Computer Architecture and Assembly Language

Lecturer: Msc. Le Ba Vui

Class ID: 152003

Project Report

RISC-V Assembly Simple Caculator and Graph Functions

Group 14:
Le Dai Lam - 20225982
Pham Thanh Nam - 20225989

Contents

Abstract

This report encompasses two distinct projects undertaken using RISC-V assembly language: a **Simple Caculator** and a **Graph Functions**.

Simple Simple Caculator: The Simple Simple Caculator project involves designing and implementing a functional Simple Simple Caculator that interfaces with a keypad for input and displays results on dual 7-segment LEDs. It supports basic arithmetic operations, including addition, subtraction, multiplication, division, and modulus. Additionally, the system incorporates robust error handling mechanisms to manage scenarios such as division by zero and invalid input sequences. This project provides hands-on experience with low-level programming, memory-mapped I/O, and state management in assembly language.

Graph Functions: The Graph Functions project focuses on developing a program that plots quadratic equations based on user-provided coefficients and color selections. The program renders the corresponding parabola on a bitmap display, ensuring accurate graphical representation through precise memory manipulation. It includes comprehensive input validation to confirm that entered values are integers and handles invalid inputs gracefully. Furthermore, the application offers users the option to plot new graphs or exit, enhancing user interaction and experience. This project demonstrates practical applications of low-level programming, memory management, and graphical rendering using assembly language.

Both projects showcase the practical applications of low-level programming, memory-mapped I/O, state management, and graphical rendering in assembly language. They provide valuable insights into computer architecture and hardware interfacing, reinforcing foundational concepts through tangible implementations.

1 Introduction

1.1 Background

RISC-V (Reduced Instruction Set Computing V) is an open-standard instruction set architecture (ISA) renowned for its simplicity, modularity, and extensibility. Widely adopted in both academic and industrial spheres, RISC-V serves as an excellent platform for exploring low-level programming, computer architecture, and hardware-software interfacing. This report details two projects developed using RISC-V assembly language: a Simple Simple Caculator and a Graph Functions. Both projects aim to reinforce understanding of assembly programming, memory management, and hardware interaction through practical implementations.

1.2 Problem Statement

Developing functional applications in RISC-V assembly language presents several challenges, including efficient input handling, memory management, arithmetic computations, graphical rendering, and robust error handling. The Simple Simple Caculator project addresses the need for a simple arithmetic tool that interacts with hardware components like keypads and LEDs. Conversely, the Graph Functions project tackles the complexities of rendering graphical data based on mathematical equations, requiring precise memory manipulation and user input validation.

2 Project 1: RISC-V Assembly Simple Caculator

2.1 Abstract

This project aims to design and implement a Simple Caculator using RISC-V assembly language. The Simple Simple Caculator interfaces with a keypad for input and displays results on dual 7-segment LEDs. It supports basic arithmetic operations including addition, subtraction, multiplication, division, and modulus. Additionally, the system handles error conditions such as division by zero and invalid input sequences. The project provides hands-on experience with low-level programming, memory-mapped I/O, and state management in assembly language.

2.2 Introduction

2.2.1 Background

RISC-V (Reduced Instruction Set Computing V) is an open-standard instruction set architecture (ISA) based on established RISC principles. It is widely used in academia and industry for teaching assembly language programming and computer architecture concepts. Understanding RISC-V assembly language provides deep insights into how high-level code translates into machine operations, memory management, and hardware interactions.

2.2.2 Problem Statement

Developing a functional Simple Simple Caculator in RISC-V assembly language presents several challenges:

- **Input Handling:** Interfacing with a keypad requires efficient scanning and debouncing techniques to accurately detect key presses.
- **Output Display:** Displaying numerical results on 7-segment LEDs necessitates correct encoding of digits and managing dual displays for multi-digit numbers.
- **Arithmetic Operations:** Implementing basic arithmetic operations in assembly involves managing registers, handling overflow, and ensuring accurate calculations.
- **Error Management:** The system must gracefully handle errors such as division by zero and invalid input sequences to prevent undefined behavior.

2.3 Objectives

The primary objectives of this project are:

- To implement keypad input scanning and key press detection in RISC-V assembly.
- To perform basic arithmetic operations: addition, subtraction, multiplication, division, and modulus.
- To display results on dual 7-segment LED displays.
- To handle error conditions, including division by zero and invalid input sequences.
- To ensure smooth user interaction through input debouncing and result rendering.

2.4 Project Implementation

2.4.1 Memory-Mapped I/O Setup

The project utilizes memory-mapped I/O to interface with hardware components:

- `0x10010011` and `0x10010010` for the two 7-segment LEDs (left and right).
- `0x10010012` and `0x10010014` for reading from and writing to the keypad.

2.4.2 Key Mapping

Each key on the keypad is assigned a unique code using the `.eqv` directive. This mapping facilitates easy identification of pressed keys during the scanning process. For example:

- Digits: `CODE_0` to `CODE_9`
- Operators: `CODE_A` (Addition), `CODE_B` (Subtraction), etc.

2.4.3 Data Structures

A lookup table, `NUMS_OF_7SEG`, is defined to map each digit (0–9) to its corresponding 7-segment LED encoding. This allows the program to write the correct bit pattern directly to the LED's memory address to display the desired digit.

