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

FINAL PROJECT REPORT

Assembly Language and Computer Architecture Lab
IT3280E

Instructor: **Ph.D. Le Ba Vui**

Students: Dang Van Nhan - 20225990
Nguyen My Duyen - 20225967

Ha Noi, December 2024

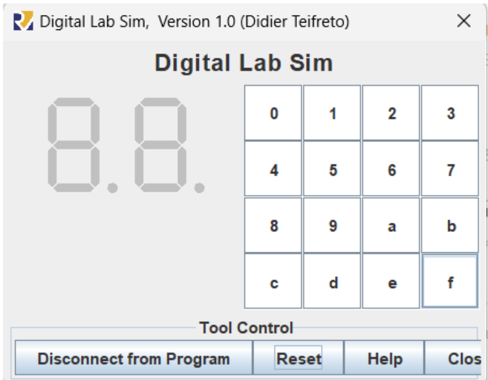
Contents

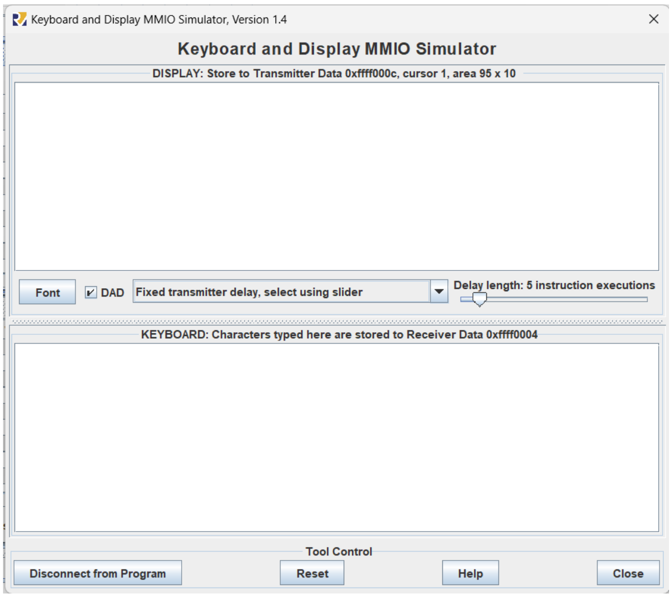
1 Problem 1: Typing test

1.1 Problem Description

Create a program to measure typing speed and display results using two 7-segment LEDs with the following requirements:

- Given a sample text, hardcoded in the source code.
- Use the timer to create measurement intervals. This is the time between two consecutive interruptions.
- The user enters text from the keyboard, the program will count the number of correct characters and display it with LEDs.
- The program also needs to estimate typing speed as the number of words per unit of time.





1.2 Approach & Method

The program will track a user's typing performance in terms of words typed, correct characters, elapsed time, and typing speed. The program presents a sample sentence, such as "The quick brown fox jumps over the lazy dog" to the user and asks them to type it. It uses a keyboard to capture user input, checks if the characters typed match the sample text, and counts correct characters and words. The test ends when the user presses the "Enter" key.

As the user types, each character entered is compared with the corresponding character in the sample text. The program keeps a count of the number of correctly typed characters. In addition, the program will monitor the total number of words typed, where words are typically separated by spaces. Once the user finishes typing or when the timer expires, the program will calculate the typing speed as the number of words per unit of time. The typing speed can be calculated using the formula:

$$\text{Typing Speed (words/min)} = \frac{\text{Number of Typed Words} * 60}{\text{Time Taken}}$$

The results are then displayed on two 7-segment LED displays. The first display shows the count of correct characters typed, while the second display shows the calculated typing speed, either in words per minute or characters per minute. The timer ensures that the user has a set time to complete the task and the interrupt mechanism allows the program to accurately track the elapsed time. Once the typing task is completed, the program ends and the final results are shown on the LEDs.

1.3 Functions & Algorithms

This section describes the key functions and the algorithms we implement in the Typing Speed Measurement Program.

1.3.1 Display Character on MMIO Output

The `display_char` function is responsible for displaying a character on the 7-segment display. To achieve this, the function first ensures that the display is ready to receive new data. This is done by checking the `DISPLAY_READY` register. Once the display is ready, the function writes the character to the `DISPLAY_CODE` register, which triggers the 7-segment display to show the corresponding character.

1.3.2 Display Correct Characters on 7-segment LEDs:

The `display_correct_chars` function handles the task of displaying the number of correct characters typed by the user on the 7-segment LEDs. First, the total number of correct characters (stored in register `s0`) is divided into tens and ones places. The function then uses the `seven_seg_patterns` lookup table to convert these decimal digits into their corresponding 7-segment display codes. The tens digit is displayed on the left 7-segment display, while the ones digit is shown on the right.

1.3.3 Calculate Typing Speed

The `calculate_speed` function calculates the user's typing speed based on the number of correct characters typed and the elapsed time. It first converts the number of correct characters (`s0`) and the elapsed time (`s4`) into floating-point values for accurate calculation. Since the elapsed time is in milliseconds, the function divides it by 1000 to convert it to seconds. The typing speed is then computed by dividing the number of correct characters by the time in seconds. Finally, the result is displayed as the typing speed in characters per second using the floating-point value.

1.3.4 Reset Display

The `reset_display_chars` function resets the 7-segment display to show the number `0`. It does so by sending the corresponding 7-segment display code for zero to both the tens and ones places. This function is called when the program starts or when a reset is required, ensuring the display starts from a clean slate.

The wait_for_key function waits for the user to press any key to begin the typing test. It continuously checks the KEYBOARD_CONTROL register, looking for a key press signal. Once a key is detected, the function allows the program to proceed, thus starting the typing test.

1.3.5 Get Keyboard Input

The get_keyboard_input function is responsible for reading input from the keyboard during the typing test. It waits for a key press by checking the KEYBOARD_CONTROL register. When a key is pressed, the function retrieves the typed character from the KEYBOARD_DATA register and returns it to the program for further processing.

