HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY
ICT K67

IT3280E

ASSEMBLY LANGUAGE AND COMPUTER ARCHITECTURE LAB

FINAL PROJECT REPORT

PROBLEM 1: MOVING A BALL IN THE BITMAP DISPLAY
PROBLEM 9: TESTING SORTING ALGORITHMS

MEMBER LIST

Trương Linh Duyên	20225968	duyen.tl225968@sis.hust.edu.vn
Đinh Nguyễn Sơn	20225997	son.dn225997@sis.hust.edu.vn

Teacher: Lê Bá Vui

Hà Nội — 2024

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

In this report, we explore solutions to two problems using the RISC-V assembly language, a modern open-source instruction set architecture (ISA) that emphasizes simplicity and modularity. RISC-V's clean and flexible design makes it a suitable choice for low-level programming tasks that require efficiency and control. Our team presents solutions to Problem 1: Moving a Ball in the Bitmap Display and Problem 9: Testing Sorting Algorithms, both implemented in assembly language on the RISC-V architecture.

For Problem 1, we design an interactive program where a ball moves within a bitmap display, responding to keyboard inputs to adjust its direction and speed. The ball's movement is constrained within the display boundaries, reversing direction when it hits the screen edges. This solution demonstrates the application of basic input-output handling, boundary checking, and real-time user interaction, making use of fundamental assembly constructs.

For Problem 9, we implement various sorting algorithms to test their performance and functionality. The algorithms are implemented from scratch, providing insights into their efficiency and behavior at the assembly level. Through this problem, we highlight the importance of understanding sorting mechanics and optimizing code for performance in low-level programming.

Both problems showcase the power and flexibility of assembly programming, especially with RISC-V, allowing us to leverage the hardware directly while optimizing performance. Our solutions illustrate key concepts of system-level programming, hardware control, and algorithmic implementation.

2 Problem 1: MOVING A BALL IN THE BITMAP DISPLAY

2.1 Problem Statement

Create a program that displays a movable round ball on the bitmap screen. If the ball touches the edge of the screen, it will move in the opposite direction.

Requirement:

- Set display width and height to 512 pixels, unit width and height to 1 pixel.
- The direction of movement depends on the key pressed from the keyboard. (W moves up, S moves down, A moves left, D moves right, Z speeds up, X slows down).
- The default position is the center of the screen.

2.2 Algorithm Idea

In this section, we mainly focus on our general idea to solve the problem, read the following section for more details.

Initialization

Initially, the program sets up the following parameters:

- The ball's starting position is set to the center of the screen, with coordinates $(x = 256, y = 256)$.
- The ball has a radius of 20 pixels, which is used to calculate its boundary when moving.
- The default movement distance per step is set to 10 pixels.
- The ball's color is set to yellow.
- The program defines a speed variable that controls the delay between updates, thus affecting the movement speed of the ball.

Keyboard Input Handling

The program continuously monitors the keyboard for key presses. Depending on the key pressed, the ball's movement and speed are adjusted:

- **Movement Keys:**
  - `W`: Move the ball up.
  - `S`: Move the ball down.
  - `A`: Move the ball left.
  - `D`: Move the ball right.
- **Speed Adjustment Keys:**
  - `Z`: Increase the speed (decrease delay).
  - `X`: Decrease the speed (increase delay).
- `Enter`: Exit the program.

Boundary Check

Before each move, the program checks whether the ball will go beyond the screen boundaries. The ball's position is compared with the screen dimensions (512x512 pixels), considering its radius. The following checks are performed:

- If the ball exceeds the right edge, the horizontal direction (dx) is reversed.
- If the ball exceeds the left edge, the horizontal direction (dx) is reversed.
- If the ball exceeds the top edge, the vertical direction (dy) is reversed.
- If the ball exceeds the bottom edge, the vertical direction (dy) is reversed.

These checks ensure that the ball bounces back when hitting the edges.

Ball Movement and Redrawing

After the boundary check, the ball's position is updated:

- The ball's previous position is erased by setting the pixel values in the bitmap to the background color.

- The new position of the ball is computed by adding the respective movement distances (dx, dy) to the current coordinates (x, y).
- The ball is redrawn at the new position using the Bresenham Circle Algorithm.

Bresenham Circle Algorithm

To render the ball as a circle on the screen, the program uses the Bresenham Circle Algorithm, which efficiently computes the points of a circle and plots them on the bitmap. The algorithm calculates the points symmetrically around the center of the circle, ensuring that all eight octants are plotted correctly. The key steps involve:

- Starting with initial values of $(x = 0, y = r)$ and an initial decision parameter $d = 3 - 2r$, where r is the radius.
- Iterating through the circle's points and plotting the corresponding pixel locations in all eight symmetrical positions.
- Adjusting the decision parameter d after each point is plotted, depending on the relative positions of the calculated points.

Speed Control

The ball's speed is adjusted by changing the delay between updates:

- Pressing `Z` decreases the delay, causing the ball to move faster.
- Pressing `X` increases the delay, slowing the ball down.

The speed change is managed by adjusting the delay variable, which controls how quickly the game loop proceeds.

Game Loop

The program operates in a continuous game loop:

- The program waits for keyboard input and processes it.
- Based on the input, the direction of movement or speed is updated.
- After updating the position, the ball is erased and redrawn at the new location.
- A delay is introduced, and the loop repeats.

Program Exit

The program runs in a loop until the `Enter` key is pressed, at which point it terminates gracefully by invoking the exit system call.

2.3 Implementation Pipeline

The implementation of the game is organized into multiple stages, each focusing on a specific task, from initialization to handling user input, checking boundaries, moving the ball, drawing it on the screen, and adjusting speed. This section outlines the key steps involved in the implementation, breaking down each section of the code for better understanding.

Initialization of Game Variables

The first step in the pipeline is the initialization of game variables. This step sets up the initial state of the ball, the screen parameters, movement variables, and the color of the ball.

```
# Initialization
li s0, 256      # x = 256 (initial x-coordinate)
li s1, 256      # y = 256 (initial y-coordinate)
li s2, 20       # R = 20 (radius)
li s3, 512      # Screen width (512 pixels)
li s4, 512      # Screen height (512 pixels)
li s5, YELLOW   # Ball color (yellow)
li s6, MOVE_DISTANCE  # Move distance per step (10 pixels)
li s7, 0        # dx = 0 (horizontal movement change)
li s8, 0        # dy = 0 (vertical movement change)
li s9, 70       # Initial speed (delay)
```

Explanation:

- The variables ‘s0‘ and ‘s1‘ represent the initial coordinates of the ball (set to the center of the screen at ‘(256, 256)‘).
- ‘s2‘ defines the ball’s radius, which is ‘20‘ pixels.
- ‘s3‘ and ‘s4‘ define the screen’s width and height, which are both ‘512‘ pixels.
- ‘s5‘ stores the color of the ball, which is set to yellow.
- ‘s6‘ defines the movement distance per step (10 pixels).
- ‘s7‘ and ‘s8‘ are initialized to zero and will store the change in position (‘dx‘ and ‘dy‘) for horizontal and vertical movement respectively.
- ‘s9‘ is the delay used to control the speed of the game, initially set to ‘70‘.

Keyboard Input Handling

The program continuously reads the keyboard input to detect key presses that control the ball’s movement. Each key has a specific function.

```
# Read Keyboard Input
li t0, KEYBOARD_CONTROL    # Load the control register address
lw t1, 0(t0)               # Read the status of the keyboard
beqz t1, check_bounds      # If no key pressed, check bounds

# Read the key data
li t0, KEYBOARD_DATA
lw t2, 0(t0)               # Read the key code

# Handle movement keys
li t3, KEY_A
beq t2, t3, move_left      # Move left (A)
li t3, KEY_D
beq t2, t3, move_right     # Move right (D)
li t3, KEY_W
beq t2, t3, move_up        # Move up (W)
li t3, KEY_S
beq t2, t3, move_down      # Move down (S)

# Handle speed keys
li t3, KEY_Z
beq t2, t3, speed_up       # Speed up (Z)
li t3, KEY_X
```

```
beq t2, t3, speed_down    # Speed down (X)

# Exit program
li t3, KEY_ENTER
beq t2, t3, exit_program  # Exit (Enter)

j check_bounds
```

Explanation:

- The first step is to check if any key is pressed by reading the keyboard control register. If no key is pressed, the program jumps to the boundary checking phase (‘check_bounds‘).
- If a key is pressed, the key code is read from the ‘KEYBOARD_DATA‘ register.
- The program then checks if the key corresponds to one of the predefined movement or speed control keys (‘W‘, ‘A‘, ‘S‘, ‘D‘, ‘Z‘, ‘X‘, ‘Enter‘).
  - ‘A‘ moves the ball left (‘move_left‘), ‘D‘ moves it right (‘move_right‘), ‘W‘ moves it up (‘move_up‘), and ‘S‘ moves it down (‘move_down‘).
  - ‘Z‘ and ‘X‘ are used to increase or decrease the movement speed (‘speed_up‘ and ‘speed_down‘).
  - ‘Enter‘ triggers the exit of the program (‘exit_program‘).

Boundary Checking

The ball's movement is restricted within the screen boundaries. If the ball touches the edge of the screen, it will reverse direction along the respective axis.

```
check_bounds:
    # Check right boundary (x + R + dx >= screen_width)
    add t0, s0, s2        # x + R
    add t0, t0, s7        # x + R + dx
    bge t0, s3, reverse_x # If x + R + dx >= screen_width, reverse direction

    # Check left boundary (x - R + dx < 0)
    sub t0, s0, s2        # x - R
    add t0, t0, s7        # x - R + dx
    bltz t0, reverse_x    # If x - R + dx < 0, reverse direction

    # Check top boundary (y - R + dy < 0)
    sub t0, s1, s2        # y - R
    add t0, t0, s8        # y - R + dy
    bltz t0, reverse_y    # If y - R + dy < 0, reverse direction

    # Check bottom boundary (y + R + dy >= screen_height)
    add t0, s1, s2        # y + R
    add t0, t0, s8        # y + R + dy
    bge t0, s4, reverse_y # If y + R + dy >= screen_height, reverse direction

    j update_position     # If within bounds, update position
```

Explanation:

- The boundaries of the screen are checked along both the x and y axes:

– Right boundary: If the ball's rightmost edge ('x + R + dx') exceeds or equals the screen width, the horizontal direction is reversed ('reverse_x').
– Left boundary: If the ball's leftmost edge ('x - R + dx') is less than zero, the horizontal direction is reversed ('reverse_x').
– Top boundary: If the ball's topmost edge ('y - R + dy') is less than zero, the vertical direction is reversed ('reverse_y').
– Bottom boundary: If the ball's bottommost edge ('y + R + dy') exceeds or equals the screen height, the vertical direction is reversed ('reverse_y').

Updating the Position and Redrawing the Ball

Once the boundaries have been checked and movement is determined, the ball's position is updated, and it is redrawn at its new location.

```
update_position:
    # Erase previous ball position (set background color)
    li s5, BACKGROUND       # Set color to background (black)
    jal draw_circle         # Draw ball at previous position

    # Update position (x += dx, y += dy)
    add s0, s0, s7          # x += dx
    add s1, s1, s8          # y += dy

    # Draw the ball at the new position
    li s5, YELLOW           # Set color to yellow
    jal draw_circle         # Draw ball at new position

    # Delay
    mv a0, s9
    li a7, 32
    ecall                   # System call for delay

    j game_loop             # Return to game loop
```

Explanation:

- The previous position of the ball is erased by drawing it in the background color ('BACKGROUND').
- The new position is computed by adding 'dx' and 'dy' to the current position ('s0' and 's1').
- The ball is then drawn at its updated position using the 'draw_circle' subroutine, and the ball is colored yellow.
- A delay is introduced to control the speed of the game, based on the value in 's9'.
- The game loop then repeats.