2.4.4 Control Flow

The program initializes necessary registers and enters a polling loop to continuously scan the keypad for input. Upon detecting a key press, it determines whether the input is a digit, an operator, or the equals sign, and processes it accordingly. Arithmetic operations are performed based on the current state and inputs, with results displayed on both the LEDs and the console.

2.4.5 State Management

Several registers are used to maintain the Simple Simple Caculator's state:

`s0` Stores the current key code.

`s1` Holds the numeric value or operator code extracted from the key press.

`s2` Indicates the type of input (1 for number, 2 for operator, 3 for equals).

`s3` Tracks the current number being entered.

`s4` Stores the pending operator code.

`s5` Holds interim or final results.

`s6` Flag indicating whether a number has been entered.

`s7` Flag indicating if there is a pending operation.

2.4.6 Arithmetic Operations

Basic arithmetic operations are implemented using RISC-V instructions. For example:

- **Addition:** Uses the `add` instruction.
- **Subtraction:** Uses the `sub` instruction.
- **Multiplication:** Uses the `mul` instruction.
- **Division:** Uses the `div` and `rem` instructions for quotient and remainder.

Each operation updates the relevant registers and handles any necessary state transitions.

2.4.7 Error Handling

The system includes mechanisms to handle errors gracefully:

- **Division by Zero:** Checks for division or modulus by zero and displays an error indication ('E') without performing the operation.
- **Invalid Input Sequences:** Detects if an operator is pressed before any number and prompts the user to enter a number first.

2.4.8 Delay and Debouncing

A sleep routine introduces a 100ms delay after processing each key press to prevent rapid, unintended multiple detections of the same key. This simulates debouncing, ensuring that each key press is registered accurately.

2.5 Methodology

The project follows a systematic approach to ensure all objectives are met:

2.5.1 Design Phase

- Define the key mapping and identify memory-mapped I/O addresses.
- Design the state management scheme using registers.
- Create the lookup table for 7-segment LED encoding.

2.5.2 Implementation Phase

- Develop the keypad scanning and input detection routine.
- Implement arithmetic operations and state transitions.
- Create subroutines for rendering numbers on the 7-segment LEDs.
- Integrate error handling mechanisms.

2.5.3 Testing Phase

- Simulate key presses and verify correct detection.
- Test each arithmetic operation for accuracy.
- Validate error handling by inducing error conditions.
- Ensure proper display of results on LEDs and console.

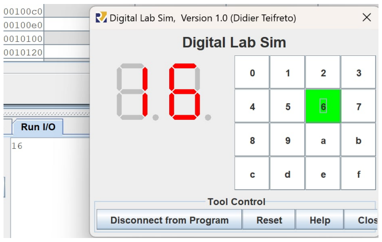
2.5.4 Debugging Phase

- Use debugging tools to trace and fix issues in the assembly code.
- Optimize code for efficiency and reliability.

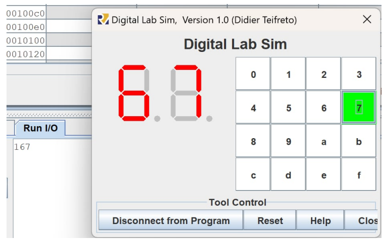
2.6 Expected Results

Upon successful completion, the project will deliver:

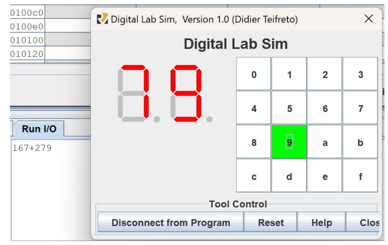
- A fully functional Simple Simple Caculator implemented in RISC-V assembly language.
- Accurate detection and processing of keypad inputs.
- Correct execution of basic arithmetic operations with appropriate result display.
- Robust error handling for invalid inputs and division by zero scenarios.
- Enhanced understanding of low-level programming and hardware interfacing.



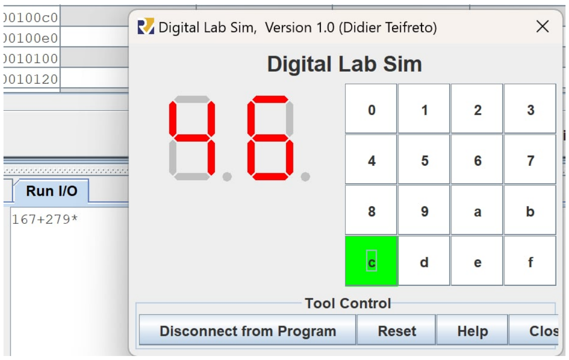
(a) Input 2-digits number



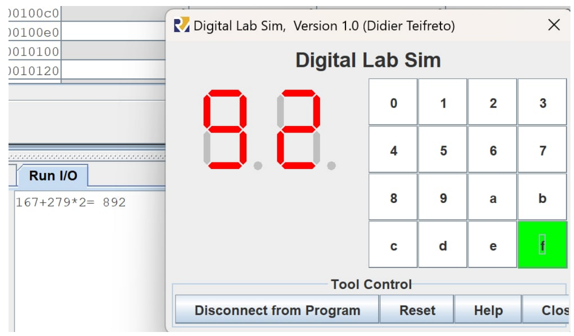
(b) input 3-digits number



(c) Input 167+279



(d) 167+279 = 446 -> display 46



(e) Last 2 digit of final result

Figure 1: Some examples illustrate the expected results implement code

2.7 Conclusion

This project provides an opportunity to delve into the intricacies of RISC-V assembly language programming, focusing on practical applications such as building a functional Simple Simple Caculator. By interfacing with hardware components like keypads and 7-segment LEDs, the project enhances understanding of memory-mapped I/O and state management in a low-level programming environment. Successfully implementing this Simple Simple Caculator will demonstrate proficiency in assembly language and foundational computer architecture concepts.

