Computer Architecture Lab

Lecturer: Dr. Le Ba Vui

Class ID: 152003

Final Project Report

AUTHORS

Vu Hai Dang - 20225962
Nguyen Minh Khoi - 20226050

Contents

1 Simple Calculator

1.1 Description

Use Key Matrix and 7-segments LEDs to implement a simple calculator that support +, -, *, /, integer operands.

- Press a for addition
- Press b for subtraction
- Press c for multiplication
- Press d for division
- Press e for division with remainder
- Press f to get the result

Detail requirements:

- When pressing digital key, show the last two digits on LEDs. For example, press 1 -> show 01,press 2 -> show 12, press 3 -> show 23.
- After entering an operand, press + - * /
- After pressing f (=) , calculate and show two digits at the right of the result on LEDs.
- Can calculate continuously (use Calculator on Windows for reference)

1.2 Idea

This program simulates a simple calculator, allowing users to input numbers and operators via a hex keypad and display the results on a 7-segment LED screen. It includes the following main functions:

- Scanning the hex keypad to detect pressed keys.
- Processing key codes corresponding to numbers (0-9) and operators (+, -, *, /,
- Displaying values and results on the 7-segment LED.

1.3 Function

1.3.1 Data

- SEVENSEG_LEFT and SEVENSEG_RIGHT: Memory addresses controlling the left and right 7-segment LEDs to display numbers.
- IN_ADDRESS_HEXA_KEYBOARD and OUT_ADDRESS_HEXA_KEYBOARD: Memory addresses for reading input from the hexadecimal keyboard and writing output back.

Codes representing numbers (0-9) and operations (addition, subtraction, multiplication, division, modulo, and equals) that the user can press on the keyboard. Each key is mapped to a unique code, such as:

- CODE_0 = 0x11 for the number 0
- CODE_ADD = 0x44 for the addition operation
- CODE_SUB = 0x84 for subtraction, and so on.

```
# Define LED and Keyboard Addresses
.eqv SEVENSEG_LEFT    0xFFFF0011    # Left 7—segment LED address
.eqv SEVENSEG_RIGHT   0xFFFF0010    # Right 7—segment LED address
.eqv IN_ADDRESS_HEXA_KEYBOARD     0xFFFF0012    # Keyboard input address
.eqv OUT_ADDRESS_HEXA_KEYBOARD    0xFFFF0014    # Keyboard output address

# Key Code Definitions
.eqv CODE_0           0x11    # Code for number 0
.eqv CODE_1           0x21    # Code for number 1
.eqv CODE_2           0x41    # Code for number 2
.eqv CODE_3           0x81    # Code for number 3
.eqv CODE_4           0x12    # Code for number 4
.eqv CODE_5           0x22    # Code for number 5
.eqv CODE_6           0x42    # Code for number 6
.eqv CODE_7           0x82    # Code for number 7
.eqv CODE_8           0x14    # Code for number 8
.eqv CODE_9           0x24    # Code for number 9
.eqv CODE_ADD          0x44    # Key 'a' — Addition
.eqv CODE_SUB           0x84    # Key 'b' — Subtraction
.eqv CODE_MUL           0x18    # Key 'c' — Multiplication
.eqv CODE_DIV           0x28    # Key 'd' — Division
.eqv CODE_MOD           0x48    # Key 'e' — Modulo
.eqv CODE_EQUAL           0x88    # Key 'f' — Equals
```

```
.data
VALUE_7SEGMENT:    .word   0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F
   , 0x6F
message1:             .asciz "You haven't entered a number. Please enter a
   number before performing calculation.\n"
message2:             .asciz "ERROR FOR DIVISION ZERO\n"
```

1.3.2 Polling (Keypad Scanning Loop)

Functionality:

- Acts as the main loop, scanning and checking each row of the hex keypad.
- Detects whether any key is pressed.
- If a key is detected, it reads the corresponding scan code and transitions to the key code handling function.

Details:

- Each row of the keypad is activated sequentially.
- If a keypress is detected, its scan code is decoded and processed.
- When no key is pressed, the program continues the polling loop.

```
    polling:
    # Scan keyboard rows
check_row_1:
    li      t3,     0x01
    sb      t3,     0(t1)
    lbu     a0,     0(t2)
    beq     a0,     zero,    check_row_2
    bne     a0,     s0,      process_key
    j       back_to_polling

check_row_2:
    li      t3,     0x02
    sb      t3,     0(t1)
    lbu     a0,     0(t2)
    beq     a0,     zero,    check_row_3
```

```
    bne     a0,     s0,      process_key
    j       back_to_polling

check_row_3:
    li      t3,     0x04
    sb      t3,     0(t1)
    lbu     a0,     0(t2)
    beq     a0,     zero,    check_row_4
    bne     a0,     s0,      process_key
    j       back_to_polling

check_row_4:
    li      t3,     0x08
    sb      t3,     0(t1)
    lbu     a0,     0(t2)
    beq     a0,     zero,    process_key
    bne     a0,     s0,      process_key
    j       back_to_polling

# Number process functions
```

1.3.3 Code Processing

Functionality:

- Determines the action to be performed on the basis of the read key code.
- Classifies the key code into groups: numeric keys (from 0 to 9), operator keys (+, -, *, /, mode, =), or other special keys.