Bresenham Circle Drawing Algorithm

To draw the ball (represented as a circle), the 'draw_circle' subroutine uses Bresenham's circle algorithm, which efficiently computes the points of a circle.

```
draw_circle:
    # Initialize circle drawing variables
    li t0, 0               # x = 0
    mv t1, s2              # y = r (radius)
```

```
    li t2, 3               # d = 3 - 2r (initial decision parameter)
    sub t3, t2, s2         # d = 3 - 2r - r (adjusted decision parameter)
    sub t3, t3, s2

draw_loop:
    bgt t0, t1, draw_end  # Loop while x <= y

    # Plot the 8 symmetric points of the circle
    add a0, s0, t0        # Plot point (x0 + x, y0 + y)
    add a1, s1, t1
    jal plot_pixel_safe    # Plot point on screen

    # Repeat for other symmetrical points...

    # Update decision parameter d and other variables
    bgez t3, adjust_d_positive

    # Adjust decision parameter
adjust_d_positive:
    add t3, t3, t4
    addi t1, t1, -1
    continue_draw:
    addi t0, t0, 1         # x++
    j draw_loop

draw_end:
    ret
```

Explanation:

- The ‘draw_circle‘ subroutine uses Bresenham’s algorithm to plot a circle by calculating points along the perimeter of the circle.
- The algorithm uses a decision parameter (‘d‘) that determines whether the next point should be plotted horizontally or diagonally.
- For each step, the algorithm plots the 8 symmetric points of the circle using the ‘plot_pixel_safe‘ subroutine.

Speed Control

Speed control is handled by adjusting the delay between game loop iterations. The delay is decreased to speed up the game or increased to slow it down.

```
speed_up:
    addi s9, s9, -10  # Decrease delay, increase speed
    j check_bounds

speed_down:
    addi s9, s9, 10   # Increase delay, decrease speed
    j check_bounds
```

Explanation:

- Pressing ‘Z‘ reduces the delay by 10 units, making the ball move faster.
- Pressing ‘X‘ increases the delay by 10 units, making the ball move slower.

Exiting the Program

The program can be exited by pressing the 'Enter' key. The exit procedure is handled using the 'ecall' system call.

```
exit_program:
    li a7, 10          # Exit system call
    ecall              # Exit the program
```

Explanation:

- When the 'Enter' key is pressed, the program triggers an exit system call ('ecall' with argument '10') to terminate the program.

Game Loop

The game loop is the core of the program, repeatedly checking for input, updating the position of the ball, drawing it on the screen, and controlling the game speed.

```
game_loop:
    # Process keyboard input
    # Move ball or adjust speed based on key press
    # Update position and redraw ball
    # Add delay and repeat
    j game_loop
```

Explanation:

- The 'game_loop' is a continuous loop that checks the keyboard input, updates the ball's position, redraws the ball at the new position, and then introduces a delay before repeating the process.

2.4 Source Code

```
# Cac hang so cho man hinh va mau sac
.eqv DISPLAY_ADDRESS 0x10010000  # Dia chi bat dau cua Bitmap Display
.eqv YELLOW 0x00FFFF66
.eqv BACKGROUND 0x00000000

# Cac hang so cho phim
.eqv KEYBOARD_CONTROL 0xFFFF0000
.eqv KEYBOARD_DATA 0xFFFF0004
.eqv KEY_A 0x61  # sang trai
.eqv KEY_D 0x64  # sang phai
.eqv KEY_S 0x73  # xuong duoi
.eqv KEY_W 0x77  # len tren
.eqv KEY_Z 0x7a  # giam toc
.eqv KEY_X 0x78  # tang toc
.eqv KEY_ENTER 0x0a  # thoat

# Hang so khac
.eqv MOVE_DISTANCE 10  # Khoang cach di chuyen
.eqv CIRCLE_RADIUS 20  # Ban kinh hinh tron

.data
```

```
    coords: .space 512  # Mang luu toa do cac diem

.text
.globl main

main:
    # Khoi tao cac gia tri
    li s0, 256      # x = 256 (toa do x ban dau)
    li s1, 256      # y = 256 (toa do y ban dau)
    li s2, 20       # R = 20 (ban kinh)
    li s3, 512      # Chieu rong man hinh
    li s4, 512      # Chieu cao man hinh
    li s5, YELLOW   # Mau hinh tron
    li s6, MOVE_DISTANCE  # Khoang cach di chuyen
    li s7, 0        # dx (thay doi theo x)
    li s8, 0        # dy (thay doi theo y)
    li s9, 70       # Toc do delay

    # Ve hinh tron ban dau
    jal draw_circle

game_loop:
    # Doc phim
    li t0, KEYBOARD_CONTROL
    lw t1, 0(t0)
    beqz t1, check_bounds

    # Doc ma phim
    li t0, KEYBOARD_DATA
    lw t2, 0(t0)

    # Xu ly phim
    li t3, KEY_A
    beq t2, t3, move_left
    li t3, KEY_D
    beq t2, t3, move_right
    li t3, KEY_W
    beq t2, t3, move_up
    li t3, KEY_S
    beq t2, t3, move_down
    li t3, KEY_Z
    beq t2, t3, speed_up
    li t3, KEY_X
    beq t2, t3, speed_down
    li t3, KEY_ENTER
    beq t2, t3, exit_program
    j check_bounds

move_left:
    neg s7, s6      # dx = -MOVE_DISTANCE
    li s8, 0        # dy = 0
    j check_bounds

move_right:
    mv s7, s6       # dx = MOVE_DISTANCE
    li s8, 0        # dy = 0
    j check_bounds

move_up:
    li s7, 0        # dx = 0
```

```
    neg s8, s6      # dy = -MOVE_DISTANCE
    j check_bounds

move_down:
    li s7, 0        # dx = 0
    mv s8, s6       # dy = MOVE_DISTANCE
    j check_bounds

speed_up:
    addi s9, s9, -10  # Giam delay = tang toc
    j check_bounds

speed_down:
    addi s9, s9, 10   # Tang delay = giam toc
    j check_bounds

check_bounds:
    # Kiem tra bien phai
    add t0, s0, s2    # x + R
    add t0, t0, s7    # + dx
    bge t0, s3, reverse_x

    # Kiem tra bien trai
    sub t0, s0, s2    # x - R
    add t0, t0, s7    # + dx
    bltz t0, reverse_x

    # Kiem tra bien tren
    sub t0, s1, s2    # y - R
    add t0, t0, s8    # + dy
    bltz t0, reverse_y

    # Kiem tra bien duoi
    add t0, s1, s2    # y + R
    add t0, t0, s8    # + dy
    bge t0, s4, reverse_y

    j update_position

reverse_x:
    neg s7, s7        # Doi chieu dx
    j update_position

reverse_y:
    neg s8, s8        # Doi chieu dy
    j update_position

update_position:
    # Xoa hinh tron cu
    li s5, BACKGROUND
    jal draw_circle

    # Cap nhat vi tri
    add s0, s0, s7    # x += dx
    add s1, s1, s8    # y += dy

    # Ve hinh tron moi
    li s5, YELLOW
    jal draw_circle

```

```
    # Delay
    mv a0, s9
    li a7, 32
    ecall

    j game_loop

draw_circle:
    # Luu ra
    addi sp, sp, -4
    sw ra, 0(sp)

    # Bresenham circle algorithm
    li t0, 0            # x = 0
    mv t1, s2           # y = r
    li t2, 3            # d = 3
    sub t3, t2, s2      # d = 3 - 2r
    sub t3, t3, s2

draw_loop:
    bgt t0, t1, draw_end    # while x <= y

    # Plot 8 points
    # Tren phai
    add a0, s0, t0      # x0 + x
    add a1, s1, t1      # y0 + y
    jal plot_pixel_safe

    add a0, s0, t1      # x0 + y
    add a1, s1, t0      # y0 + x
    jal plot_pixel_safe

    # Tren trai
    sub a0, s0, t0      # x0 - x
    add a1, s1, t1      # y0 + y
    jal plot_pixel_safe

    sub a0, s0, t1      # x0 - y
    add a1, s1, t0      # y0 + x
    jal plot_pixel_safe

    # Duoi phai
    add a0, s0, t0      # x0 + x
    sub a1, s1, t1      # y0 - y
    jal plot_pixel_safe

    add a0, s0, t1      # x0 + y
    sub a1, s1, t0      # y0 - x
    jal plot_pixel_safe

    # Duoi trai
    sub a0, s0, t0      # x0 - x
    sub a1, s1, t1      # y0 - y
    jal plot_pixel_safe

    sub a0, s0, t1      # x0 - y
    sub a1, s1, t0      # y0 - x
    jal plot_pixel_safe

    # Cap nhat cac bien
```

```
    bgez t3, adjust_d_positive

    # d < 0
    slli t4, t0, 2     # 4x
    addi t4, t4, 6     # 4x + 6
    add t3, t3, t4     # d += 4x + 6
    j continue_draw

adjust_d_positive:
    # d >= 0
    slli t4, t0, 2     # 4x
    sub t4, t4, t1     # 4x - y
    sub t4, t4, t1     # 4x - 2y
    sub t4, t4, t1     # 4x - 3y
    sub t4, t4, t1     # 4x - 4y
    addi t4, t4, 10    # 4x - 4y + 10
    add t3, t3, t4     # d += 4x - 4y + 10
    addi t1, t1, -1    # y--

continue_draw:
    addi t0, t0, 1     # x++
    j draw_loop

draw_end:
    lw ra, 0(sp)
    addi sp, sp, 4
    ret

plot_pixel_safe:
    # Kiem tra bien
    bltz a0, plot_end      # x < 0
    bge a0, s3, plot_end   # x >= width
    bltz a1, plot_end      # y < 0
    bge a1, s4, plot_end   # y >= height

    # Tinh offset
    mul t6, a1, s3     # y * width
    add t6, t6, a0     # + x
    slli t6, t6, 2     # * 4 bytes/pixel
    li t4, DISPLAY_ADDRESS
    add t6, t6, t4

    # Ve pixel
    sw s5, 0(t6)

plot_end:
    ret

exit_program:
    li a7, 10
    ecall
```

2.5 Display Results

The following images represent our results:

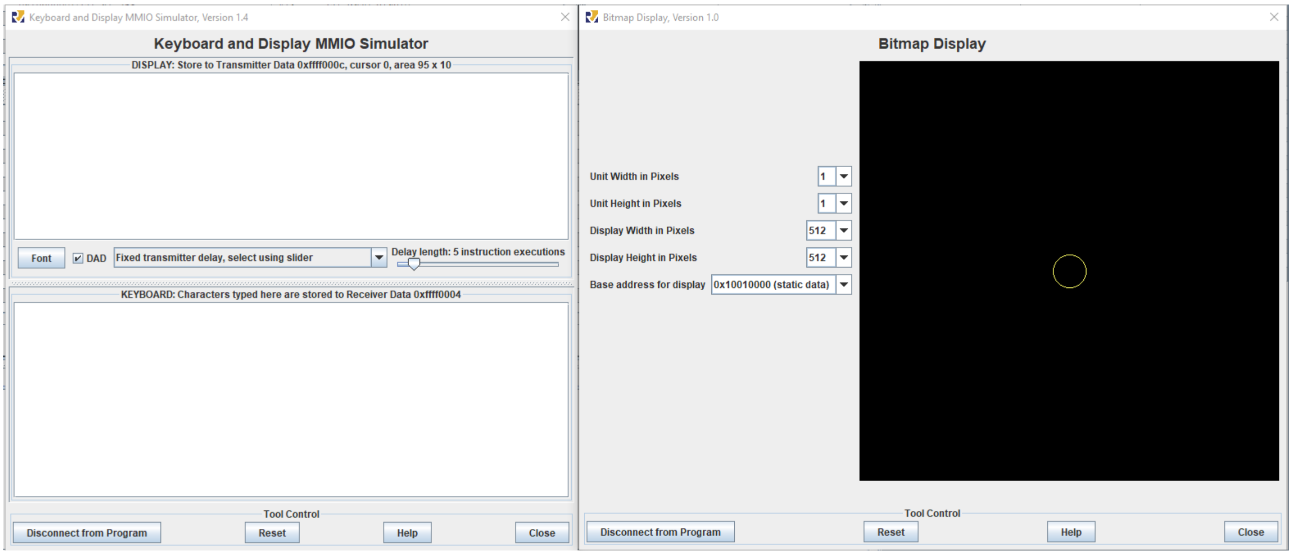

Figure 1: Start of the bitmap display

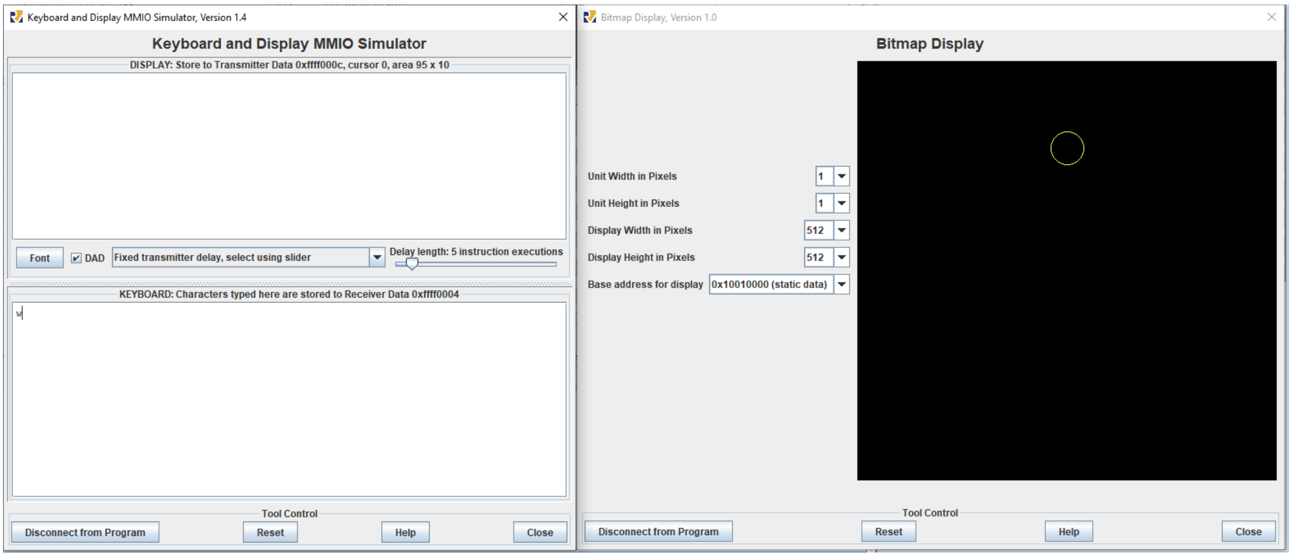

Figure 2: The ball moved up when press W or w

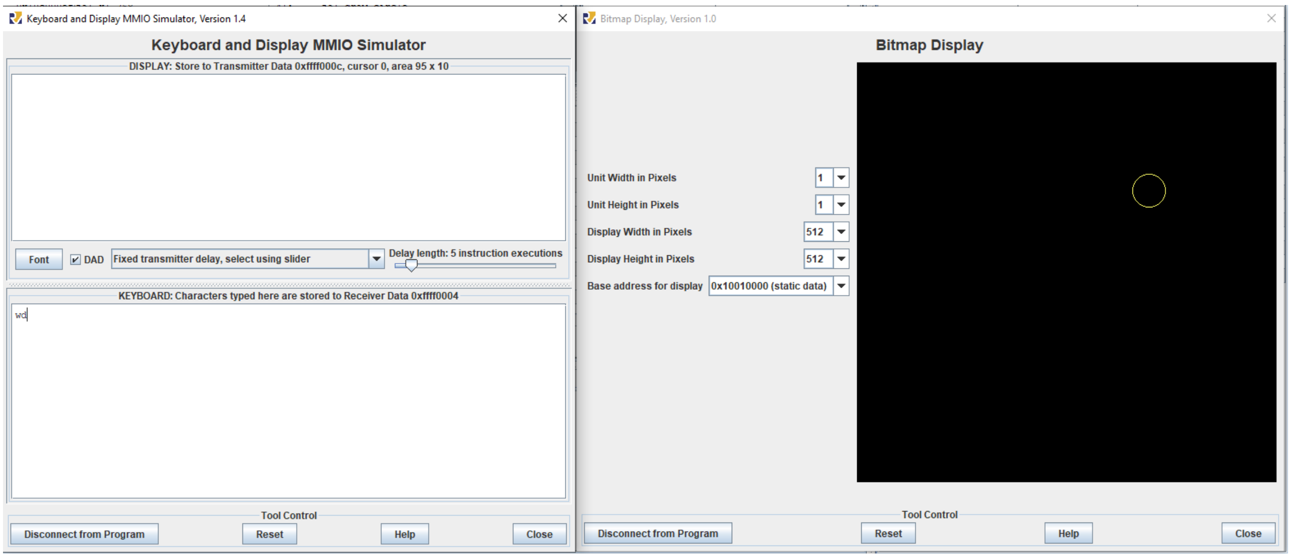

Figure 3: The ball moved right when press D or d

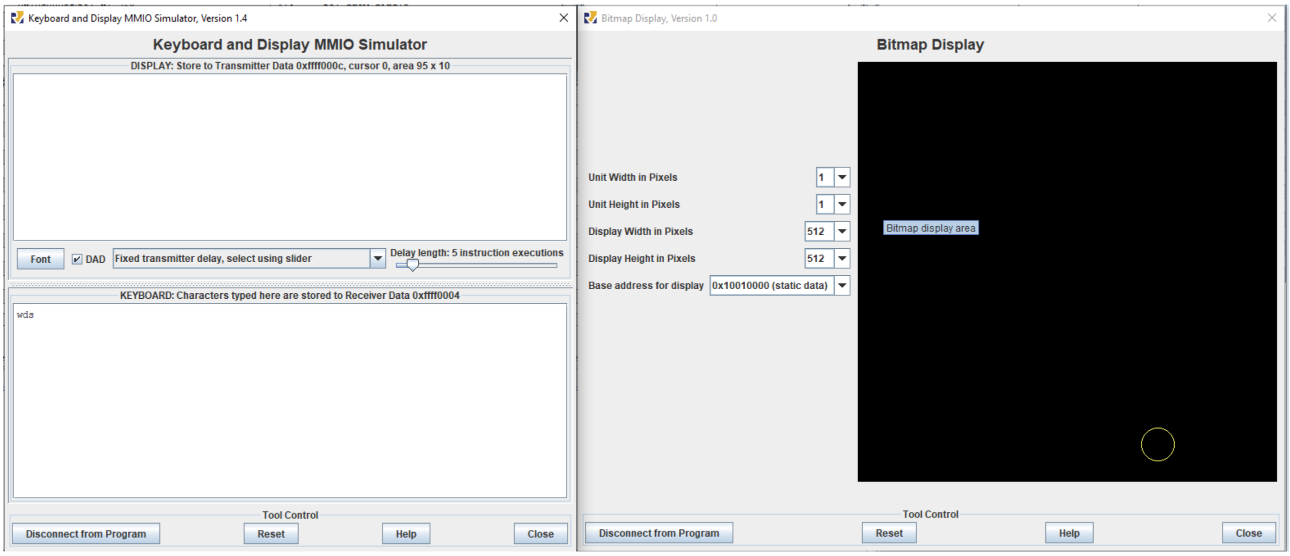

Figure 4: The ball moved down when press S or s

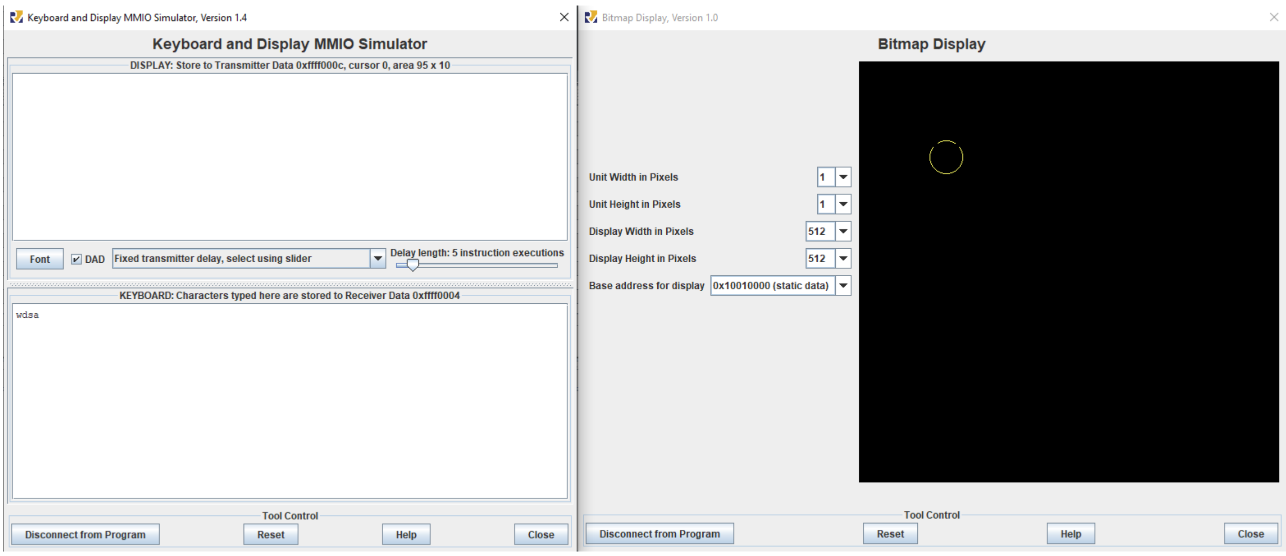

Figure 5: The ball moved left when press A or a

3 PROBLEM 9: TESTING SORTING ALGORITHMS

graphicx

3.1 Problem Statement

Create a program to read data from a text file with up to 10,000 random integers separated by spaces. The user will type in the file name, and the application will read the numbers into memory. The user may then choose one of the four sorting algorithms: Bubble Sort, Insertion Sort, Selection Sort and Quick Sort. The program will run the chosen algorithm, record the execution time, count the total number of instructions performed, and save the sorted results to an output file.