3 Project 2: RISC-V Assembly Graph Functions

3.1 Abstract

This project focuses on developing a graph plotting program using RISC-V assembly language. The program prompts the user to input coefficients for a quadratic equation, selects a color for the graph, and renders the corresponding parabola on a bitmap display. It includes robust input validation to ensure that the entered values are integers and handles invalid inputs gracefully. Additionally, the program offers the user the option to plot a new graph or exit the application. This project demonstrates the practical application of low-level programming, memory manipulation, and graphical rendering using assembly language.

3.2 Introduction

3.2.1 Background

RISC-V (Reduced Instruction Set Computing V) is an open-standard instruction set architecture (ISA) that has gained significant traction in both academic and industrial settings. Its simplicity and modularity make it an excellent choice for educational purposes, particularly in understanding the fundamentals of computer architecture and assembly language programming. This project leverages RISC-V assembly language to create a functional graph plotting tool, providing hands-on experience with low-level programming concepts and hardware interfacing.

3.2.2 Problem Statement

Creating a graph plotting application in assembly language presents several challenges:

- **User Input Handling:** Efficiently capturing and validating user inputs for coefficients and color codes.
- **Memory Management:** Manipulating bitmap memory to render graphical elements accurately.
- **Mathematical Computations:** Implementing mathematical functions, such as calculating the values of a quadratic equation.
- **Error Handling:** Ensuring the program can gracefully handle invalid inputs and provide meaningful feedback to the user.
- **User Interaction:** Providing a user-friendly interface for input prompts and options.

3.3 Objectives

The primary objectives of this project are:

- To implement user input prompts and capture integer inputs for coefficients and color codes.
- To validate user inputs to ensure they are valid integers.
- To render coordinate axes on a bitmap display.
- To plot the graph of a quadratic equation based on user-provided coefficients.
- To allow users to choose the color of the graph from predefined options.
- To provide options for users to plot new graphs or exit the program.

3.4 Project Implementation

3.4.1 Data Section

The data section defines various prompts and constants used throughout the program:

- `prompt_a`, `prompt_b`, `prompt_c`: Strings prompting the user to enter the coefficients a, b, and c of the quadratic equation.
- `prompt_color`: A string prompting the user to enter a color code from predefined options.
- `prompt_option`: A string prompting the user to choose between plotting a new graph or exiting the program.
- `error_msg`: A string displayed when the user inputs an invalid value.
- `input_buffer`: A reserved space in memory for storing user input strings.
- Color constants (`RED`, `GREEN`, `BLUE`, `WHITE`, `YELLOW`): Defined hexadecimal values representing different colors for the graph.
- `bitmap_addr`: The memory address where the bitmap display is mapped.

3.4.2 Main Function

The `main` function orchestrates the program's flow:

- **Input Collection:** Sequentially prompts the user to enter coefficients a, b, c, and a color code.
- **Input Validation:** Calls the `read_and_validate_int` subroutine to ensure each input is a valid integer.
- **Graph Drawing:** Once all inputs are validated, it proceeds to draw the coordinate axes and the graph based on the provided coefficients and color.
- **User Option Handling:** After plotting the graph, it prompts the user to either plot another graph or exit the program.

3.4.3 Input Validation

The `read_and_validate_int` subroutine is responsible for:

- **Reading Input:** Reads a string input from the user and stores it in the `input_buffer`.
- **Validation:** Checks each character to ensure it represents a valid integer (including handling negative numbers).
- **Error Handling:** Returns a special value (`-99999`) if the input is invalid, prompting the user to re-enter the value.
- **Conversion:** Converts the validated string input into an integer and returns it for further processing.

3.4.4 Drawing Coordinate Axes

The `draw_axes` subroutine draws the x-axis and y-axis on the bitmap display:

- **Y-Axis:** Plotted at the center of the screen (x = 256) by setting the corresponding bitmap pixels to white.
- **X-Axis:** Plotted at y = 256, also in white, to intersect the y-axis at the center.

This setup provides a reference frame for plotting the quadratic graph.

3.4.5 Plotting the Quadratic Graph

The `draw_graph` subroutine performs the following:

- **Initialization:** Sets the range for x values from -256 to 256 to cover the display area.
- **Calculation:** Computes the corresponding y values using the quadratic equation $y = ax^2 + bx + c$.
- **Rendering:** Converts the calculated (x, y) coordinates to bitmap addresses and colors the corresponding pixels with the user-selected color.
- **Boundary Checking:** Ensures that plotted points lie within the display boundaries to prevent memory access violations.