1.4 RISC-V Assembly Code

Listing 1: Source Code

```
.data
    sample_text:          .asciz "The quick brown fox jumps over the lazy dog"
    prompt_start:         .asciz "Press any key to start typing test:\n"
    word_count_msg:       .asciz "Number of words: "
    correct_chars_msg:    .asciz "Number of correct characters: "
    elapsed_time_msg:     .asciz "Elapsed time (seconds): "
    typing_speed_msg:     .asciz "Typing Speed (words/min): "
    retry_prompt:         .asciz "Do you want to try again?"
    newline:              .asciz "\n"
    word_count:           .word 0

# Memory-mapped I/O addresses
.eqv KEYBOARD_CONTROL   0xFFFF0000
.eqv KEYBOARD_DATA      0xFFFF0004
.eqv DISPLAY_CONTROL    0xFFFF0008
.eqv DISPLAY_DATA       0xFFFF000C
.eqv DISPLAY_7SEG_LEFT  0xFFFF0011
.eqv DISPLAY_7SEG_RIGHT 0xFFFF0010

# Seven-segment display patterns (0-9)
seven_seg_patterns:
    .byte 0x3F, 0x06, 0x5B, 0x4F, 0x66  # 0-4
    .byte 0x6D, 0x7D, 0x07, 0x7F, 0x6F  # 5-9

.text
.globl main

main:
retry_start:
    # Initialize variables
    li s0, 0            # s0 = correct character counter
    li s1, 0            # s1 = total characters typed
    sw zero, word_count, t0  # Reset word count

    # Reset seven-segment display
    jal reset_display_chars

    # Display start prompt
```

```
    li a7, 4
    la a0, prompt_start
    ecall

    # Wait for key press to start
    jal wait_for_key

    # Get start time
    li a7, 30
    ecall
    mv s2, a0          # s2 = start time

    # Initialize text pointer
    la s3, sample_text
    li s4, 0           # Previous character (for word counting)

typing_loop:
    # Get keyboard input
    jal get_keyboard_input
    mv s5, a0          # Save current character

    # Display typed character
    jal display_char

    # Check for Enter key (end condition)
    li t0, 10          # ASCII for newline
    beq s5, t0, end_typing_test

    # Compare with sample text
    lb t1, (s3)
    beq s5, t1, correct_char
    j check_word

correct_char:
    addi s0, s0, 1     # Increment correct characters
    jal display_correct_chars

check_word:
    # Check if current char is space
    li t0, 32          # ASCII for space
    bne s5, t0, next_char

    # Check if previous char wasn't space
    li t0, 32
    beq s4, t0, next_char

    # Increment word count
    lw t1, word_count
    addi t1, t1, 1
    sw t1, word_count, t0

next_char:
    mv s4, s5          # Save current char as previous
    addi s3, s3, 1     # Move to next sample text char
    addi s1, s1, 1     # Increment total chars
    j typing_loop
```

```

end_typing_test:
    # Check last word (if doesn't end with space)
    li t0, 32          # ASCII for space
    beq s4, t0, skip_last_word

    # Count last word
    lw t1, word_count
    addi t1, t1, 1
    sw t1, word_count, t0

skip_last_word:
    # Get end time
    li a7, 30
    ecall
    mv s3, a0          # s3 = end time

    # Calculate elapsed time
    sub s4, s3, s2    # s4 = elapsed time in milliseconds

    # Display results
    # 1. Word count
    li a7, 4
    la a0, word_count_msg
    ecall

    li a7, 1
    lw a0, word_count
    ecall

    li a7, 4
    la a0, newline
    ecall

    # 2. Correct characters
    li a7, 4
    la a0, correct_chars_msg
    ecall

    li a7, 1
    mv a0, s0
    ecall

    li a7, 4
    la a0, newline
    ecall

    # 3. Elapsed time
    li a7, 4
    la a0, elapsed_time_msg
    ecall

    # Convert ms to seconds
    fcvt.s.w ft0, s4          # Convert ms to float
    li t0, 1000
    fcvt.s.w ft1, t0          # Convert 1000 to float
```

```
    fdiv.s ft0, ft0, ft1      # Divide by 1000 for seconds

    fmv.s fa0, ft0
    li a7, 2                  # Print float
    ecall

    li a7, 4
    la a0, newline
    ecall

    # 4. Typing speed
    li a7, 4
    la a0, typing_speed_msg
    ecall

    # Check if elapsed time <= 1 ms
    li t0, 10
    bgt s4, t0, calculate_typing_speed  # If elapsed time > 1 ms, go to
        typing speed calculation

    # If elapsed time <= 1 ms, typing speed is 0
    li a0, 0
    li a7, 1                  # syscall print integer
    ecall

    j typing_speed_done       # Jump to done after printing 0

calculate_typing_speed:
    # Calculate words per minute
    lw t0, word_count
    fcvt.s.w ft1, t0          # Convert word count to float
    li t0, 60
    fcvt.s.w ft2, t0          # ft2 = 60
    fmul.s ft1, ft1, ft2      # ft1 = ft1 * 60
    fdiv.s ft0, ft1, ft0      # words/minute

    fmv.s fa0, ft0
    li a7, 2                  # syscall print float
    ecall

    li a7, 4
    la a0, newline
    ecall

typing_speed_done:
    li a7, 4
    la a0, newline
    ecall

    # Retry prompt
    la a0, retry_prompt
    li a7, 50
    ecall

    beqz a0, retry_start     # If input = 0 (Yes), retry

```

```
    # Else, exit program
    li a7, 10
    ecall

# Helper functions
wait_for_key:
    li t0, KEYBOARD_CONTROL
wait_key_loop:
    lw t1, (t0)
    andi t1, t1, 1
    beqz t1, wait_key_loop
    ret

get_keyboard_input:
    li t0, KEYBOARD_CONTROL
    li t1, KEYBOARD_DATA
wait_input_loop:
    lw t2, (t0)
    andi t2, t2, 1
    beqz t2, wait_input_loop
    lw a0, (t1)
    ret

display_char:
    li t0, DISPLAY_CONTROL
wait_display:
    lw t1, (t0)
    andi t1, t1, 1
    beqz t1, wait_display
    li t0, DISPLAY_DATA
    sw a0, (t0)
    ret

display_correct_chars:
    # Get patterns for digits
    la t0, seven_seg_patterns

    # Calculate tens digit
    li t1, 10
    div t2, s0, t1        # t2 = tens
    add t3, t0, t2
    lb t4, (t3)

    # Display left digit
    li t5, DISPLAY_7SEG_LEFT
    sb t4, (t5)

    # Calculate ones digit
    rem t2, s0, t1        # t2 = ones
    add t3, t0, t2
    lb t4, (t3)

    # Display right digit
    li t5, DISPLAY_7SEG_RIGHT
    sb t4, (t5)
    ret
```

```

reset_display_chars:
    la t0, seven_seg_patterns
    lb t1, (t0)          # Get pattern for '0'

    # Reset both digits to 0
    li t2, DISPLAY_7SEG_LEFT
    sb t1, (t2)
    li t2, DISPLAY_7SEG_RIGHT
    sb t1, (t2)
    ret
```

1.5 Simulation Results

The program evaluates user input under three scenarios: correct input, incorrect input, and empty input (pressing Enter without typing any characters). Each case produces distinct results, as described below.