Requirement:

- The application must be able to open and read a file containing random integers up to 10,000. The numbers must be read and saved to memory as a list or array.
- The user can select one of four sorting algorithms: Bubble Sort, Insertion Sort, Selection Sort and Quick Sort.
- The application must measure and report the sorting algorithm's execution duration, as well as determine the total number of instructions performed throughout the sorting process.
- The user interface should be straightforward, allowing for file entry, algorithm selection, and easy-to-read result presentation.
- The application should handle problems when the file does not exist or is in the improper format, as well as ensuring that the numbers in the file are valid integers.

3.2 Algorithm Idea

In this problem, we implement several sorting algorithms to process an array of integers, read from a file, and sort as required. Additionally, the algorithm marks negative numbers in the array using a bitmask array, helping to quickly identify negative numbers during processing.

Initialization

- Initialise the `numbers` array, which will be used to hold the integers read from the file. Initially, the array is empty.
- Variable count: This variable keeps track of the number of elements in the array. Initially, `count` is set to zero.
- Time Variables: Variables such as `start_time` and `end_time` are initialised to track the sorting algorithm's execution time.
- Read Data from File: The data will be read into the numbers array after the file containing the integers has been opened. The application will alert the user and request a proper file name if the file is not found or if there is a reading problem.
- Initialise Sorting Variables: Sorting algorithms like Bubble Sort and Quick Sort employ variables (such array indices) that have been initialised.

Handling Keyboard Input

- Enter File Name: The application requests that the user provide the name of the file containing the numbers. The application will ask the user to enter the file name again if it doesn't exist until a legitimate file is located.
- Verify File Name Validity: The application verifies if the file exists after the user inputs the file name. If not, it will prompt the user to provide the file name once again until a legitimate file is supplied.
- Read Data from File: The integers in the file will be read and saved in the numbers array once a valid file name has been supplied. Until all of the data in the file has been processed, each integer will be added to the array.

Checking Boundaries

- Verify for Negative Values: The numbers are examined for negative values while they are being read from the file. The bitmask array will be used to flag negative values for treatment at a later time. Only legitimate (non-negative) numbers are added to the numbers array, thanks to the application.
- Verify Array Indexes During Operations: When the sorting algorithm is being executed, comparisons and swaps are always carried out inside the acceptable boundaries of the array of integers. The program will halt and raise an error if any operation goes above the array's limits.
- Array Index Boundaries: When using sorting algorithms (such as Bubble Sort and Quick Sort), the software makes sure that array indices don't go beyond the array's size to avoid out-of-bounds access.

Sorting Algorithm

- **Bubble Sort:** A simple algorithm that alternates neighbouring items in a list if they are out of order after periodically comparing them. This method is carried out again until the array is sorted and no swaps are required. The largest unsorted element "bubbles" up to its proper location after each pass. In the worst scenario, Bubble Sort's temporal complexity is $O(n^2)$.
- **Selection Sort:**From the unsorted section, the algorithm continually chooses the smallest (or biggest) element and replaces it with the first unsorted element. With a temporal complexity of $O(n^2)$ in any scenario, this operation keeps going until the full array is sorted.
- **Quick Sort:** This divide-and-conquer method divides the array into two sub-arrays after choosing a pivot element. Until the array is sorted, the same process is used recursively to the left and right sub-arrays. The temporal complexity of Quick Sort is $O(n\log n)$ on average, but it can be $O(n^2)$ in the worst situation (with a faulty pivot).
- **Insertion Sort:** One element at a time, this algorithm creates the sorted array. With a time complexity of $O(n^2)$ in the worst case and $O(n)$ in the best, it selects the subsequent element from the unsorted section and places it in the appropriate location in the sorted section.

Results of the Output

- Print Sorted Array: To allow the user to confirm the outcome, the software will print the sorted array after the sorting procedure is complete.
- Execution Time: To notify the user, the sorting algorithm's execution time will be computed and shown.
- Save Results: The sorted results can be stored for later use in a new file if necessary.

Exception Handling

- File Handling Errors: The application will alert the user and ask for a suitable file name if an error occurs when accessing the file or reading data from it (such as file not found).
- Input Validation Errors: The application will prompt the user to re-enter the data until valid data is entered if the user enters invalid data.

3.3 Implementation Pipeline

Input File Handling:

- The program prompts the user to enter a filename using the message "Enter filename: "
- A 256-byte buffer (input_filename) is allocated to store the input filename
- A newline character removal process is implemented to clean the filename input
- File opening is performed using the RISC-V syscall 1024 (open file)
- Error handling is integrated with a specific error message if file opening fails

```
# Prompt for filename
li a7, 4
la a0, msg_prompt_input
ecall

# Read filename
li a7, 8
la a0, input_filename
li a1, 256
ecall

# Remove newline from filename
remove_newline_from_filename:
    lb t1, 0(t0)
    beqz t1, open_input_file
    li t2, 10
    beq t1, t2, replace_null
    addi t0, t0, 1
    j remove_newline_from_filename
```

Number Reading Strategy

- The (read_numbers) function implements a sophisticated number parsing algorithm

- Supports reading both positive and negative integers
- Handles different input scenarios:
  - Recognizes numbers separated by spaces or newline characters
  - Detects negative numbers by identifying the '-' sign
  - Converts ASCII characters to integer values
  - Builds numbers digit by digit
- Stores parsed numbers in a pre-allocated numbers array
- Maintains a count variable to track the total number of integers read

```
read_loop:
    # Read one character from file
    li a7, 63           # ReadFile syscall
    lw a0, fd           # File descriptor
    la a1, file_read_buffer     # Buffer address
    li a2, 1            # Read one character
    ecall

    # Check for negative number
    li t3, 45           # ASCII for '-'
    bne t2, t3, not_char_minus
    beqz t1, set_negative   # Set negative only at number start

    # Convert ASCII to number
    addi t2, t2, -48    # Convert ASCII to digit
    li t3, 10
    mul t0, t0, t3      # Current number * 10
    add t0, t0, t2      # Add new digit
```

Sorting Algorithms Implementation

The program provides four distinct sorting algorithms, each implemented with a similar structural approach:

- **Bubble Sort:**
  - Implements a classic bubble sort algorithm
  - Outer and inner loop structure for comparing and swapping adjacent elements
  - Time complexity: $O(n^2)$
  - Iterates through the array multiple times, progressively "bubbling up" larger elements
- **Insertion Sort:**
  - Uses an in-place insertion sort strategy
  - Builds the sorted portion of the array incrementally
  - Time complexity: $O(n^2)$
  - Efficiently handles partially sorted arrays
  - Shifts elements to create space for inserting the current element
- **Selection Sort:**
  - Implements selection sort with a two-loop mechanism
  - Finds the minimum element in each iteration and places it in its correct position
  - Time complexity: $O(n^2)$
  - Consistently divides the array into sorted and unsorted regions

- **Quick Sort:**
  - Utilizes the Quicksort algorithm with a recursive approach
  - Employs a partitioning strategy to divide the array
  - Chooses the last element as the pivot
  - Recursively sorts sub-arrays on both sides of the pivot
  - Average time complexity: O(n log n)
  - More efficient for larger datasets compared to other implemented algorithms

Performance Measurement

- **Execution Time Tracking**
  - Uses syscall 30 to capture start and end timestamps
  - Calculates execution time in milliseconds
  - Prints execution time after each sorting operation

```
get_time:
    li a7, 30          # GetTime syscall
    ecall
    ret

print_time:
    # Calculate execution time
    la t0, start_time
    lw t1, 0(t0)
    la t0, end_time
    lw t2, 0(t0)
    sub t3, t2, t1     # Execution time

    # Print execution time
    li a7, 4
    la a0, msg_execution_time
    ecall

    li a7, 1
    mv a0, t3
    ecall
```

Negative Number Handling:

- **Bit Masking Technique:** A unique feature of the implementation is the negative number tracking
  - Implements a custom bit masking approach for tracking negative numbers
  - Allocates a separate `neg_bitmask` to mark negative number positions
  - Efficiently uses bitwise operations to flag negative integers

```
flag_negative_numbers:
    mv s0, a0          # Array address
    mv s1, a1          # Array size
    li s2, 0           # Counter

flag_loop:
    bge s2, s1, flag_done
```

```

    # Get current number
    slli t0, s2, 2
    add t0, s0, t0
    lw t1, 0(t0)      # Get number

    # Skip if positive
    bgez t1, skip_flag

    # Calculate bit position in bitmask
    mv t0, s2          # Copy index
    srai t1, t0, 3     # Byte offset = index / 8
    andi t2, t0, 0x7   # Bit position = index % 8
    li t3, 1
    sll t3, t3, t2     # Shift 1 to exact bit position

    # Set bit in bitmask
    la t4, neg_bitmask
    add t4, t4, t1     # Add byte offset
    lb t5, 0(t4)       # Get current byte
    or t5, t5, t3      # Set bit
    sb t5, 0(t4)       # Save updated byte

skip_flag:
    addi s2, s2, 1
    j flag_loop
```

Output Processing

- **File Writing Mechanism**
  - Creates an output file at a predefined path (`/Users//Documents/Prj/Final/output.txt`)
  - Writes sorted numbers to the file
  - Includes error handling for file writing operations
  - Converts integers back to string representation for writing
  - Handles spacing between numbers in the output file

```
write_results:
    # Open output file
    li a7, 1024       # Open file syscall
    la a0, output_filename
    li a1, 1          # Write mode
    li a2, 0x1ff      # File permissions
    ecall

    # Write numbers to file
    li s0, 0          # Counter
    lw s1, count      # Total numbers
    la s2, numbers    # Array address

write_loop:
    bge s0, s1, write_done

    # Get current number
    slli t0, s0, 2
    add t1, s2, t0
    lw t2, 0(t1)       # Get number
```

```
    # Convert number to string
    la a0, buffer_number
    mv a1, t2
    jal number_to_string

    # Write number to file
    li a7, 64           # WriteFile syscall
    lw a0, out_fd
    la a1, buffer_number
    mv a2, t3           # Exact length
    ecall
```

Auxiliary Functions

Several utility functions support the main implementation:

- `number_to_string`: Converts integers to string representation
- `str_reverse`: Reverses strings in-place
- `flag_negative_numbers`: Marks negative number positions
- Error handling and input validation functions

```
number_to_string:
    # a0 = buffer address
    # a1 = number to convert
    mv s0, a0          # Save buffer address
    mv s1, a1          # Save number to convert
    li s2, 0           # Length counter
    li s3, 0           # Negative flag

    # Handle zero case
    bnez s1, check_sign
    li t0, 48          # ASCII '0'
    sb t0, 0(s0)
    li a0, 1
    j num_to_str_done

check_sign:
    # Check for negative number
    bgez s1, convert_digits
    li s3, 1           # Set negative flag
    neg s1, s1         # Make positive

convert_digits:
    # Convert digits in reverse order
    mv t0, s0
digit_loop:
    beqz s1, finalize_string
    li t1, 10
    rem t2, s1, t1     # Get last digit
    div s1, s1, t1
    addi t2, t2, 48    # Convert to ASCII
    sb t2, 0(t0)
    addi t0, t0, 1
    addi s2, s2, 1
    j digit_loop
str_reverse:
```

```
    # a0 = start address
    # a1 = end address
    bge a0, a1, str_rev_done

    # Swap characters
    lb t0, 0(a0)
    lb t1, 0(a1)
    sb t1, 0(a0)
    sb t0, 0(a1)

    # Move pointers
    addi a0, a0, 1
    addi a1, a1, -1
    j str_reverse

str_rev_done:
    ret
```