3.4.6 User Interaction

The `ask_continue` section handles user interaction post-plotting:

- **Prompting Options:** Asks the user to choose between plotting a new graph (input 1) or exiting the program (input 0).
- **Input Validation:** Ensures the user inputs either 0 or 1, displaying an error message for any invalid input and re-prompting as necessary.
- **Flow Control:** Depending on the user's choice, the program either restarts the plotting process or terminates gracefully.

3.5 Methodology

The project was developed through the following phases:

3.5.1 Design Phase

- **Requirement Analysis:** Defined the functional requirements, including user input handling, graph plotting, and error management.
- **Data Structure Design:** Planned the memory layout for storing prompts, error messages, input buffers, and color codes.
- **Algorithm Development:** Designed algorithms for input validation, coordinate axis drawing, and graph plotting.

3.5.2 Implementation Phase

- **Input Handling:** Implemented subroutines to prompt the user, capture input, and validate it.
- **Graph Rendering:** Developed the logic to translate mathematical equations into graphical representations on the bitmap.
- **Error Handling:** Ensured that invalid inputs are detected and appropriate messages are displayed to guide the user.
- **User Interface:** Created user prompts and options to enhance interactivity and usability.

3.5.3 Testing Phase

- **Unit Testing:** Tested individual subroutines, such as input validation and graph plotting, to ensure correctness.
- **Integration Testing:** Verified that all components work together seamlessly, from input collection to graph rendering.
- **Edge Case Testing:** Tested boundary conditions, such as maximum and minimum input values, to ensure robust error handling.

3.5.4 Debugging Phase

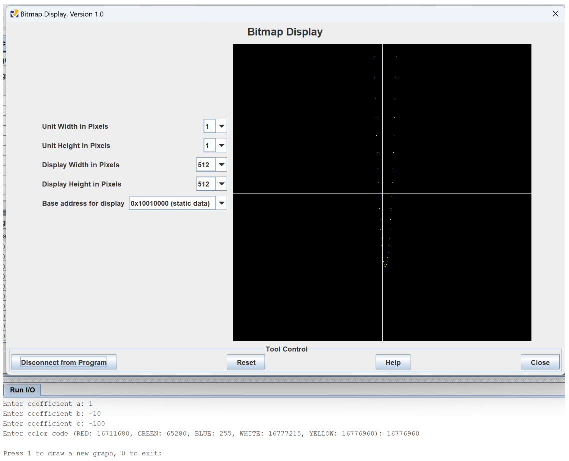
- **Code Review:** Conducted thorough reviews of the assembly code to identify and fix logical and syntactical errors.
- **Simulation:** Used RISC-V simulators to execute the program step-by-step, monitoring register values and memory states to diagnose issues.
- **Optimization:** Refined the code for efficiency, minimizing memory usage and execution time where possible.

3.6 Expected Results

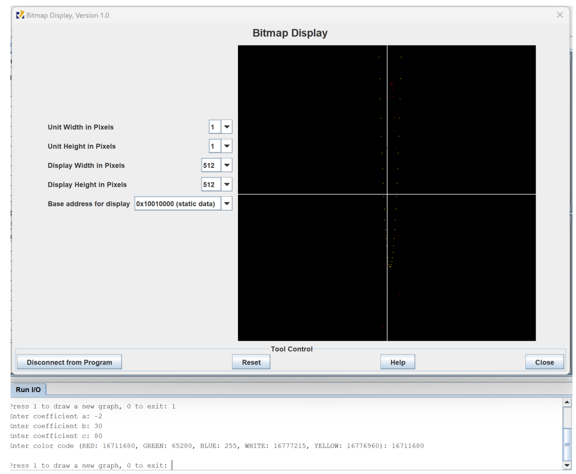
Upon successful completion, the project is expected to deliver:

- A functional graph plotting tool written in RISC-V assembly language.
- Accurate rendering of quadratic graphs based on user-provided coefficients.
- Correct handling of user inputs, including validation and error messaging.
- Flexible color selection for the plotted graph from predefined color codes.

- A user-friendly interface that allows continuous plotting or graceful exit.
- Enhanced understanding of low-level programming, memory manipulation, and graphical rendering.



(a) Graph display as user requirement (Yellow)



(b) Add 1 more graph in bitmap (Red)

```
Press 1 to draw a new graph, 0 to exit: 0

-- program is finished running (0) --
```

(c) After draw graphs, enter 0 to stop and 1 to draw a new one

Figure 2: Some examples illustrate the expected results implement code

3.7 Conclusion

This project demonstrates the feasibility of creating a graph plotting application using RISC-V assembly language. By handling user inputs, performing mathematical computations, and rendering graphics at the assembly level, the project provides deep insights into the intricacies of low-level programming and hardware interfacing. The successful implementation of this tool not only showcases proficiency in assembly language but also reinforces foundational concepts in computer architecture and memory management.

4 Overall Conclusion

The successful completion of both the RISC-V Assembly Simple Simple Caculator and Graph Functions projects underscores the practical applications of low-level programming and hardware interfacing. These projects not only demonstrate proficiency in assembly language but also enhance understanding of computer architecture, memory management, and state handling. Through meticulous design, implementation, and testing, the projects highlight the challenges and rewards of working directly with hardware components and managing system states in a constrained programming environment. The hands-on experience gained from these endeavors provides a solid foundation for further exploration and development in computer architecture and embedded systems.

