the timer countdown to 0, the Display screen will show 16 numbers for user to choose and enter 4 numbers into keyboard.

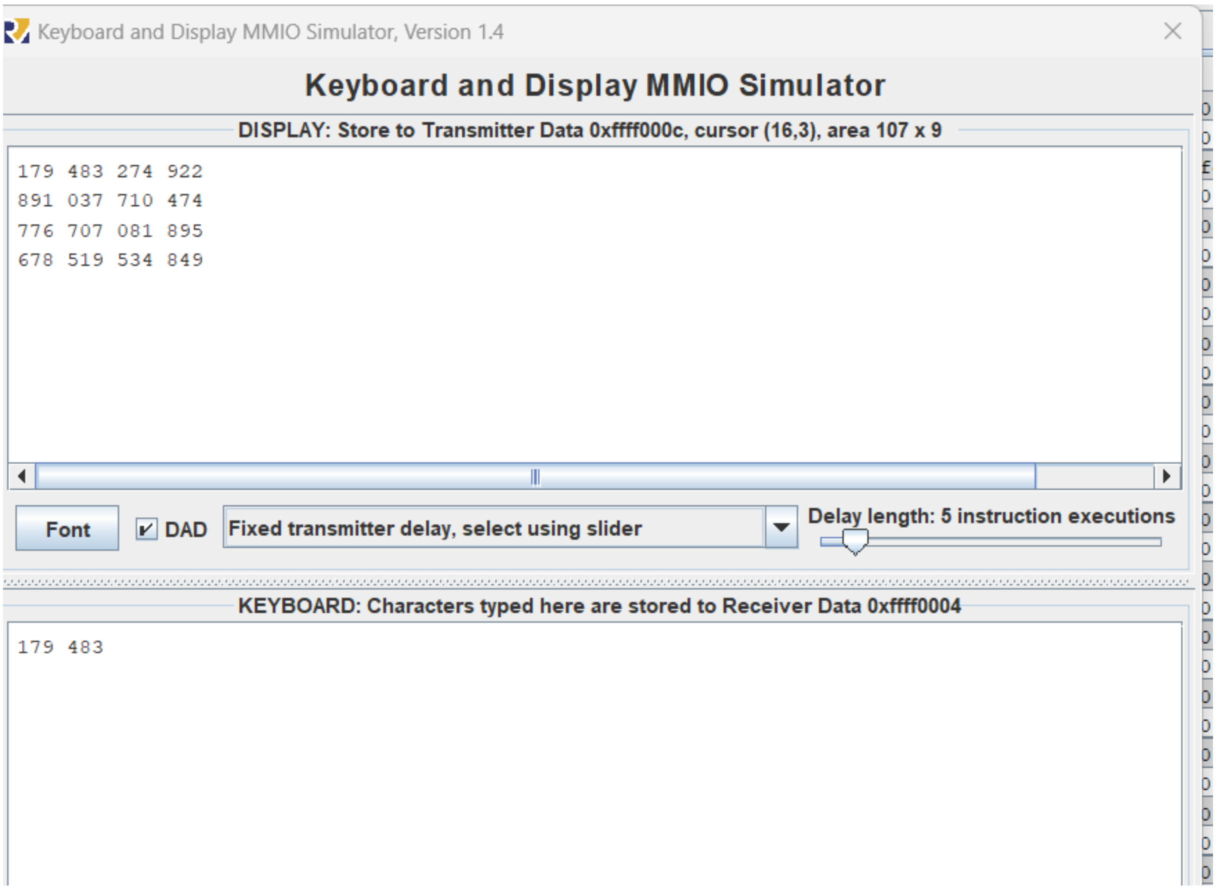

Figure 27: User enter numbers

Finally, based on the numbers entered by the user, the program determines whether to proceed to the next round or terminate.
The two pictures below illustrate two scenarios: in the first picture, the user enters the correct four numbers (in this case, correct numbers are 139,469 848, 549 in the previous round) then advances to the next round, in the second case, the user enters incorrect numbers, and the program ends.