Memory Management

- Efficient use of stack for storing temporary variables
- Careful register allocation and preservation
- Minimizes memory overhead during sorting operations

```
quick_sort_logic:
    # Allocate stack space and save registers
    addi sp, sp, -24
    sw ra, 20(sp)      # Return address
    sw s0, 16(sp)      # Saved register 0
    sw s1, 12(sp)      # Saved register 1
    sw s2, 8(sp)       # Saved register 2
    sw s3, 4(sp)       # Saved register 3
    sw s4, 0(sp)       # Saved register 4

    # Function body...

    # Restore registers and deallocate stack
    lw ra, 20(sp)
    lw s0, 16(sp)
    lw s1, 12(sp)
    lw s2, 8(sp)
    lw s3, 4(sp)
    lw s4, 0(sp)
    addi sp, sp, 24
    ret
```

User Interaction

- Provides a menu-driven interface for sorting algorithm selection
- Allows multiple sorting attempts on the same input data
- Offers an option to exit the program

```
menu_loop:
    # Display menu options
    li a7, 4
```

```
    la a0, menu
    ecall

    # Menu string content
    # "\nUser select sorting algorithm:
    #  1. Bubble Sort
    #  2. Insertion Sort
    #  3. Selection Sort
    #  4. Quick Sort
    #  5. Close
    #  Choice: "

    # Read user choice
    li a7, 5
    ecall

    # Branch to selected sorting algorithm
    li t0, 1
    beq a0, t0, bubble_sort_array
    li t0, 2
    beq a0, t0, insertion_sort_array
    li t0, 3
    beq a0, t0, selection_sort_array
    li t0, 4
    beq a0, t0, quick_sort_array
    li t0, 5
    beq a0, t0, exit

    # Invalid choice handling implicitly falls through to menu
    j menu_loop

# Exit program
exit:
    li a7, 10         # Exit syscall
    ecall
file_error_open:
    # Display error message for file opening failure
    li a7, 4
    la a0, error_msg   # "\nError opening file\n"
    ecall

    # Return to main menu or exit
    j exit

msg_file_error_openor:
    # Error message for file writing failure
    li a7, 4
    la a0, msg_file_error_open
    ecall

    # Cleanup and return
    lw ra, 12(sp)
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw s2, 0(sp)
    addi sp, sp, 16
    ret
```

3.4 Source Code

```
.data
    # Phan du lieu hien co
    numbers:    .space  80000
    input_buffer_size:   .word   80000
    count:      .word   0

    # Them mang bitmask cho so am (1 bit moi so)
    # Voi 80000 byte so (20000 so nguyen), chung ta can 20000 bit = 2500 byte
    neg_bitmask:   .space  2500

    input_filename:   .space  256
    file_read_buffer:    .space  1024
    msg_prompt_input:     .string "Enter filename: "
    error_msg:  .string "\nError opening file\n"
    menu:       .string "\nUser select sorting algorithm:\n1. Bubble Sort\n2. Insertion Sort\n3. Selection Sort\n4. Quick Sort\n5.Close\nChoice: "

    fd:         .word   0
    newline:    .string "\n"
    space:      .string " "
    start_time: .word   0
    end_time:   .word   0

    msg_execution_time:   .string "\nExecution time (ms): "

    # Du lieu moi cho file output
    output_filename:    .string "/Users/truonglinhduyen/Documents/Prj/Final/ouuuu.txt"
    out_fd:     .word   0
    buffer_number:     .space  12
    msg_file_error_open:  .string "\nError writing to output file\n"
    char_minus:       .string "-"
    # Them buffer tam cho sap xep tron
    tmp_sort_buffer:   .space  80000
.text
.globl main
main:
    # In ra msg_prompt_input
    li a7, 4
    la a0, msg_prompt_input
    ecall
    # Doc input_filename
    li a7, 8
    la a0, input_filename
    li a1, 256
    ecall
    # Loai bo newline khoi input_filename
    la t0, input_filename
remove_newline_from_filename:
    lb t1, 0(t0)
    beqz t1, open_input_file
    li t2, 10
    beq t1, t2, replace_null
    addi t0, t0, 1
    j remove_newline_from_filename
replace_null:
```

```
    sb zero, 0(t0)

open_input_file:
    li a7, 1024
    la a0, input_filename
    li a1, 0
    ecall
    bltz a0, file_error_open
    la t1, fd
    sw a0, 0(t1)
    jal read_numbers

menu_loop:
    # Hien thi menu
    li a7, 4
    la a0, menu
    ecall

    # Doc lua chon
    li a7, 5
    ecall

    # Nhanh den lua chon
    li t0, 1
    beq a0, t0, bubble_sort_array
    li t0, 2
    beq a0, t0, insertion_sort_array
    li t0, 3
    beq a0, t0, selection_sort_array
    li t0, 4
    beq a0, t0, quick_sort_array
    li t0,5
    beq a0, t0, exit
    j exit
file_error_open:
    # In ra thong bao loi
    li a7, 4
    la a0, error_msg
    ecall
    j exit
read_numbers:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    sw s1, 4(sp)
    sw s2, 0(sp)

    # Reset count
    la t1, count        # Tai dia chi cua count
    sw zero, 0(t1)      # Luu gia tri zero vao count

    # Khoi tao bien
    li t0, 0            # So hien tai dang duoc xay dung
    li t1, 0            # Co cho biet dang trong so
    li t6, 0            # Co danh dau (0 = duong, 1 = am)

read_loop:
    # Doc mot ky tu tu file
    li a7, 63           # Syscall ReadFile
    lw a0, fd           # Mo ta file
```

```
    la a1, file_read_buffer     # Dia chi buffer
    li a2, 1            # Doc mot ky tu
    ecall

    # Kiem tra neu cuoi file
    beqz a0, read_done

    # Tai ky tu
    lb t2, 0(a1)

    # Kiem tra dau tru
    li t3, 45           # ASCII cho '-'
    bne t2, t3, not_char_minus
    beqz t1, set_negative   # Chi dat am neu o dau so
    j read_loop

set_negative:
    li t6, 1            # Dat co danh dau la am
    li t1, 1            # Dat co trong so
    j read_loop

not_char_minus:
    # Kiem tra neu la dau cach hoac newline
    li t3, 32           # Space
    beq t2, t3, save_number
    li t3, 10           # Newline
    beq t2, t3, save_number

    # Chuyen doi ASCII thanh so va cong vao so hien tai
    addi t2, t2, -48    # Chuyen doi ASCII thanh so
    li t3, 10
    mul t0, t0, t3      # So hien tai * 10
    add t0, t0, t2      # Cong vao chu so moi
    li t1, 1            # Dat co trong so
    j read_loop

save_number:
    beqz t1, read_loop  # Neu khong trong so, tiep tuc

    # Ap dung dau neu la am
    beqz t6, save_positive
    neg t0, t0          # Doi dau so neu co am duoc dat

save_positive:
    # Luu so vao mang
    la t3, count        # Tai dia chi cua count
    lw t3, 0(t3)        # Tai gia tri count
    slli t4, t3, 2      # t4 = count * 4
    la t5, numbers
    add t5, t5, t4
    sw t0, 0(t5)        # Luu so

    # Tang count
    addi t3, t3, 1
    la t4, count        # Tai dia chi cua count
    sw t3, 0(t4)        # Luu count moi

    # Reset cho so tiep theo
    li t0, 0            # Reset so hien tai
    li t1, 0            # Reset co trong so
```

```
    li t6, 0            # Reset co am
    j read_loop

read_done:
    # Neu chung ta dang trong so khi file ket thuc, luu no
    beqz t1, close_file

    # Ap dung dau neu la am
    beqz t6, save_last_positive
    neg t0, t0          # Doi dau so neu co am duoc dat

save_last_positive:
    la t3, count        # Tai dia chi cua count
    lw t3, 0(t3)        # Tai gia tri count
    slli t4, t3, 2
    la t5, numbers
    add t5, t5, t4
    sw t0, 0(t5)
    addi t3, t3, 1
    la t4, count        # Tai dia chi cua count
    sw t3, 0(t4)        # Luu count moi

close_file:
    # Dong file
    li a7, 57           # Syscall Close file
    lw a0, fd
    ecall

    lw ra, 12(sp)
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw s2, 0(sp)
    addi sp, sp, 16
    ret

flag_negative_numbers:
    # a0 = dia chi mang
    # a1 = size
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    sw s1, 4(sp)
    sw s2, 0(sp)

    mv s0, a0           # s0 = dia chi mang
    mv s1, a1           # s1 = size
    li s2, 0            # s2 = bo dem

flag_loop:
    bge s2, s1, flag_done

    # Tai so hien tai
    slli t0, s2, 2      # t0 = bo dem * 4
    add t0, s0, t0
    lw t1, 0(t0)       # Tai so

    # Bo qua neu duong
    bgez t1, skip_flag

    # Tinh toan byte va vi tri bit trong bitmask
```

```
    mv t0, s2           # Sao chep chi so
    srai t1, t0, 3      # Byte offset = chi so / 8
    andi t2, t0, 0x7    # Vi tri bit = chi so % 8
    li t3, 1
    sll t3, t3, t2      # Dich chuyen 1 den vi tri bit chinh xac

    # Dat bit trong bitmask
    la t4, neg_bitmask
    add t4, t4, t1      # Them byte offset
    lb t5, 0(t4)        # Tai byte hien tai
    or t5, t5, t3       # Dat bit
    sb t5, 0(t4)        # Luu byte da cap nhat

skip_flag:
    addi s2, s2, 1
    j flag_loop

flag_done:
    lw ra, 12(sp)
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw s2, 0(sp)
    addi sp, sp, 16
    ret
quick_sort_array:
    # Lay thoi gian bat dau
    jal get_time
    sw a0, start_time, t0

    # Khoi tao quicksort
    la a0, numbers
    li a1, 0
    lw a2, count
    addi a2, a2, -1
    jal quick_sort_logic

    # Danh dau cac so am
    la a0, numbers
    lw a1, count
    jal flag_negative_numbers

    # Lay thoi gian ket thuc va tinh thoi gian thuc thi
    jal get_time
    sw a0, end_time, t0
    jal print_time

    # Ghi ket qua vao file
    jal write_results
    j menu_loop

quick_sort_logic:
    # a0 = dia chi mang
    # a1 = chi so ben trai
    # a2 = chi so ben phai
    addi sp, sp, -24
    sw ra, 20(sp)
    sw s0, 16(sp)
    sw s1, 12(sp)
    sw s2, 8(sp)
    sw s3, 4(sp)
```

```
    sw s4, 0(sp)

    # Luu cac tham so
    mv s0, a0          # s0 = dia chi mang
    mv s1, a1          # s1 = ben trai
    mv s2, a2          # s2 = ben phai

    # Dieu kien dung: neu ben trai >= ben phai, thoat
    bge s1, s2, quick_sort_end

    # Goi ham partition_elements
    mv a0, s0          # dia chi mang
    mv a1, s1          # chi so ben trai
    mv a2, s2          # chi so ben phai
    jal partition_elements
    mv s3, a0          # s3 = chi so pivot

    # De quy sap xep phan ben trai
    mv a0, s0          # dia chi mang
    mv a1, s1          # chi so ben trai
    addi a2, s3, -1    # pivot - 1
    jal quick_sort_logic

    # De quy sap xep phan ben phai
    mv a0, s0          # dia chi mang
    addi a1, s3, 1     # pivot + 1
    mv a2, s2          # chi so ben phai
    jal quick_sort_logic

quick_sort_end:
    lw ra, 20(sp)
    lw s0, 16(sp)
    lw s1, 12(sp)
    lw s2, 8(sp)
    lw s3, 4(sp)
    lw s4, 0(sp)
    addi sp, sp, 24
    ret

partition_elements:
    addi sp, sp, -24
    sw ra, 20(sp)
    sw s0, 16(sp)
    sw s1, 12(sp)
    sw s2, 8(sp)
    sw s3, 4(sp)
    sw s4, 0(sp)

    mv s0, a0          # s0 = dia chi mang
    mv s1, a1          # s1 = ben trai
    mv s2, a2          # s2 = ben phai

    slli t0, s2, 2     # t0 = ben phai * 4
    add t0, s0, t0
    lw s3, 0(t0)       # s3 = gia tri pivot

    addi s4, s1, -1    # i = ben trai - 1
    mv t1, s1          # j = ben trai

partition_loop_elements:
```

```
    bge t1, s2, partition_elements_done

    # Tai phan tu hien tai
    slli t0, t1, 2
    add t0, s0, t0
    lw t2, 0(t0)        # t2 = arr[j]

    # So sanh voi pivot
    bgt t2, s3, skip_swap

    # Tang i
    addi s4, s4, 1

    # Doi phan tu neu i != j
    slli t0, s4, 2
    add t0, s0, t0      # Dia chi cua arr[i]
    slli t3, t1, 2
    add t3, s0, t3      # Dia chi cua arr[j]

    lw t4, 0(t0)        # t4 = arr[i]
    lw t5, 0(t3)        # t5 = arr[j]
    sw t5, 0(t0)        # arr[i] = arr[j]
    sw t4, 0(t3)        # arr[j] = arr[i]

skip_swap:
    addi t1, t1, 1      # j++
    j partition_loop_elements

partition_elements_done:
    # Dat pivot vao vi tri cuoi cung
    addi s4, s4, 1      # i++

    # Doi pivot voi phan tu tai i
    slli t0, s4, 2
    add t0, s0, t0      # Dia chi cua arr[i]
    slli t1, s2, 2
    add t1, s0, t1      # Dia chi cua arr[right]

    lw t2, 0(t0)        # t2 = arr[i]
    lw t3, 0(t1)        # t3 = arr[right]
    sw t3, 0(t0)        # arr[i] = arr[right]
    sw t2, 0(t1)        # arr[right] = arr[i]

    # Tra ve chi so pivot
    mv a0, s4

    lw ra, 20(sp)
    lw s0, 16(sp)
    lw s1, 12(sp)
    lw s2, 8(sp)
    lw s3, 4(sp)
    lw s4, 0(sp)
    addi sp, sp, 24
    ret
##################
# buble_sort
##################
bubble_sort_array:
    # Lay thoi gian bat dau
    jal get_time
```

```
    sw a0, start_time, t0

    # Thuc hien bubble sort
    la a0, numbers     # Tai dia chi mang
    lw a1, count       # Tai kich thuoc mang
    jal bubble_sort_core
        # Danh dau cac so am
    la a0, numbers
    lw a1, count
    jal flag_negative_numbers
    # Lay thoi gian ket thuc va tinh thoi gian thuc thi
    jal get_time
    sw a0, end_time, t0
    jal print_time

    # Ghi ket qua vao file
    jal write_results
    j exit
##################
# insertion_sort_array:
##################
insertion_sort_array:
    # Lay thoi gian bat dau
    jal get_time
    sw a0, start_time, t0

    # Thuc hien insertion sort
    la a0, numbers     # Tai dia chi mang
    lw a1, count       # Tai kich thuoc mang
    jal insertion_sort_array_impl
        # Danh dau cac so am
    la a0, numbers
    lw a1, count
    jal flag_negative_numbers
    # Lay thoi gian ket thuc va tinh thoi gian thuc thi
    jal get_time
    sw a0, end_time, t0
    jal print_time

    # Ghi ket qua vao file
    jal write_results
    j exit
##################
# selection_sort_array:
##################
selection_sort_array:
    # Lay thoi gian bat dau
    jal get_time
    sw a0, start_time, t0

    # Thuc hien selection sort
    la a0, numbers     # Tai dia chi mang
    lw a1, count       # Tai kich thuoc mang
    jal selection_sort_array_impl

    # Danh dau cac so am
    la a0, numbers
    lw a1, count
    jal flag_negative_numbers