1.5.1 Correct Input Case

In the correct input case, the user typed the given sample text correctly. The program counted the number of correct characters, computed the typing speed, and displayed the results accordingly.

Input:

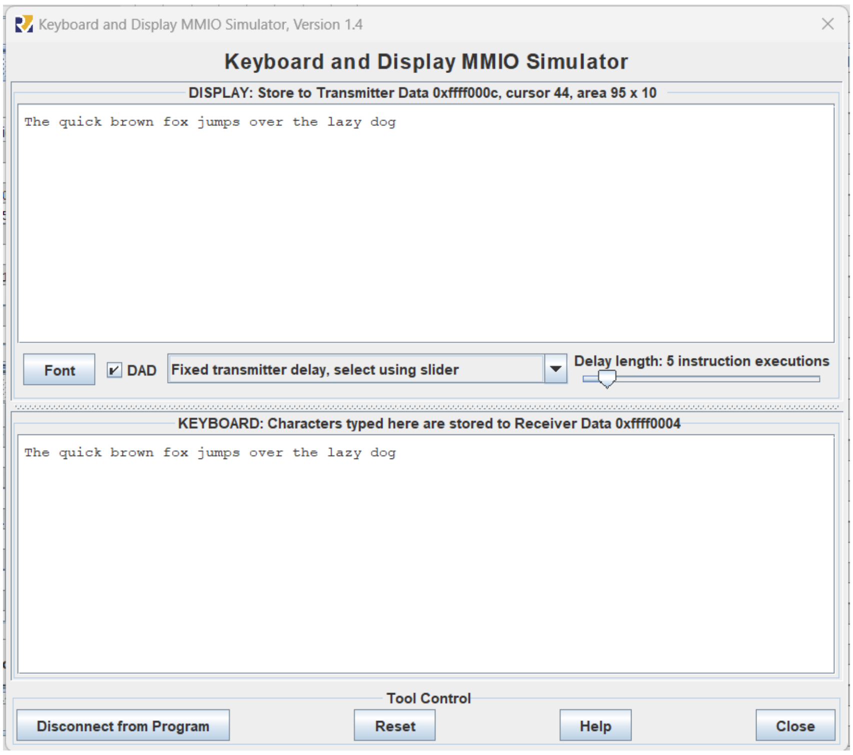


Figure 1: Correct input

Output:

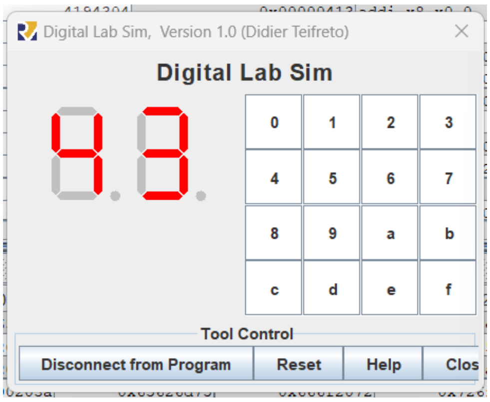


Figure 2: Digital Lab result

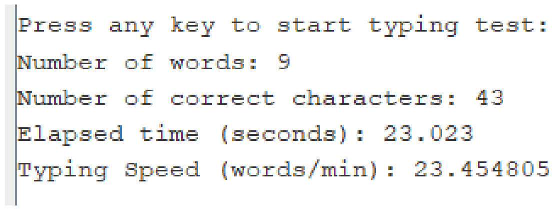


Figure 3: Terminal result

1.5.2 Incorrect Input Case

In the incorrect input case, the user made some typing mistakes. The program counts only the correct characters typed, and the typing speed is affected by the errors.

Input:

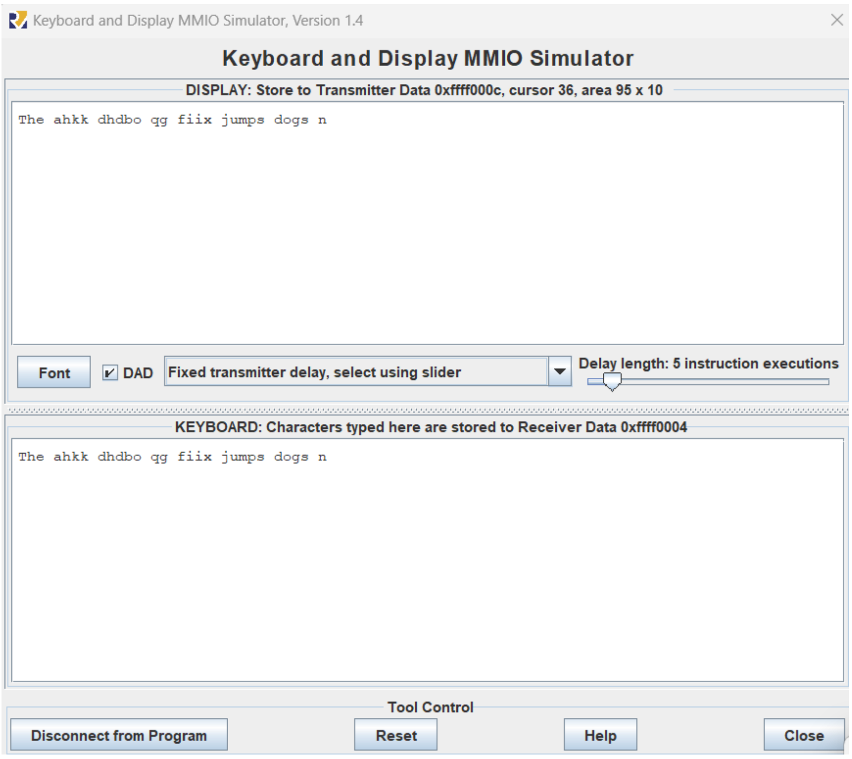


Figure 4: Incorrect input

Output:

1.5.3 Empty Input Case

In the empty input case, the user presses Enter without typing any characters. The program detects this scenario and handles it appropriately by displaying a result of zero correct characters and zero typing speed.