5 References

- Patterson, D. A., & Hennessy, J. L. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2017.

- RISC-V Foundation. *RISC-V Specifications*. Available at: `https://riscv.org/specifications/`
- Stallings, W. *Computer Organization and Architecture: Designing for Performance*. Pearson, 2018.
- Online resources and documentation related to RISC-V assembly programming.

A RISC-V Assembly Code for Simple Simple Caculator

Listing 1: RISC-V Assembly Code for Simple Simple Caculator

```
# Define Key Codes
.eqv CODE_0           0x11    # Code for key '0'
.eqv CODE_1           0x21    # Code for key '1'
.eqv CODE_2           0x41    # Code for key '2'
.eqv CODE_3           0x81    # Code for key '3'
.eqv CODE_4           0x12    # Code for key '4'
.eqv CODE_5           0x22    # Code for key '5'
.eqv CODE_6           0x42    # Code for key '6'
.eqv CODE_7           0x82    # Code for key '7'
.eqv CODE_8           0x14    # Code for key '8'
.eqv CODE_9           0x24    # Code for key '9'
.eqv CODE_A           0x44    # Key 'A' - Add
.eqv CODE_B           0x84    # Key 'B' - Subtract
.eqv CODE_C           0x18    # Key 'C' - Multiply
.eqv CODE_D           0x28    # Key 'D' - Divide
.eqv CODE_E           0x48    # Key 'E' - Modulo
.eqv CODE_F           0x88    # Key 'F' - Equals

# Define Addresses for LED Display and Keyboard
.eqv SEVENSEG_LEFT    0xFFFF0011    # Address for left 7-segment LED
.eqv SEVENSEG_RIGHT   0xFFFF0010    # Address for right 7-segment LED
.eqv IN_ADDRESS_HEXA_KEYBOARD     0xFFFF0012    # Input address for keyboard
.eqv OUT_ADDRESS_HEXA_KEYBOARD    0xFFFF0014    # Output address for keyboard
.data
NUMS_OF_7SEG:    .word   0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F
str:             .asciiz "You have not entered a number! Please enter a number before performing calculations.\n"

.text
main:
    # Initialize Keyboard Addresses
    li      t1,     IN_ADDRESS_HEXA_KEYBOARD
    li      t2,     OUT_ADDRESS_HEXA_KEYBOARD

start:
    # Initialize Registers for Storage
    li      s0,     0    # s0: Store current key code
    li      s1,     0    # s1: Store current number or operator value
    li      s2,     0    # s2: Input type flag (1: number, 2: operator, 3: equals)
    li      s3,     0    # s3: Store the currently entered number
    li      s4,     0    # s4: Store the operator
    li      s5,     0    # s5: Store the calculation result
    li      s6,     0    # s6: Flag indicating if a number has been entered
    li      s7,     0    # s7: Flag indicating if an operator is pending

polling:
    # Scan Keyboard Rows
check_row_1:
    li      t3,     0x01
    sb      t3,     0(t1)                 # Select row 1
    lbu     a0,     0(t2)                 # Read key code
    beq     a0,     zero,    check_row_2  # If no key pressed, check next row
```

```
    bne     a0,     s0,     process_key   # If a different key is pressed, process it
    j       back_to_polling                # Otherwise, continue polling

check_row_2:
    li      t3,     0x02
    sb      t3,     0(t1)               # Select row 2
    lbu     a0,     0(t2)               # Read key code
    beq     a0,     zero,   check_row_3  # If no key pressed, check next row
    bne     a0,     s0,     process_key   # If a different key is pressed, process it
    j       back_to_polling                # Otherwise, continue polling

check_row_3:
    li      t3,     0x04
    sb      t3,     0(t1)               # Select row 3
    lbu     a0,     0(t2)               # Read key code
    beq     a0,     zero,   check_row_4  # If no key pressed, check next row
    bne     a0,     s0,     process_key   # If a different key is pressed, process it
    j       back_to_polling                # Otherwise, continue polling

check_row_4:
    li      t3,     0x08
    sb      t3,     0(t1)               # Select row 4
    lbu     a0,     0(t2)               # Read key code
    beq     a0,     zero,   process_key   # If a key is pressed, process it
    bne     a0,     s0,     process_key   # If a different key is pressed, process it
    j       back_to_polling                # Otherwise, continue polling

process_key:
    # Handle the received key
    add     s0,     zero,   a0              # Update current key code
    beq     s0,     zero,   back_to_polling  # If key code is zero, return to polling

    # Check for number keys
    li      s11,    CODE_0
    beq     s0,     s11,    process_number_0
    li      s11,    CODE_1
    beq     s0,     s11,    process_number_1
    li      s11,    CODE_2
    beq     s0,     s11,    process_number_2
    li      s11,    CODE_3
    beq     s0,     s11,    process_number_3
    li      s11,    CODE_4
    beq     s0,     s11,    process_number_4
    li      s11,    CODE_5
    beq     s0,     s11,    process_number_5
    li      s11,    CODE_6
    beq     s0,     s11,    process_number_6
    li      s11,    CODE_7
    beq     s0,     s11,    process_number_7
    li      s11,    CODE_8
    beq     s0,     s11,    process_number_8
    li      s11,    CODE_9
    beq     s0,     s11,    process_number_9