```

```
    # Lay thoi gian ket thuc va tinh thoi gian thuc thi
    jal get_time
    sw a0, end_time, t0
    jal print_time

    # Ghi ket qua vao file
    jal write_results
    j exit

bubble_sort_core:
    # a0 = dia chi mang
    # a1 = kich thuoc
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    sw s1, 4(sp)
    sw s2, 0(sp)

    mv s0, a0           # s0 = dia chi mang
    mv s1, a1           # s1 = kich thuoc
    li s2, 0            # s2 = i

outer_loop_bubble_sort:
    bge s2, s1, bubble_done
    li t0, 0            # j = 0

inner_loop_bubble_sort:
    sub t1, s1, s2
    addi t1, t1, -1
    bge t0, t1, inner_done_bubble_sort

    # So sanh cac phan tu ke tiep nhau
    slli t2, t0, 2
    add t2, s0, t2
    lw t3, 0(t2)       # arr[j]
    lw t4, 4(t2)       # arr[j+1]

    ble t3, t4, no_swap_bubble_sort

    # Hoan doi cac phan tu
    sw t4, 0(t2)
    sw t3, 4(t2)

no_swap_bubble_sort:
    addi t0, t0, 1
    j inner_loop_bubble_sort

inner_done_bubble_sort:
    addi s2, s2, 1
    j outer_loop_bubble_sort

bubble_done:
    lw ra, 12(sp)
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw s2, 0(sp)
    addi sp, sp, 16
    ret

insertion_sort_array_impl:
```

```
    # a0 = dia chi mang
    # a1 = kich thuoc
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    sw s1, 4(sp)
    sw s2, 0(sp)

    mv s0, a0           # s0 = dia chi mang
    mv s1, a1           # s1 = kich thuoc
    li s2, 1            # s2 = i = 1

outer_loop_insertion:
    bge s2, s1, insertion_done

    # Lay phan tu hien tai
    slli t0, s2, 2      # t0 = i * 4
    add t0, s0, t0
    lw t1, 0(t0)       # key = arr[i]
    addi t2, s2, -1     # j = i-1

inner_loop_insertion:
    bltz t2, inner_done_insertion    # neu j < 0, thoat

    # So sanh cac phan tu
    slli t3, t2, 2
    add t3, s0, t3
    lw t4, 0(t3)       # arr[j]

    ble t4, t1, inner_done_insertion

    # Di chuyen phan tu
    sw t4, 4(t3)       # arr[j+1] = arr[j]

    addi t2, t2, -1     # j--
    j inner_loop_insertion

inner_done_insertion:
    # Dat key vao vi tri chinh xac
    addi t2, t2, 1
    slli t3, t2, 2
    add t3, s0, t3
    sw t1, 0(t3)

    addi s2, s2, 1      # i++
    j outer_loop_insertion

insertion_done:
    lw ra, 12(sp)
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw s2, 0(sp)
    addi sp, sp, 16
    ret

selection_sort_array_impl:
    # a0 = dia chi mang
    # a1 = kich thuoc
    addi sp, sp, -16
    sw ra, 12(sp)
```

```
    sw s0, 8(sp)
    sw s1, 4(sp)
    sw s2, 0(sp)

    mv s0, a0           # s0 = dia chi mang
    mv s1, a1           # s1 = kich thuoc
    li s2, 0            # s2 = i

outer_loop_selection:
    addi t0, s1, -1
    bge s2, t0, selection_done

    mv t1, s2           # min_idx = i
    addi t2, s2, 1      # j = i + 1

inner_loop_selection:
    bge t2, s1, inner_done_selection

    # So sanh cac phan tu
    slli t3, t2, 2
    add t3, s0, t3
    lw t4, 0(t3)       # arr[j]

    slli t5, t1, 2
    add t5, s0, t5
    lw t6, 0(t5)       # arr[min_idx]

    bge t4, t6, no_update_min
    mv t1, t2           # Cap nhat min_idx

no_update_min:
    addi t2, t2, 1
    j inner_loop_selection

inner_done_selection:
    # Hoan doi cac phan tu neu can
    beq t1, s2, no_swap_selection

    slli t2, s2, 2
    add t2, s0, t2
    lw t3, 0(t2)       # temp = arr[i]

    slli t4, t1, 2
    add t4, s0, t4
    lw t5, 0(t4)       # arr[min_idx]

    sw t5, 0(t2)       # arr[i] = arr[min_idx]
    sw t3, 0(t4)       # arr[min_idx] = temp

no_swap_selection:
    addi s2, s2, 1
    j outer_loop_selection

selection_done:
    lw ra, 12(sp)
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw s2, 0(sp)
    addi sp, sp, 16
    ret
```

```

parse_loop:
    bge s2, s0, read_loop  # Neu da parse het buffer, doc tiep

    # Doc ky tu
    add t0, s1, s2
    lb t1, 0(t0)

    # Kiem tra xem co phai space khong
    li t2, 32          # ASCII cho space
    beq t1, t2, next_char

    # Chuyen doi ASCII sang so
    addi t1, t1, -48  # Chuyen ASCII thanh so

    # Luu so vao mang numbers
    lw t3, count
    slli t4, t3, 2    # t4 = count * 4 (de tinh offset)
    la t5, numbers
    add t5, t5, t4
    sw t1, 0(t5)      # Luu so vao mang

    # Tang count
    addi t3, t3, 1
    sw t3, count, t6

next_char:
    addi s2, s2, 1
    j parse_loop

# ======================== IN THONG TIN ===========================
get_time:
    li a7, 30          # Syscall GetTime
    ecall
    ret

print_time:
    # Tinh toan va in thoi gian thuc thi
    la t0, start_time
    lw t1, 0(t0)
    la t0, end_time
    lw t2, 0(t0)
    sub t3, t2, t1    # Thoi gian thuc thi

    li a7, 4
    la a0, msg_execution_time
    ecall

    li a7, 1
    mv a0, t3
    ecall
    ret

# Ham tro giup chuyen doi so thanh chuoi
number_to_string:
    # a0 = dia chi buffer
    # a1 = so can chuyen doi
    addi sp, sp, -24
    sw ra, 20(sp)
    sw s0, 16(sp)
```

```
    sw s1, 12(sp)
    sw s2, 8(sp)
    sw s3, 4(sp)
    sw s4, 0(sp)

    mv s0, a0          # Luu dia chi buffer
    mv s1, a1          # Luu so can chuyen doi
    li s2, 0           # Khoi tao bo dem do dai
    li s3, 0           # Co danh dau so am

    # Xu ly truong hop so 0
    bnez s1, check_sign
    li t0, 48          # ASCII '0'
    sb t0, 0(s0)
    li a0, 1           # Do dai la 1
    j num_to_str_done

check_sign:
    # Kiem tra so am
    bgez s1, convert_digits
    li s3, 1                    # Dat co danh dau am
    neg s1, s1                  # Chuyen so thanh duong

convert_digits:
    # Chuyen doi cac chu so theo thu tu nguoc lai
    mv t0, s0
digit_loop:
    beqz s1, finalize_string
    li t1, 10
    rem t2, s1, t1     # Lay chu so cuoi cung
    div s1, s1, t1
    addi t2, t2, 48    # Chuyen thanh ASCII
    sb t2, 0(t0)
    addi t0, t0, 1
    addi s2, s2, 1
    j digit_loop

finalize_string:
    # Them dau tru neu la so am
    beqz s3, reverse_string
    li t1, 45          # ASCII '-'
    sb t1, 0(t0)
    addi t0, t0, 1
    addi s2, s2, 1

reverse_string:
    mv a0, s0
    addi a1, t0, -1    # Cuoi chuoi
    jal str_reverse

    mv a0, s2          # Tra ve do dai

num_to_str_done:
    lw ra, 20(sp)
    lw s0, 16(sp)
    lw s1, 12(sp)
    lw s2, 8(sp)
    lw s3, 4(sp)
    lw s4, 0(sp)
    addi sp, sp, 24
```

```
    ret

# Ham tro giup dao nguoc chuoi tai cho
str_reverse:
    # a0 = dia chi bat dau
    # a1 = dia chi ket thuc
    bge a0, a1, str_rev_done

    # Hoan doi ky tu
    lb t0, 0(a0)
    lb t1, 0(a1)
    sb t1, 0(a0)
    sb t0, 0(a1)

    # Di chuyen con tro
    addi a0, a0, 1
    addi a1, a1, -1
    j str_reverse

str_rev_done:
    ret

# ==============================================
# Ghi ket qua vao file (da cap nhat)
# ==============================================
write_results:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    sw s1, 4(sp)
    sw s2, 0(sp)

    # Mo file results.txt de ghi
    li a7, 1024       # Syscall Open file
    la a0, output_filename    # input_filename
    li a1, 1          # Chi ghi
    li a2, 0x1ff      # Quyen truy cap file (777 trong octal)
    ecall

    # Kiem tra neu mo file thanh cong
    bltz a0, msg_file_error_openor
    sw a0, out_fd, t0 # Luu mo ta file

    # Ghi so vao file
    li s0, 0          # Khoi tao bo dem
    lw s1, count      # Tai tong so luong
    la s2, numbers    # Tai dia chi mang

write_loop:
    bge s0, s1, write_done

    # Tai so hien tai
    slli t0, s0, 2
    add t1, s2, t0
    lw t2, 0(t1)       # Tai so

    # Chuyen so thanh chuoi
    la a0, buffer_number
    mv a1, t2
    jal number_to_string
```

```
    mv t3, a0           # t3 = do dai cua chuoi

    # Ghi so vao file
    li a7, 64           # Syscall WriteFile
    lw a0, out_fd
    la a1, buffer_number
    mv a2, t3           # Do dai chinh xac
    ecall

    # Ghi dau cach sau so (tru so cuoi cung)
    addi t0, s1, -1
    bge s0, t0, skip_space

    li a7, 64
    lw a0, out_fd
    la a1, space
    li a2, 1
    ecall

skip_space:
    addi s0, s0, 1
    j write_loop

write_done:
    # Ghi newline o cuoi
    li a7, 64
    lw a0, out_fd
    la a1, newline
    li a2, 1
    ecall

    # Dong file output
    li a7, 57          # Syscall Close file
    lw a0, out_fd
    ecall

    lw ra, 12(sp)
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw s2, 0(sp)
    addi sp, sp, 16
    ret

msg_file_error_openor:
    # In thong bao loi
    li a7, 4
    la a0, msg_file_error_open
    ecall

    lw ra, 12(sp)
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw s2, 0(sp)
    addi sp, sp, 16
    ret
# ======================== GHI ==========================

write_positive_numbers:
    # Tai so
    slli t0, s0, 2
```

```
    add t1, s2, t0
    lw t2, 0(t1)

    # Lay gia tri tuyet doi thu cong
    bgez t2, skip_abs
    neg t2, t2
skip_abs:

    la a0, buffer_number
    mv a1, t2
    jal number_to_string

    # Ghi so
    li a7, 64
    lw a0, out_fd
    la a1, buffer_number
    mv a2, a0         # Do dai tra ve boi number_to_string
    ecall

    # Ghi dau cach (tru so cuoi cung)
    addi t0, s1, -1
    bge s0, t0, skip_space

    li a7, 64
    lw a0, out_fd
    la a1, space
    li a2, 1
    ecall

check_negative:
    bgez s1, positive_conversion   # Neu so >= 0, bo qua xu ly so am

    # Xu ly so am
    li t0, 45         # ASCII '-'
    sb t0, 0(s0)      # Luu ky tu tru
    addi s0, s0, 1    # Di chuyen con tro buffer
    addi s2, s2, 1    # Tang do dai
    neg s1, s1        # Chuyen so thanh duong

positive_conversion:
    mv t0, s0         # Vi tri hien tai trong buffer
    mv t1, s1         # Ban sao lam viec cua so

reverse_digits:
    mv a0, s0         # Bat dau cua chuoi
    add a1, s0, s2
    addi a1, a1, -1   # Cuoi chuoi

    # Them ky tu ket thuc
    add t0, s0, s2
    sb zero, 0(t0)

    jal str_reverse

    mv a0, s2         # Tra ve do dai

exit:
    li a7, 10         # Syscall Exit
    ecall
```