Input:

Output:

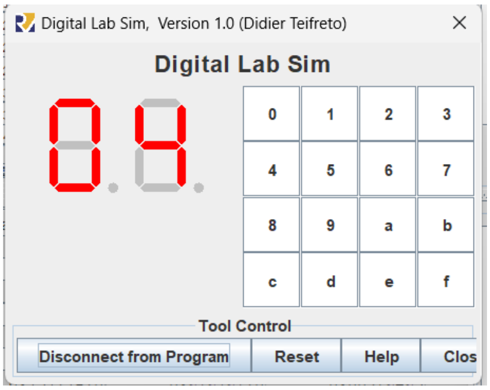


Figure 5: Digital Lab result

```
Press any key to start typing test:
Number of words: 8
Number of correct characters: 4
Elapsed time (seconds): 14.639
Typing Speed (words/min): 32.789124
```

Figure 6: Terminal result

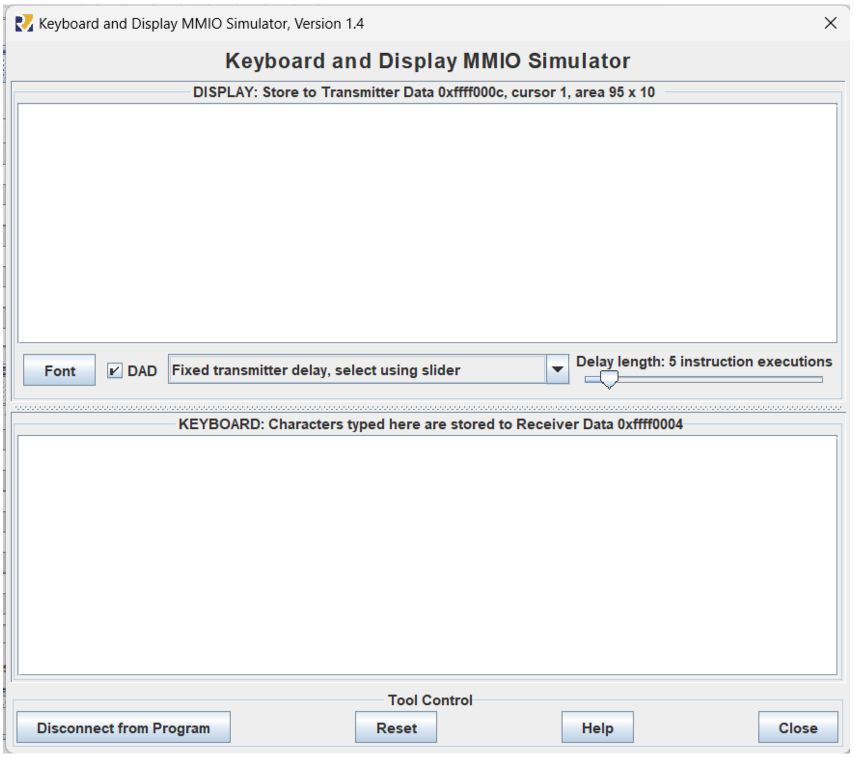


Figure 7: Empty input

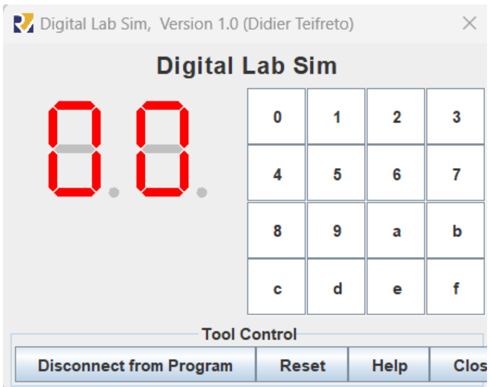


Figure 8: Digital Lab result

```
Press any key to start typing test:
Number of words: 1
Number of correct characters: 0
Elapsed time (seconds): 0.001
Typing Speed (words/min): 0
```

Figure 9: Terminal result

1.5.4 Retry Prompt

At the end of each test, the program displays a retry prompt, as shown in Figure 17. This allows the user to choose whether to try the typing test again or exit the program.

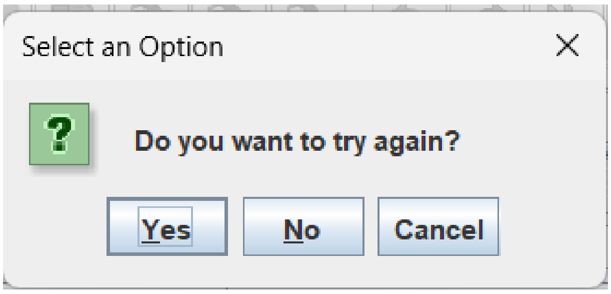


Figure 10: Retry Prompt

In this prompt, the user has three options:

- **Yes**: Restart the typing test.
- **No**: Exit the test and display a goodbye message.
- **Cancel**: Return to the current screen and exit.

Behavior of Each Option:

1. If **Yes** is selected, the program performs the following actions:
   - Resets the typing environment, including clearing the 7-segment LED display and resetting counters.
   - Starts a new typing test with the same sample text.

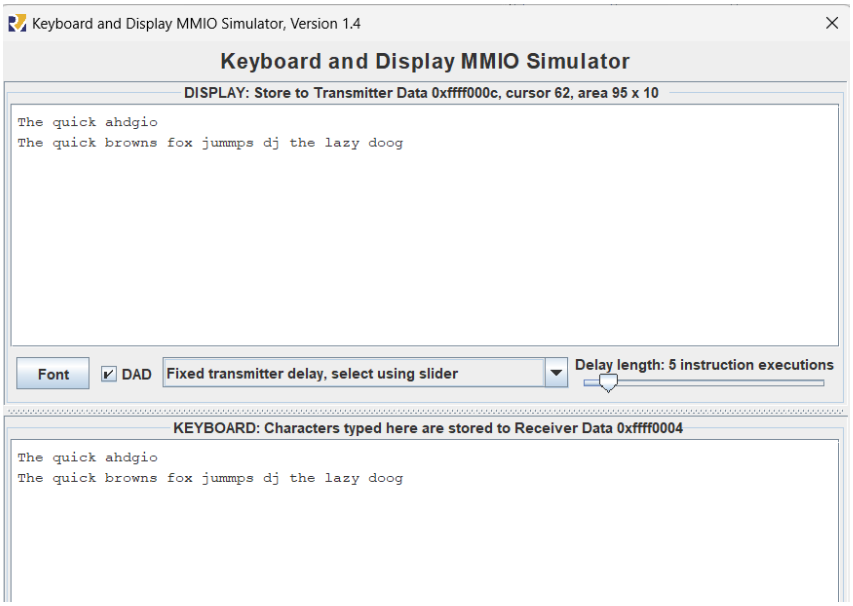


Figure 11: User has new turn for inputing

2. If **No** is selected, the program terminates gracefully and displays a goodbye message:

"Exiting..."