```

```
    # Check for operator keys
    li      s11,    CODE_A
    beq     s0,     s11,     process_add
    li      s11,    CODE_B
    beq     s0,     s11,     process_sub
    li      s11,    CODE_C
    beq     s0,     s11,     process_mul
    li      s11,    CODE_D
    beq     s0,     s11,     process_div
    li      s11,    CODE_E
    beq     s0,     s11,     process_mod
    li      s11,    CODE_F
    beq     s0,     s11,     process_equal

# Handle Number Keys
process_number_0:
    li      s1,     0
    li      s2,     1    # Flag indicating number input
    li      s6,     1    # Flag indicating a number has been entered
    j       after_processing

process_number_1:
    li      s1,     1
    li      s2,     1
    li      s6,     1
    j       after_processing

process_number_2:
    li      s1,     2
    li      s2,     1
    li      s6,     1
    j       after_processing

process_number_3:
    li      s1,     3
    li      s2,     1
    li      s6,     1
    j       after_processing

process_number_4:
    li      s1,     4
    li      s2,     1
    li      s6,     1
    j       after_processing

process_number_5:
    li      s1,     5
    li      s2,     1
    li      s6,     1
    j       after_processing

process_number_6:
    li      s1,     6
    li      s2,     1
    li      s6,     1
    j       after_processing

process_number_7:
    li      s1,     7
```

```
    li      s2,     1
    li      s6,     1
    j       after_processing

process_number_8:
    li      s1,     8
    li      s2,     1
    li      s6,     1
    j       after_processing

process_number_9:
    li      s1,     9
    li      s2,     1
    li      s6,     1
    j       after_processing

# Handle Operator Keys
process_add:
    li      s1,     10    # Code for addition operator
    li      s2,     2     # Flag indicating operator input
    j       after_processing

process_sub:
    li      s1,     11    # Code for subtraction operator
    li      s2,     2
    j       after_processing

process_mul:
    li      s1,     12    # Code for multiplication operator
    li      s2,     2
    j       after_processing

process_div:
    li      s1,     13    # Code for division operator
    li      s2,     2
    j       after_processing

process_mod:
    li      s1,     14    # Code for modulo operator
    li      s2,     2
    j       after_processing

process_equal:
    # Check if there is a pending operator
    beq     s7,     zero,   display_current

    # Print equal sign
    li      a0,     '='
    li      a7,     11
    ecall

    # Print space after equal sign
    li      a0,     ' '
    li      a7,     11
    ecall

    # Perform the final calculation based on the stored operator
    li      s11,    10
    beq     s4,     s11,    do_final_add
```

```
    li      s11,    11
    beq     s4,     s11,    do_final_sub
    li      s11,    12
    beq     s4,     s11,    do_final_mul
    li      s11,    13
    beq     s4,     s11,    do_final_div
    li      s11,    14
    beq     s4,     s11,    do_final_mod
    j       display_result

do_final_add:
    add     s5,     s5,     s3
    j       after_final_calc

do_final_sub:
    sub     s5,     s5,     s3
    j       after_final_calc

do_final_mul:
    mul     s5,     s5,     s3
    j       after_final_calc

do_final_div:
    beq     s3,     zero,   error_div_zero
    div     s5,     s5,     s3
    j       after_final_calc

do_final_mod:
    beq     s3,     zero,   error_div_zero
    rem     s5,     s5,     s3
    j       after_final_calc

after_final_calc:
    # Print the result
    add     a0,     zero,   s5
    li      a7,     1
    ecall

    # Print newline
    li      a0,     '\n'
    li      a7,     11
    ecall

    # Display the result on the LED
    add     a0,     zero,   s5
    jal     render

    # Reset flags
    li      s7,     0       # Clear pending operator flag
    li      s4,     15      # Indicate calculation is complete
    add     s3,     zero,   s5  # Store the result for the next calculation
    j       sleep

after_processing:
    li      s11,    1
    beq     s2,     s11,     handle_number
    li      s11,    2
    beq     s2,     s11,     handle_operator