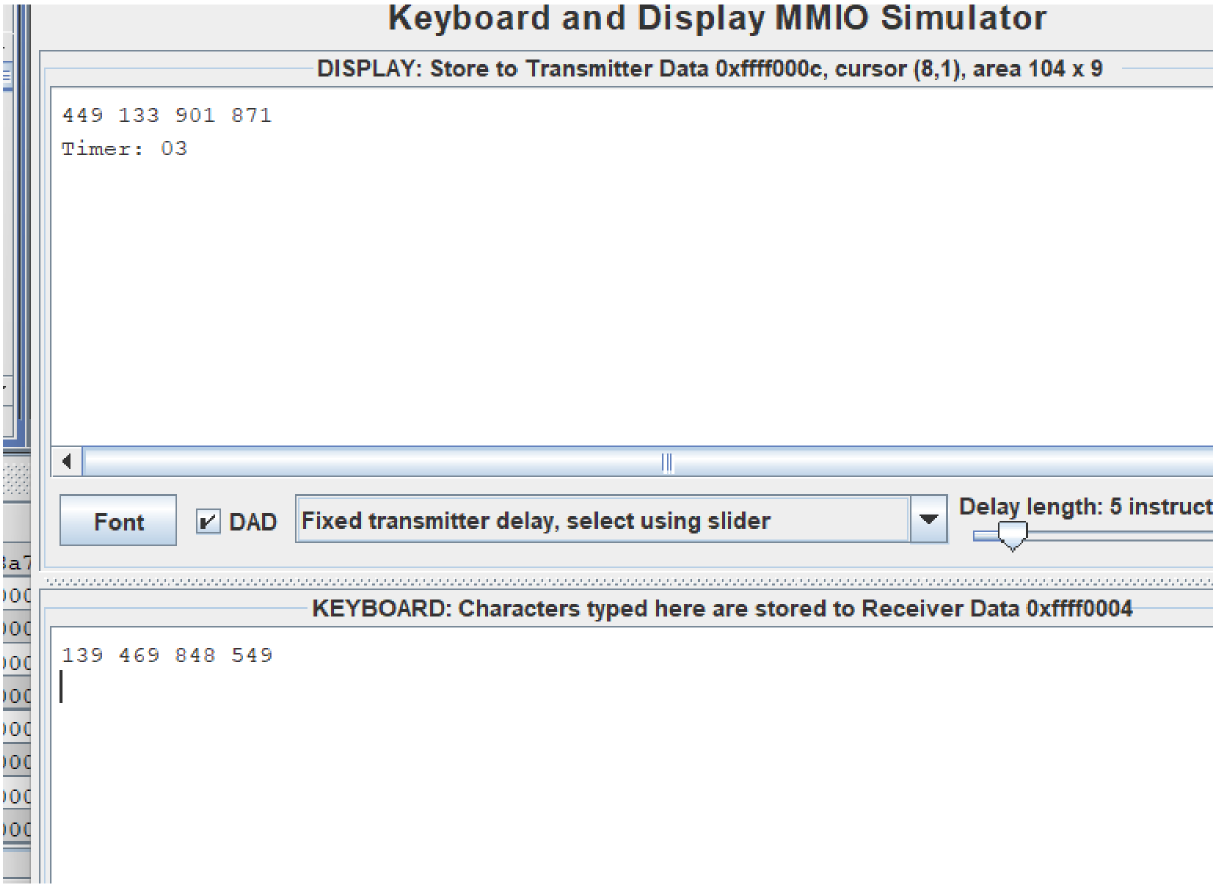

Figure 28: Case 1

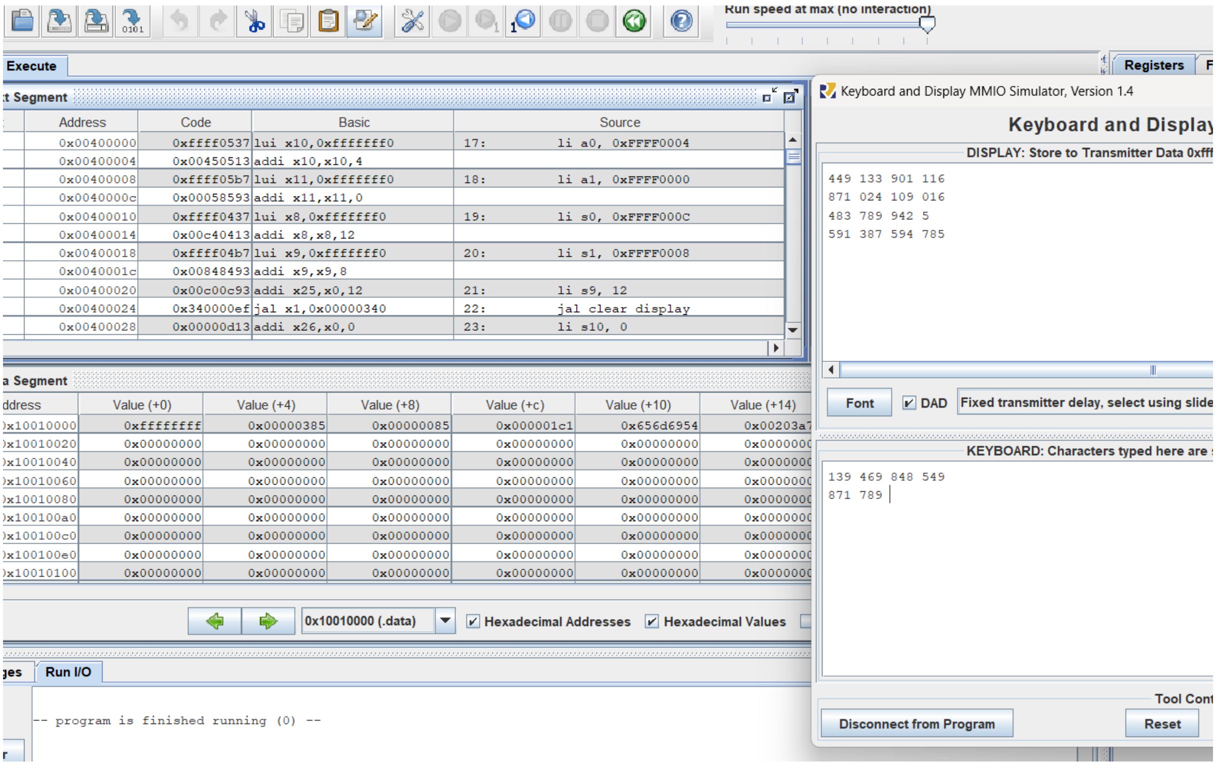

Figure 29: Case 2