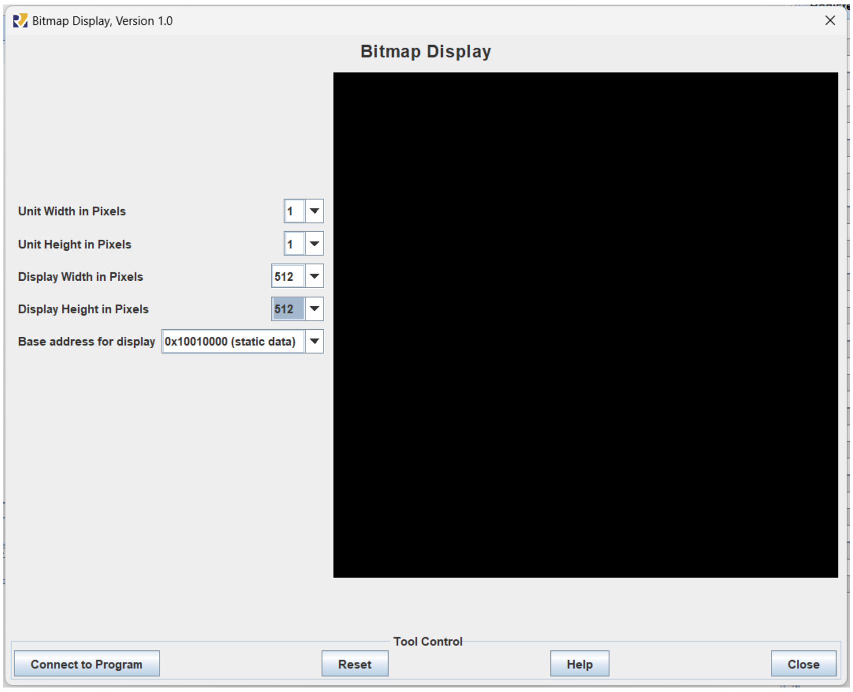
3. If **Cancel** is selected, the program returns to the current state, maintaining the last recorded results (e.g., correct characters and elapsed time) and exiting.

2 Problem 2: Graph functions

2.1 Problem Description

Draw a graph representing a function in the Bitmap Display window with the following requirements:

- The user enters the coefficients of the equation $y = ax^2 + bx + c$ and enters the graph color code in Run I/O window.
- The program draws a graph with the Oxy coordinate axis in the middle of the Bitmap Display screen (set the screen size to 512x512).
- After drawing a graph, the program allows the user to choose to draw another graph or end the program.



2.2 Approach & Method

The program will allow the user to input the coefficients of a quadratic equation $y = ax^2 + bx + c$ and a color code in hexadecimal format, which will be used to draw the graph on a Bitmap Display window. The screen resolution is set to 512x512 pixels, with the Oxy coordinate axes drawn at the center of the screen. The program validates user inputs to ensure that the coefficients and color code are entered correctly, with appropriate error handling for invalid inputs.

Upon receiving valid inputs, the program calculates the values of y for a range of x-coordinates, from -256 to 255, and plots the corresponding points on the screen. The graph is drawn using the specified color code, and the coordinate axes are displayed in white for clarity. The graph is continuously updated on the Bitmap Display, clearing the screen before drawing each new function.

After completing the graph drawing, the program prompts the user to decide whether they want to draw another graph or exit the program. If the user opts to continue, the program repeats the process, asking for new input values. If the user chooses to exit, the program displays a farewell message and terminates. This loop ensures that the user can visualize multiple graphs without restarting the program.

2.3 Functions & Algorithms

This section describes the key functions and the algorithms implemented in the **Quadratic Function Graphing Program**.

2.3.1 Input and Validate Integer Values

The `validate_integer` function ensures that user input is a valid integer. It proceeds as follows:

1. Check for an optional negative sign at the beginning of the input.
2. Iterate through each character to verify that it is a numeric digit (0–9).
3. Convert each digit character into its numeric value and accumulate the result.
4. If any invalid character is found, the function returns 0, indicating an invalid input.
5. For valid inputs, the integer value is returned, accounting for the sign if present.

2.3.2 Input and Validate Hexadecimal Color Code

The `validate_hex_color` function verifies whether the user input follows the correct format for a hexadecimal color code. The algorithm works as follows:

1. Ensure that the input starts with the prefix `0x`.
2. Validate that the next six characters are valid hexadecimal digits (0–9, A–F, a–f).
3. Convert each valid hex digit to its corresponding numeric value.
4. Combine these values to form the final hexadecimal color code.
5. If any character is invalid or the format is incorrect, the function returns 0 to indicate failure.

2.3.3 Draw Coordinate Axes

The `draw_axes` function draws the X-axis and Y-axis on the screen. The steps are:

1. The screen's middle row (for X-axis) and middle column (for Y-axis) are calculated.
2. Pixels along the X-axis are updated to display a horizontal line at the screen's center.
3. Pixels along the Y-axis are updated to display a vertical line at the center of the screen.
4. The axes are drawn using a predefined color (e.g., white).

2.3.4 Plot the Quadratic Function

The `draw_function` function plots the quadratic equation $y = ax^2 + bx + c$ on the screen. The algorithm follows:

1. Iterate through all x-coordinates from -256 to 256 (screen width).
2. For each x, compute y using the formula:

$$y = ax^2 + bx + c$$

3. Scale y to fit within the screen's pixel coordinates.
4. Verify that the computed point (x, y) lies within the screen boundaries.
5. Update the corresponding pixel on the screen with the user-specified color.

2.3.5 Prompt User to Continue

The program prompts the user to decide whether to draw another graph using the `continue_prompt` function. The steps are:

1. Display a message asking the user if they want to draw another graph.
2. Read the user's response.
3. If the response is `0` (yes), the program restarts from the beginning.
4. Otherwise, the program exits.