3.5 Display Result

Overview of Sorting Algorithm Performance

The implementation compared four distinct sorting algorithms on a common input dataset, measuring their performance across multiple key metrics:

- Execution Time
- Sorted Output
- Computational Complexity

Comparative Analysis of Sorting Algorithms

Bubble Sort

- Execution Time: 15385
- Sorting Type: In-place sorting algorithm
- Best Use Case: Suitable for small datasets or educational purposes
- Time Complexity: $O(n^2)$
- Space Complexity: $O(1)$
- Stability: Yes
- Characteristics: Iteratively compares and swaps adjacent elements until the list is sorted

Figure 6: Bubble Sort Result

- **Total number of executed instructions**: 386,407 instructions.
- **Instruction breakdown**:
  - **R-type**: 56,704 instructions (14%).
  - **R4-type**: 0 instructions (0%).
  - **I-type**: 183,332 instructions (47%) – the largest portion, primarily including load and immediate instructions such as `lw` and `addi`.
  - **S-type**: 27,779 instructions (7%).
  - **B-type**: 68,573 instructions (17%) – related to branch instructions like `beq` and `bne` within the Bubble Sort loops.
  - **U-type**: 27,627 instructions (7%).
  - **J-type**: 22,900 instructions (5%) – jump instructions used for loop control.
- **Observations**: The temporal complexity of the Bubble Sort algorithm is $O(n^2)$, which accounts for a large percentage of branch and data-processing instructions. The high proportion of **I-type** and **B-type** instructions reflects the frequent data loading, comparison, and looping operations in the algorithm.

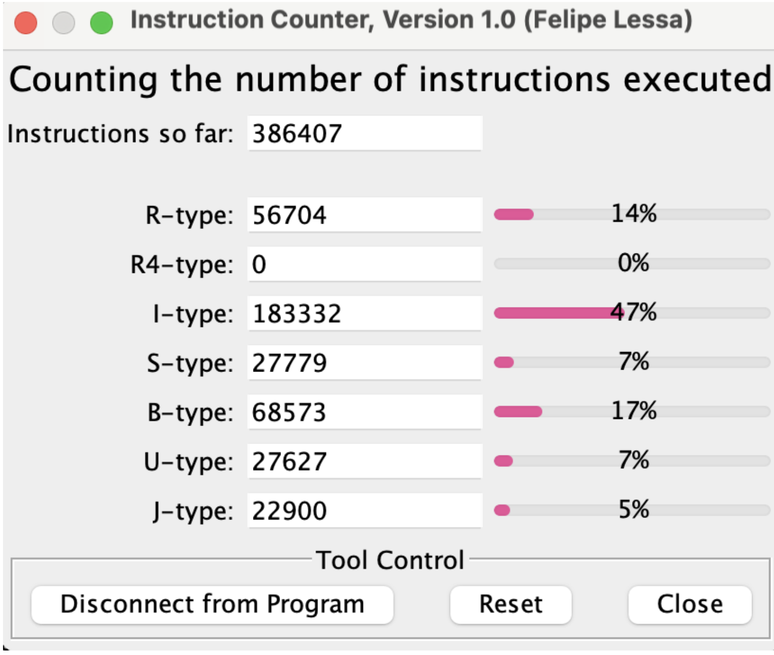

Figure 7: Instruction Count Analysis for Bubble Sort

Insertion Sort

- Execution Time: 5504
- Sorting Type: In-place sorting algorithm
- Best Use Case: Efficient for small or nearly sorted datasets
- Time Complexity: $O(n^2)$
- Space Complexity: $O(1)$
- Stability: Yes
- Visualization: See results in Figure 8.

Figure 8: Insertion Sort Result

- **Total number of executed instructions**: 2,706,132 instructions.
- **Instruction breakdown**:
  - **R-type**: 678,804 instructions (25%) – representing computation instructions like `add` and `sub`.
  - **R4-type**: 0 instructions (0%).
  - **I-type**: 677,678 instructions (25%) – primarily load and immediate instructions such as `lw` and `addi`.
  - **S-type**: 337,131 instructions (12%) – store instructions.
  - **B-type**: 675,388 instructions (24%) – branch instructions like `beq` and `bne`.
  - **U-type**: 3 instructions (0%).
  - **J-type**: 337,128 instructions (12%) – jump instructions for controlling program flow.

- **Observations**: The execution reflects a balanced distribution between **R-type**, **I-type**, and **B-type** instructions, each accounting for approximately a quarter of the total instructions. The presence of branch and jump instructions indicates significant looping and control flow operations, consistent with sorting algorithms.

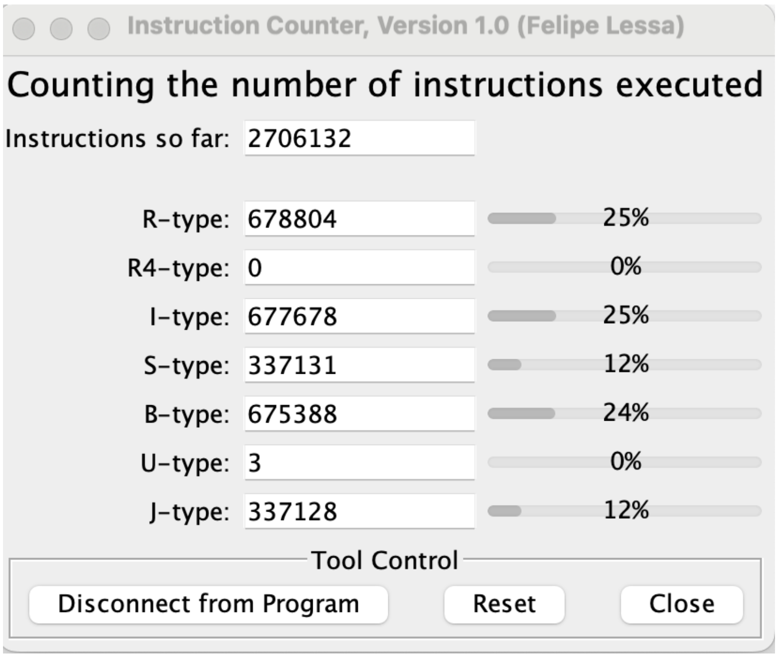

Figure 9: Instruction Count Analysis for Insertion Sort

Selection Sort

- Execution Time: 18110
- Sorting Type: Simple selection-based approach
- Consistent $O(n^2)$ performance
- Minimal number of swaps compared to Bubble Sort
- Visualization: See results in Figure 10.