```

```
handle_number:
    # Handle number input
    li      s11,    15
    beq     s4,     s11,     reset_Simple Simple Caculator
    j       continue_number

reset_Simple Simple Caculator:
    # Reset Simple Simple Caculator for a new operation
    li      s3,     0
    li      s4,     0
    li      s5,     0

continue_number:
    # Update current number (current_number * 10 + new_number)
    li      s11,    10
    mul     s3,     s3,     s11
    add     s3,     s3,     s1

    # Get the last two digits of s3
    li      t0,     100
    rem     a3,     s3,     t0   # a3 = s3 % 100

    j       display_number

display_number:
    # Display the number
    add     a0,     zero,   s1
    li      a7,     1
    ecall
    add     a0,     zero,   a3
    jal     render
    j       sleep

handle_operator:
    # Check if a number has been entered
    beq     s6,     zero,   error_no_operand

    # If there is a pending operator, perform it first
    beq     s7,     zero,   store_for_next

    # Perform the pending operation with the previous number
    li      s11,    10
    beq     s4,     s11,    do_pending_add
    li      s11,    11
    beq     s4,     s11,    do_pending_sub
    li      s11,    12
    beq     s4,     s11,    do_pending_mul
    li      s11,    13
    beq     s4,     s11,    do_pending_div
    li      s11,    14
    beq     s4,     s11,    do_pending_mod
    j       store_for_next

do_pending_add:
    add     s5,     s5,     s3
    j       after_pending_calc

do_pending_sub:
    sub     s5,     s5,     s3
```

```
    j       after_pending_calc

do_pending_mul:
    mul     s5,     s5,     s3
    j       after_pending_calc

do_pending_div:
    beq     s3,     zero,   error_div_zero
    div     s5,     s5,     s3
    j       after_pending_calc

do_pending_mod:
    beq     s3,     zero,   error_div_zero
    rem     s5,     s5,     s3
    j       after_pending_calc

after_pending_calc:
    # Display the calculation result
    add     a0,     zero,   s5
    jal     render
    j       store_current_op

store_for_next:
    # Store the current number as the primary operand for the next operation
    add     s5,     zero,   s3

store_current_op:
    # Store the current operator and set the pending operator flag
    add     s4,     zero,   s1
    li      s7,     1
    li      s3,     0    # Reset the current number

    # Display the operator
    li      s11,    10
    beq     s1,     s11,    print_add_op
    li      s11,    11
    beq     s1,     s11,    print_sub_op
    li      s11,    12
    beq     s1,     s11,    print_mul_op
    li      s11,    13
    beq     s1,     s11,    print_div_op
    li      s11,    14
    beq     s1,     s11,    print_mod_op
    j       sleep

print_add_op:
    li      a0,     '+'
    li      a7,     11
    ecall
    j       handle_operator_end

print_sub_op:
    li      a0,     '-'
    li      a7,     11
    ecall
    j       handle_operator_end

print_mul_op:
    li      a0,     '*'
```

```
    li      a7,     11
    ecall
    j       handle_operator_end

print_div_op:
    li      a0,     '/'
    li      a7,     11
    ecall
    j       handle_operator_end

print_mod_op:
    li      a0,     '%'
    li      a7,     11
    ecall
    j       handle_operator_end

handle_operator_end:
    li      s3,     0    # Reset the current number
    j       sleep

display_current:
    # If no operator is pending, display the current number
    add     s5,     zero,   s3
    j       after_final_calc

display_result:
    # Display the final result
    add     a0,     zero,   s5
    li      a7,     1
    ecall
    j       sleep

after_calc:
    # Display the calculation result
    li      s4,     15          # Flag indicating calculation is complete
    add     s3,     zero,   s5  # Update current number with the result

    # Print equal sign and result
    li      a0,     '='
    li      a7,     11
    ecall
    add     a0,     zero,   s5
    li      a7,     1
    ecall

    # Display the result on the LED
    add     a0,     zero,   s5
    jal     render
    j       sleep

# Function: Render - Display number on 7-segment LEDs
render:
    # Save necessary registers
    addi    sp,     sp,     -24
    sw      ra,     20(sp)
    sw      s0,     16(sp)
    sw      a0,     12(sp)
    sw      a1,     8(sp)
    sw      t0,     4(sp)
```

```
    sw      t1,     0(sp)

    # Split the number into tens and units
    li      t0,     10
    mv      t1,     a0
    div     t1,     t1,     t0      # Get the tens place
    rem     a0,     a0,     t0      # Get the units place

    # Display the units place on the right LED
    li      a1,     SEVENSEG_RIGHT
    jal     ra,     show_digit

    # Display the tens place on the left LED
    rem     a0,     t1,     t0
    li      a1,     SEVENSEG_LEFT
    jal     ra,     show_digit

    # Restore saved registers
    lw      t1,     0(sp)
    lw      t0,     4(sp)
    lw      a1,     8(sp)
    lw      a0,     12(sp)
    lw      s0,     16(sp)
    lw      ra,     20(sp)
    addi    sp,     sp,     24
    jr      ra

# Function: show_digit - Display a single digit on a 7-segment LED
show_digit:
    # Save registers
    addi    sp,     sp,     -12
    sw      ra,     8(sp)
    sw      t0,     4(sp)
    sw      t1,     0(sp)

    # Get the LED segment code and display it
    la      t0,     NUMS_OF_7SEG
    slli    t1,     a0,     2       # Multiply by 4 to get the array offset
    add     t0,     t0,     t1
    lw      t0,     0(t0)
    sb      t0,     0(a1)               # Write the segment code to the LED

    # Restore registers
    lw      t1,     0(sp)
    lw      t0,     4(sp)
    lw      ra,     8(sp)
    addi    sp,     sp,     12
    jr      ra

# Error Handling
error_no_operand:
    # Display error message for missing operand
    la      a0,     str
    li      a7,     4
    ecall
    j       sleep

error_div_zero:
    # Display error message for division by zero
```

```
    li      a0,     'E'     # Display 'E' for error
    li      a7,     11
    ecall
    j       sleep

sleep:
    # Wait for 100ms to prevent continuous key presses
    li      a0,     100
    li      a7,     32
    ecall

back_to_polling:
    j       polling
```