2.4 RISC-V Assembly Code

Listing 2: Source Code

```
.data
prompt_a:        .asciz "Enter coefficient a: "    # Prompt message for coefficient a
prompt_b:        .asciz "Enter coefficient b: "    # Prompt message for coefficient b
prompt_c:        .asciz "Enter coefficient c: "    # Prompt message for coefficient c
prompt_color:    .asciz "Enter color code (hex format, e.g., 0xFF0000 for RED): "  # Prompt for color code
continue_msg:    .asciz "Do you want to draw another graph?"    # Ask if the user wants to continue
error_int:       .asciz "Invalid input! Please enter an integer.\n"    # Error message for invalid integer input
error_hex:       .asciz "Invalid color code! Please enter a valid hex color (e.g., 0xFF0000).\n"  # Error message for invalid hex color input
buffer:          .space 10    # Buffer to store string input from user

.eqv MONITOR_SCREEN 0x10010000   # Screen base address
.eqv SCREEN_WIDTH 512            # Screen width in pixels
```

```
.eqv SCREEN_HEIGHT 512          # Screen height in pixels

# Predefined colors
.eqv RED 0xFF0000
.eqv GREEN 0x00FF00
.eqv BLUE 0x0000FF
.eqv WHITE 0xFFFFFF
.eqv YELLOW 0xFFFF00
.eqv BLACK 0x000000

.text

main:
input_a:
    # Prompt for coefficient a
    li a7, 4
    la a0, prompt_a
    ecall

    # Read input as string
    li a7, 8
    la a0, buffer
    li a1, 20
    ecall

    # Validate and convert string to integer
    jal validate_integer
    beqz a0, invalid_int_a  # If input is invalid, go to error handling
    mv s0, a1               # Store valid integer in s0
    j input_b                # Move to next coefficient input

invalid_int_a:
    # Display error for invalid integer input
    li a7, 4
    la a0, error_int
    ecall
    j input_a  # Retry input for coefficient a

input_b:
    # Prompt for coefficient b
    li a7, 4
    la a0, prompt_b
    ecall

    li a7, 8
    la a0, buffer
    li a1, 20
    ecall

    # Validate and convert string to integer
    jal validate_integer
    beqz a0, invalid_int_b
    mv s1, a1               # Store valid integer in s1
    j input_c                # Move to next coefficient input

invalid_int_b:
```

```
    # Display error for invalid integer input
    li a7, 4
    la a0, error_int
    ecall
    j input_b  # Retry input for coefficient b

input_c:
    # Prompt for coefficient c
    li a7, 4
    la a0, prompt_c
    ecall

    li a7, 8
    la a0, buffer
    li a1, 20
    ecall

    # Validate and convert string to integer
    jal validate_integer
    beqz a0, invalid_int_c
    mv s2, a1                # Store valid integer in s2
    j input_color             # Move to color input

invalid_int_c:
    # Display error for invalid integer input
    li a7, 4
    la a0, error_int
    ecall
    j input_c  # Retry input for coefficient c

input_color:
    # Prompt for color code input
    li a7, 4
    la a0, prompt_color
    ecall

    li a7, 8
    la a0, buffer
    li a1, 20
    ecall

    # Validate and convert hex color code
    jal validate_hex_color
    beqz a0, invalid_color
    mv s3, a1                # Store valid color code in s3
    j draw_axes              # Proceed to draw the graph

invalid_color:
    # Display error for invalid hex color code
    li a7, 4
    la a0, error_hex
    ecall
    j input_color            # Retry color input

# ---------------------- Validation Functions ---------------------- #