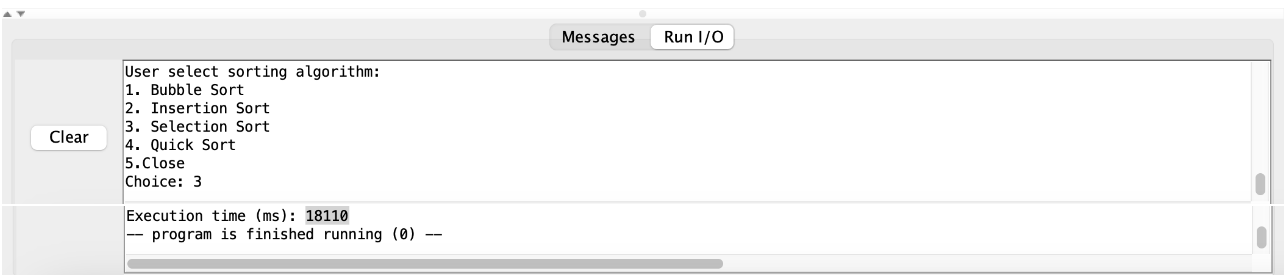

Figure 10: Selection Sort Result

- **Total number of executed instructions**: 3,994,477 instructions.
- **Instruction breakdown**:
  - **R-type**: 1,598,444 instructions (40%) – representing computation instructions like `add` and `sub`.
  - **R4-type**: 0 instructions (0%).
  - **I-type**: 1,197,911 instructions (29%) – primarily load and immediate instructions such as `lw` and `addi`.

  - **S-type**: 425 instructions (0%) – store instructions.
  - **B-type**: 798,533 instructions (19%) – branch instructions like `beq` and `bne`.
  - **U-type**: 3 instructions (0%).
  - **J-type**: 399,161 instructions (9%) – jump instructions for controlling program flow.
- **Observations**: The execution shows that **R-type** and **I-type** instructions dominate, together accounting for nearly 70% of the total instructions. **B-type** instructions (19%) highlight frequent conditional branches, while **J-type** instructions (9%) indicate program flow control. The minimal presence of **S-type** and **U-type** reflects limited store operations and unused upper immediate instructions.

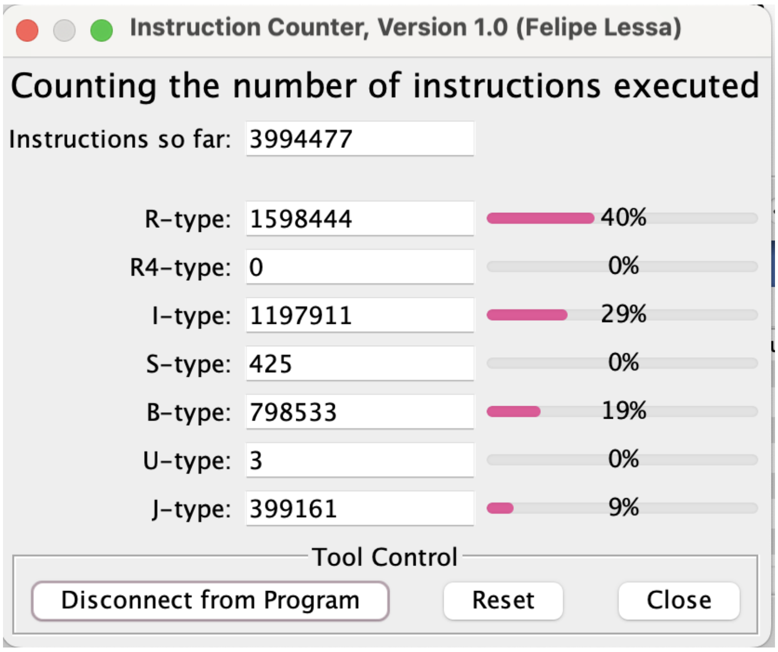

Figure 11: Instruction Count Analysis for Selection Sort

Quick Sort

- Execution Time: 218
- Sorting Type: Divide-and-conquer algorithm
- Most efficient for large datasets
- Average time complexity of $O(n \log n)$
- Space Complexity: $O(\log n)$
- Stability: No
- Visualization:

Figure 12: Quick Sort Result

- **Total number of executed instructions**: 434,841 instructions.

- **Instruction breakdown**:
  - **R-type**: 145,131 instructions (33%) – representing computation instructions like `add` and `sub`.
  - **R4-type**: 0 instructions (0%).
  - **I-type**: 143,536 instructions (33%) – primarily load and immediate instructions such as `lw` and `addi`.
  - **S-type**: 53,023 instructions (12%) – store instructions.
  - **B-type**: 60,072 instructions (13%) – branch instructions like `beq` and `bne`.
  - **U-type**: 1,037 instructions (0%) – upper immediate operations.
  - **J-type**: 32,042 instructions (7%) – jump instructions for controlling program flow.
- **Observations**: The execution shows that **R-type** and **I-type** dominate, each contributing 33% of the total instructions. This reflects the frequent arithmetic computations and memory accesses required by the Quick Sort algorithm. **B-type** instructions (13%) highlight conditional branches for partitioning and recursive calls, while **J-type** instructions (7%) reflect program flow control in recursion. The relatively smaller percentage of **S-type** instructions indicates moderate memory store operations.

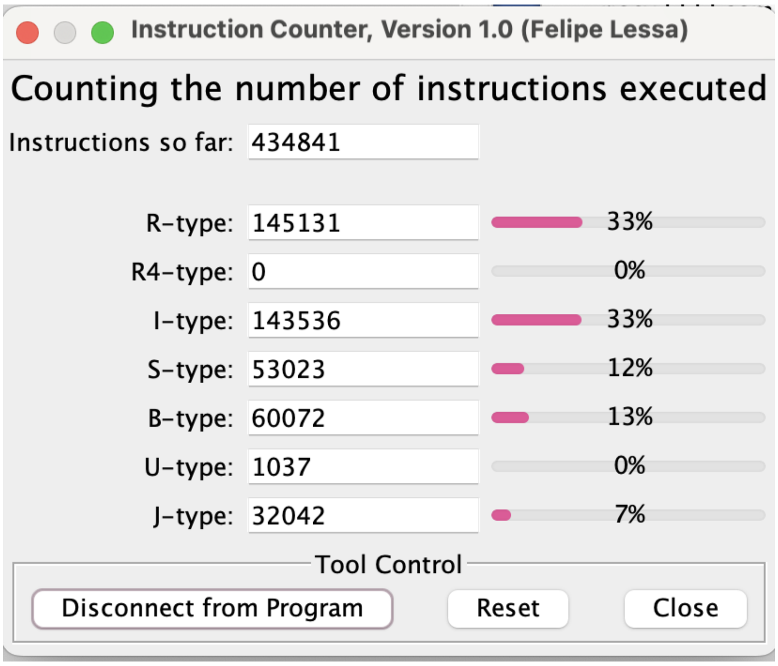

Figure 13: Instruction Count Analysis for Quick Sort

Performance Comparison Table

Algorithm	Execution Time (ms)	Time Complexity	Space Complexity	Stability
Bubble Sort	[15385]	$O(n^2)$	$O(1)$	Yes
Insertion Sort	[5504]	$O(n^2)$	$O(1)$	Yes
Selection Sort	[18110]	$O(n^2)$	$O(1)$	No
Quick Sort	[359]	$O(n \log n)$	$O(\log n)$	No

Table 1: Sorting Algorithm Performance Comparison

Key Observations

- **Execution Time Variation**:

  - Quick Sort demonstrated the most consistent and efficient performance.
  - Bubble Sort showed the longest execution times.
  - Insertion Sort performed moderately well, especially with partially sorted data.
- **Algorithm Efficiency**:
  - For small datasets: Insertion Sort shows comparable performance.
  - For large datasets: Quick Sort significantly outperforms other algorithms.

Sample Output Demonstration

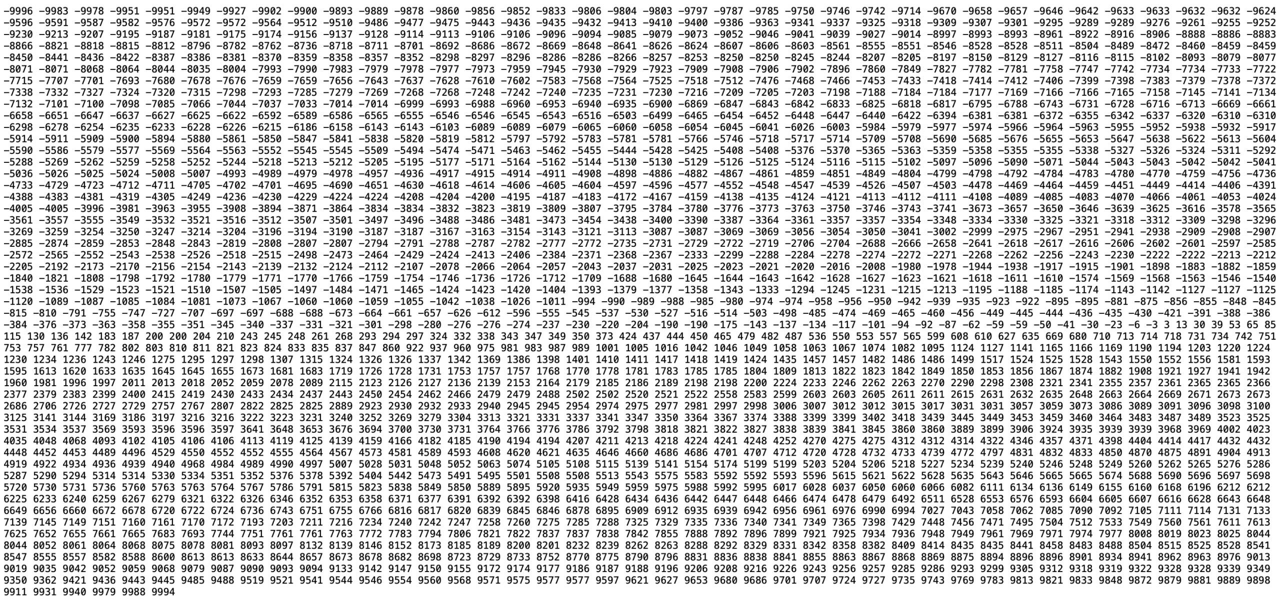

Figure 14: Output Demonstration

Error Handling and Robustness

- The program successfully handles various input scenarios.
- Supports both positive and negative integers.
- Provides clear error messages for file operations.

Recommendations

- For large datasets: Prefer Quick Sort.
- For small or nearly sorted datasets: Consider Insertion Sort.
- Avoid Bubble Sort for performance-critical applications.

Limitations

- Current implementation uses a fixed pivot selection strategy in Quick Sort.
- Memory usage is fixed and pre-allocated.
- No dynamic memory allocation implemented.

REFERENCES