B RISC-V Assembly Code for Graph Functions

Listing 2: RISC-V Assembly Code for Graph Functions

```
.data
prompt_a:       .asciz "Enter coefficient a: "
prompt_b:       .asciz "Enter coefficient b: "
prompt_c:       .asciz "Enter coefficient c: "
prompt_color:   .asciz "Enter color code (RED: 16711680, GREEN: 65280, BLUE: 255, WHITE: 16777215, YELLOW: 16776960): "
prompt_option:  .asciz "\nPress 1 to draw a new graph, 0 to exit: "
.eqv bitmap_addr 0x10010000
.eqv RED 0x00FF0000
.eqv GREEN 0x0000FF00
.eqv BLUE 0x000000FF
.eqv WHITE 0x00FFFFFF
.eqv YELLOW 0x00FFFF00

.text
.globl main

main:
    # Display prompt and get coefficient a
    li a7, 4
    la a0, prompt_a
    ecall

    li a7, 5
    ecall
    mv s0, a0               # Store a in s0

    # Display prompt and get coefficient b
    li a7, 4
    la a0, prompt_b
    ecall

    li a7, 5
    ecall
    mv s1, a0               # Store b in s1

    # Display prompt and get coefficient c
    li a7, 4
    la a0, prompt_c
```

```
    ecall

    li a7, 5
    ecall
    mv s2, a0               # Store c in s2

    # Display prompt and get color code
    li a7, 4
    la a0, prompt_color
    ecall

    li a7, 5
    ecall
    mv s3, a0               # Store color code in s3

# ------------------------- Draw Coordinate Axes -------------------------
draw_axes:
    # Draw the vertical axis (x = 256)
    li t0, 0                # y = 0
    li t1, 512              # Height limit for the axis
    li t4, 2048             # Width of the bitmap in pixels
    li t2, 1024             # Vertical axis is at the center (x = 256 * 4 bytes =
        1024)

draw_y_axis_loop:
    li t3, bitmap_addr
    mul t5, t0, t4          # t5 = y * 2048
    add t5, t5, t2          # t5 = y * 2048 + 1024
    add t3, t3, t5          # t3 = base bitmap address + (y * width + x)

    li s8, WHITE            # Set color to WHITE
    sw s8, 0(t3)            # Draw white pixel

    addi t0, t0, 1          # Increment y
    blt t0, t1, draw_y_axis_loop

    # Draw the horizontal axis (y = 1024)
    li t0, 0                # x = 0
    li t1, 512              # Width limit for the axis
    li t2, 256              # Horizontal axis is at the center (y = 256)
    li t4, 2048             # Width of the bitmap in pixels

draw_x_axis_loop:
    li t3, bitmap_addr
    mul t5, t2, t4          # t5 = 256 * 2048
    li s7, 4                # Each pixel is 4 bytes
    mul s7, t0, s7          # s7 = 4 * x
    add t5, t5, s7          # t5 = 256 * 2048 + 4 * x
    add t3, t3, t5          # t3 = base bitmap address + (y * width + x)

    li s8, WHITE            # Set color to WHITE
    sw s8, 0(t3)            # Draw white pixel

    addi t0, t0, 1          # Increment x
    blt t0, t1, draw_x_axis_loop

# ------------------------- Draw Quadratic Function Graph
    -------------------------
draw_graph:
```

```
    # Set initial x range from -256 to 256
    li t0, -256
    li t1, 256
    li a1, 2048             # Bitmap width in pixels
    li a6, 513              # Offset to detect out-of-bound pixels
    li a3, 4                # 4 bytes per pixel

    # Calculate the address of (0,0) on the bitmap
    mul s5, t1, a1          # s5 = 2048 * 256
    addi s5, s5, 1024       # s5 = 2048 * 256 + 1024
    li s4, bitmap_addr
    add s5, s5, s4          # s5 points to (0,0) on the bitmap

    mul a6, a1, a6          # a6 = 2048 * 513
    add a6, a6, s4          # a6 = bitmap address + 2048 * 513 (lowest point
        outside the bitmap)

draw_graph_loop:
    # Compute y = a*x^2 + b*x + c
    mul t2, t0, t0          # t2 = x^2
    mul t3, s0, t2          # t3 = a * x^2
    mul t4, s1, t0          # t4 = b * x
    add t5, t3, t4          # t5 = a * x^2 + b * x
    add t5, t5, s2          # t5 = a * x^2 + b * x + c = y

    # Calculate the bitmap address for (x, y)
    li t2, 2048             # Width in pixels
    li t3, 4                # 4 bytes per pixel
    mul t2, t2, t5          # t2 = 2048 * y
    sub t2, s5, t2          # t2 = (0,0) - 2048 * y
    mul t4, t3, t0          # t4 = 4 * x
    add t2, t2, t4          # t2 = (0,0) - 2048 * y + 4 * x

    # Check if the pixel is within the bitmap boundaries
    blt t2, s4, skip_point
    bge t2, a6, skip_point

    # Set the selected color
    mv s8, s3
    sw s8, 0(t2)            # Draw the pixel

skip_point:
    addi t0, t0, 1          # Increment x
    ble t0, t1, draw_graph_loop

# Prompt the user to continue or exit
    li a7, 4
    la a0, prompt_option
    ecall

    li a7, 5
    ecall
    beq a0, zero, exit      # If input is 0, exit
    j main                  # Otherwise, restart the program

# ------------------------ Exit Program ------------------------
exit:
    li a7, 10
    ecall
```