```

```
# Function to validate integer input
# Returns: a0 = 1 if valid, 0 if invalid
#          a1 = converted integer value
validate_integer:
    # Save registers
    addi sp, sp, -12
    sw ra, 0(sp)
    sw s0, 4(sp)
    sw s1, 8(sp)

    la t0, buffer
    li t1, 0        # Initialize result to 0
    li t2, 0        # Initialize sign (0 = positive, 1 = negative)
    lb t3, 0(t0)    # Load first character

    # Check if input has a negative sign
    li t4, '-'
    bne t3, t4, not_negative
    li t2, 1
    addi t0, t0, 1
    lb t3, 0(t0)

not_negative:
    # Process the digits in the string
process_digit:
    lb t3, 0(t0)
    beqz t3, end_validate_int    # End of string
    li t4, '\n'
    beq t3, t4, end_validate_int # Handle newline character

    # Check if character is a digit (0-9)
    li t4, '0'
    blt t3, t4, invalid_integer
    li t4, '9'
    bgt t3, t4, invalid_integer

    # Convert character to digit
    addi t3, t3, -48
    li t4, 10
    mul t1, t1, t4
    add t1, t1, t3

    addi t0, t0, 1
    j process_digit

invalid_integer:
    li a0, 0    # Invalid input, return 0
    j validate_int_exit

end_validate_int:
    # Apply sign (negative if needed)
    beqz t2, skip_negate
    neg t1, t1  # Negate if negative sign was found
skip_negate:
    li a0, 1    # Valid input
    mv a1, t1   # Store the result
```

```

validate_int_exit:
    # Restore registers
    lw ra, 0(sp)
    lw s0, 4(sp)
    lw s1, 8(sp)
    addi sp, sp, 12
    ret

# Function to validate hex color code
# Returns: a0 = 1 if valid, 0 if invalid
#          a1 = converted hex value
validate_hex_color:
    # Save registers
    addi sp, sp, -12
    sw ra, 0(sp)
    sw s0, 4(sp)
    sw s1, 8(sp)

    la t0, buffer
    li t1, 0       # Initialize result to 0

    # Check for "0x" prefix
    lb t2, 0(t0)
    li t3, '0'
    bne t2, t3, invalid_hex
    lb t2, 1(t0)
    li t3, 'x'
    bne t2, t3, invalid_hex
    addi t0, t0, 2

    # Process hex digits (0-9, A-F, a-f)
    li t6, 6       # Limit to 6 hex digits
process_hex:
    beqz t6, end_validate_hex  # Stop after 6 digits
    lb t2, 0(t0)
    beqz t2, invalid_hex    # String too short

    # Convert hex digit
    li t3, '0'
    blt t2, t3, invalid_hex
    li t3, '9'
    ble t2, t3, hex_digit_number

    # Check for A-F characters
    li t3, 'A'
    blt t2, t3, check_lowercase
    li t3, 'F'
    bgt t2, t3, check_lowercase
    addi t2, t2, -55        # Convert A-F to 10-15
    j hex_digit_ok

check_lowercase:
    li t3, 'a'
    blt t2, t3, invalid_hex
    li t3, 'f'
```

```
    bgt t2, t3, invalid_hex
    addi t2, t2, -87        # Convert a-f to 10-15
    j hex_digit_ok

hex_digit_number:
    addi t2, t2, -48        # Convert '0'-'9' to 0-9

hex_digit_ok:
    slli t1, t1, 4          # Shift result left by 4 bits
    or t1, t1, t2           # Combine hex digit
    addi t0, t0, 1           # Move to next character
    addi t6, t6, -1          # Decrease remaining digits count
    j process_hex

invalid_hex:
    li a0, 0    # Invalid hex, return 0
    j validate_hex_exit

end_validate_hex:
    li a0, 1     # Valid hex
    mv a1, t1    # Store hex value

validate_hex_exit:
    # Restore registers
    lw ra, 0(sp)
    lw s0, 4(sp)
    lw s1, 8(sp)
    addi sp, sp, 12
    ret

# ------------------------- Draw Coordinate Axes
   -------------------------- #
# Function to draw coordinate axes (X and Y axes on the screen)
draw_axes:
    # Draw X-axis (horizontal line at center)
    li t0, MONITOR_SCREEN     # Base address for screen memory
    li t1, SCREEN_WIDTH       # Screen width in pixels
    li t2, SCREEN_HEIGHT      # Screen height in pixels
    li t3, WHITE              # Axis color (white)

    # Calculate middle row for Y-axis
    srli t4, t2, 1           # t4 = height/2 (y=height/2 for X-axis)
    mul t4, t4, t1           # Multiply by screen width
    slli t4, t4, 2           # Multiply by 4 (bytes per pixel)
    add t4, t4, t0           # Add base address for X-axis

    # Draw horizontal line for X-axis
    li t5, 0                 # Initialize counter
draw_x_axis:
    sw t3, 0(t4)            # Draw pixel
    addi t4, t4, 4          # Next pixel
    addi t5, t5, 1          # Increment counter
    blt t5, t1, draw_x_axis # Continue until width is reached

    # Draw Y-axis (vertical line at center)
    li t0, MONITOR_SCREEN    # Reset base address
```

```
    srli t4, t1, 1            # t4 = width/2 for Y-axis
    slli t4, t4, 2            # Multiply by 4 (bytes per pixel)
    add t4, t4, t0            # Add to base address for Y-axis

    # Draw vertical line for Y-axis
    li t5, 0                  # Initialize counter
    li t6, SCREEN_WIDTH       # Screen width for offset calculation
    slli t6, t6, 2            # Multiply by 4 (bytes per pixel)
draw_y_axis:
    sw t3, 0(t4)              # Draw pixel
    add t4, t4, t6            # Move down one row (screen width * 4)
    addi t5, t5, 1            # Increment counter
    blt t5, t2, draw_y_axis # Continue until height is reached

# -------------------------- Draw the Quadratic Function --------------------------- #
# Function to plot the quadratic equation y = ax^2 + bx + c
draw_function:
    # Parameters: s0 = a (coefficient of x^2), s1 = b (coefficient of x), s2 = c (constant), s3 = color
    li t0, MONITOR_SCREEN     # Base address for screen memory
    li t1, -256               # Start x-coordinate at -256 (left side)
    li t2, 256                # End x-coordinate at 256 (right side)

plot_loop:
    # Calculate y = ax^2 + bx + c
    mul t3, t1, t1            # t3 = x^2
    mul t3, t3, s0            # t3 = ax^2
    mul t4, s1, t1            # t4 = bx
    add t3, t3, t4            # t3 = ax^2 + bx
    add t3, t3, s2            # t3 = ax^2 + bx + c

    # Convert to screen coordinates (invert Y-axis)
    neg t3, t3                # Negate y value (since screen coordinates are inverted)
    addi t3, t3, 256          # Center Y-coordinate on screen (height/2)

    # Ensure y is within screen bounds
    li t4, 0
    blt t3, t4, skip_point    # Skip if y < 0
    li t4, 512
    bge t3, t4, skip_point    # Skip if y >= screen height

    # Calculate pixel address for (x, y)
    li t4, SCREEN_WIDTH
    mul t4, t4, t3            # t4 = y * screen width
    add t4, t4, t1            # Add x to get the correct pixel position
    addi t4, t4, 256          # Center x-coordinate on screen
    slli t4, t4, 2            # Multiply by 4 (bytes per pixel)
    add t4, t4, t0            # Add base address

    # Draw pixel at calculated address
    sw s3, 0(t4)              # Store color at the pixel location

skip_point:
    addi t1, t1, 1            # Increment x-coordinate
```

```
    ble t1, t2, plot_loop    # Continue until all points are plotted

    # Prompt to ask user if they want to draw another graph
continue_prompt:
    li a7, 50
    la a0, continue_msg
    ecall

    beqz a0, main            # If user chooses 'yes', restart program

    # Exit program if user chooses 'no'
    li a7, 10
    ecall
```

2.5 Simulation Results

In this section, we will test various input cases and examine how the program handles them.

2.5.1 Valid Inputs

When valid inputs are entered for the coefficients a, b, c, and the color code, the program correctly computes the values of the quadratic function $y = ax^2 + bx + c$ and plots the graph.

For example:

- Entering $a = 1$, $b = -3$, $c = 2$.
- The program calculates and plots the graph for $y = x^2 - 3x + 2$.
- Entering the color code $0xFF0000$ for red.
- The graph is plotted with the red color.

The result is a parabola with the vertex and x-intercepts correctly plotted. The graph will be drawn using the color chosen from the color code input. The program supports various valid color codes such as $0x00FF00$ (green), $0x0000FF$ (blue), and $0xFFFFFF$ (white).

2.5.2 Zero Coefficients

When any of the coefficients a, b, or c is zero, the program will still calculate and plot the graph correctly. For example:

- Entering $a = 0$, $b = 3$, $c = -5$.
- The program calculates and plots the graph for $y = 3x - 5$.

In this case, the graph will not be a parabola, but a straight line. The program will handle this scenario without issues and plot the line correctly.

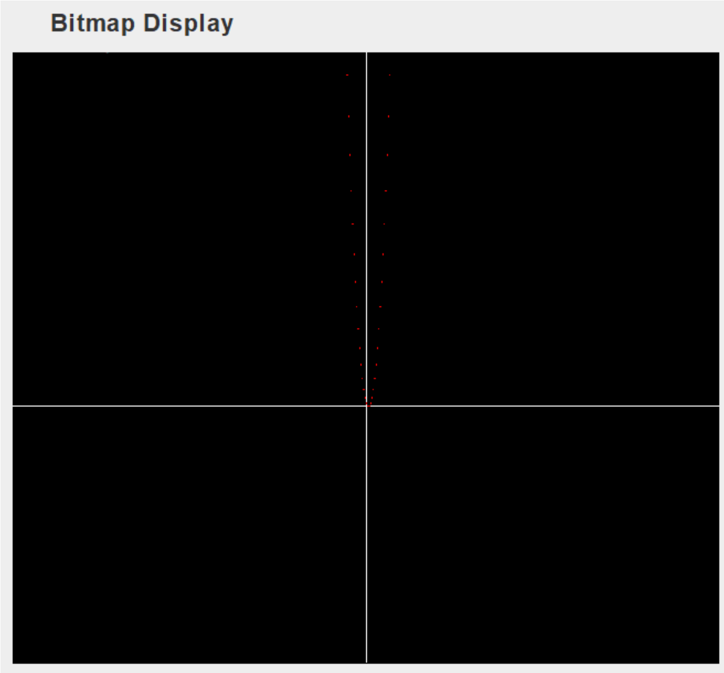


Figure 12: Bitmap Display in case of valid input

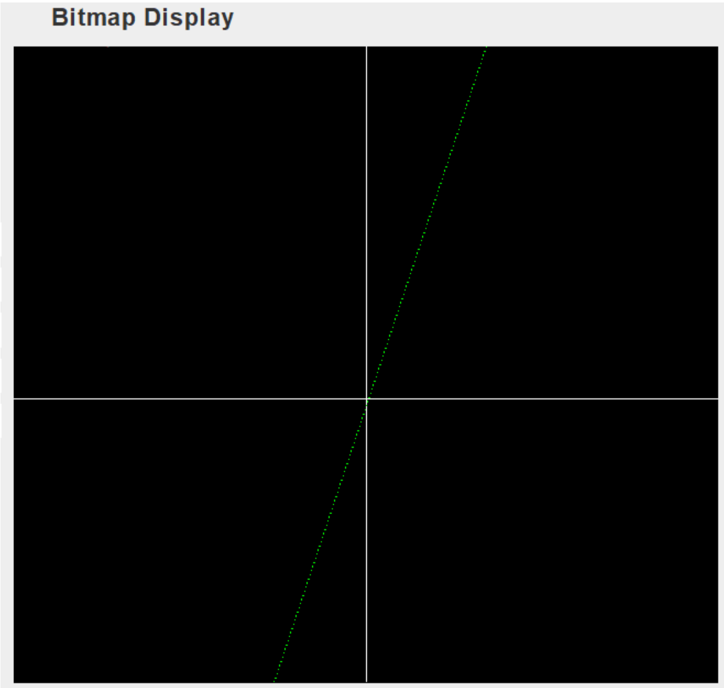


Figure 13: Bitmap Display in case of zero coefficient

2.5.3 Invalid Coefficients

When an invalid input (e.g., a non-integer) is provided for any coefficient a, b, or c, the program will display an error message and prompt the user to enter a valid value. For example:

- Entering $a =$ `abc`, $b = 3$, $c = 2$.
- The program will display the error message: "`Invalid input! Please enter an integer.`" and ask the user to re-enter a valid value for a.

```
Enter coefficient a: abc
Invalid input! Please enter an integer.
Enter coefficient a: 1
Enter coefficient b: 2
Enter coefficient c: 3
```

Figure 14: Invalid input message

The program ensures that only valid integers are accepted for the coefficients.

2.5.4 Invalid Color Code

When an invalid hexadecimal color code is entered (e.g., a code that does not conform to the valid hex format), the program will display an error message and prompt the user to enter a correct color code. For example:

- Entering $0xGG0000$ as the color code.
- The program will display the error message: "`Invalid color code! Please enter a valid hex color.`".

```
Enter color code (hex format, e.g., 0xFF0000 for RED): 0xGG0000
Invalid color code! Please enter a valid hex color (e.g., 0x00FF0000).
Enter color code (hex format, e.g., 0xFF0000 for RED): 0abs
Invalid color code! Please enter a valid hex color (e.g., 0x00FF0000).
Enter color code (hex format, e.g., 0xFF0000 for RED):
```

Figure 15: Invalid color input message

Only valid hexadecimal color codes like $0xFF0000$, $0x00FF00$, $0x0000FF$, etc., are accepted by the program.

2.5.5 Menu Choice

After plotting a graph, the program asks the user if they want to plot another graph. If the user chooses "Yes", the program will prompt for new coefficients and a color code. If the user chooses "No", the program will terminate.

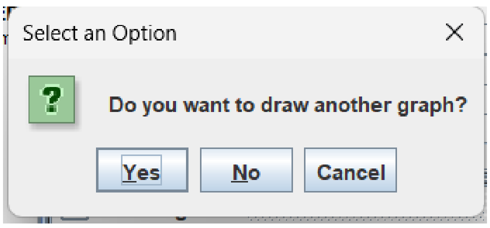


Figure 16: Menu choice

- After plotting the graph, the program asks: "Do you want to draw another graph?".
- If the user enters "Yes", the program will prompt for new coefficients and a color code to plot another graph.
- If the user enters "No", the program will exit.

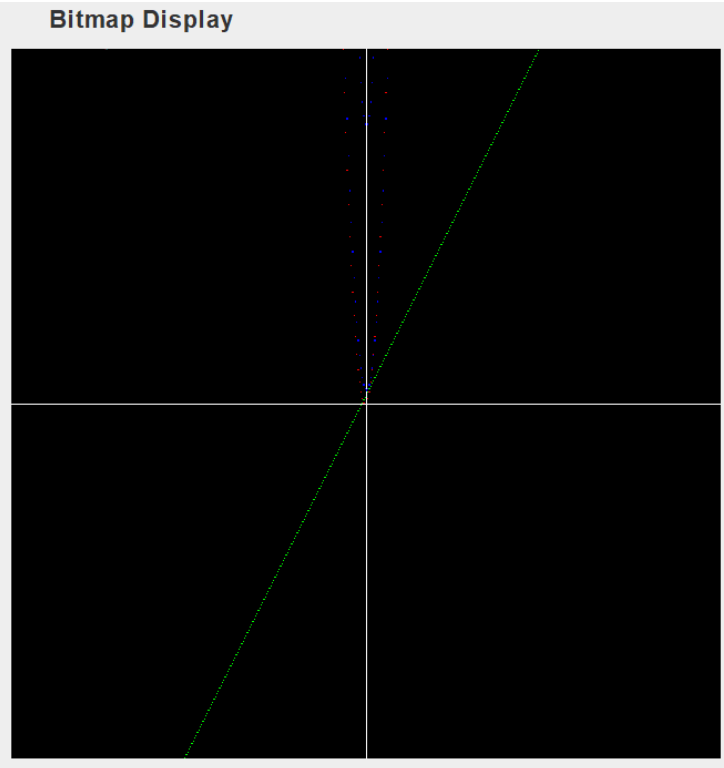


Figure 17: Multi-graphs draw