Main Operations:

- **Numeric keys (from 0 to 9):** Redirects to a specific function to update the operand.
- **Operator keys (+, -, *, /, mode, =):** Redirects to a function to set the operator or calculate the result.
- **Other operations (if any):** Redirects to error handling or additional functionality.

```
    process_key:
    add     s0,     zero,    a0
    beq     s0,     zero,    back_to_polling
```

```
    li      s11,    CODE_0
    beq     s0,     s11,     process_0
    li      s11,    CODE_1
    beq     s0,     s11,     process_1
    li      s11,    CODE_2
    beq     s0,     s11,     process_2
    li      s11,    CODE_3
    beq     s0,     s11,     process_3
    li      s11,    CODE_4
    beq     s0,     s11,     process_4
    li      s11,    CODE_5
    beq     s0,     s11,     process_5
    li      s11,    CODE_6
    beq     s0,     s11,     process_6
    li      s11,    CODE_7
    beq     s0,     s11,     process_7
    li      s11,    CODE_8
    beq     s0,     s11,     process_8
    li      s11,    CODE_9
    beq     s0,     s11,     process_9

    # Operator process functions
    li      s11,    CODE_ADD
    beq     s0,     s11,     process_add
    li      s11,    CODE_SUB
    beq     s0,     s11,     process_sub
    li      s11,    CODE_MUL
    beq     s0,     s11,     process_mul
    li      s11,    CODE_DIV
    beq     s0,     s11,     process_div
    li      s11,    CODE_MOD
    beq     s0,     s11,     process_mod
    li      s11,    CODE_EQUAL
    beq     s0,     s11,     process_equal
```

1.3.4 Process Code

Each process (process_1 to process_9) represents a numerical input from 1 to 9. These functions assign values to registers s1, s2, and s6:

- s1: Stores the number (1 to 9).
- s2: Marks the input as a number (1 for number input, 2 for operator input).
- s6: A flag for handling input state.

After setting the registers, each process jumps to after_process to continue. The functions process_add, process_sub, process_mul, process_div, and process_mod handle the operator input for each corresponding arithmetic operation. The operator code is stored in s1, and s2 is set to 2 to mark it as an operator:

- s1: Operator code (e.g., 10 for addition, 11 for subtraction).
- s2: Marks this as an operator input.

Checks if there is a pending operation stored in s7. If there is no pending operation, it directly displays the current input. If there is a pending operation, it prints the equals sign and a space.

Based on the operator code in s4, the function performs the corresponding arithmetic operation on the stored numbers:

- final_add, final_sub, final_mul, final_div, and final_mod handle their respective operations (addition, subtraction, multiplication, division, and modulo).

After performing the operation, the result is printed and displayed on the 7-segment LED. The flags (s7 and s4) are reset and the result is saved in s3 for use in the next operation.

```
process_1:
    li      s1,     1
    li      s2,     1
    li      s6,     1
    j       after_process

process_2:
    li      s1,     2
    li      s2,     1
    li      s6,     1
    j       after_process

process_3:
```

```
    li      s1,     3
    li      s2,     1
    li      s6,     1
    j       after_process

process_4:
    li      s1,     4
    li      s2,     1
    li      s6,     1
    j       after_process

process_5:
    li      s1,     5
    li      s2,     1
    li      s6,     1
    j       after_process

process_6:
    li      s1,     6
    li      s2,     1
    li      s6,     1
    j       after_process

process_7:
    li      s1,     7
    li      s2,     1
    li      s6,     1
    j       after_process

process_8:
    li      s1,     8
    li      s2,     1
    li      s6,     1
    j       after_process

process_9:
    li      s1,     9
    li      s2,     1
    li      s6,     1
    j       after_process
```

```
# Operator process functions
process_add:
    li      s1,     10    # Code for addition
    li      s2,     2     # Mark as operator input
    j       after_process

process_sub:
    li      s1,     11    # Code for sub
    li      s2,     2
    j       after_process

process_mul:
    li      s1,     12    # Code for mul
    li      s2,     2
    j       after_process

process_div:
    li      s1,     13    # Code for div
    li      s2,     2
    j       after_process

process_mod:
    li      s1,     14    # Code for mod
    li      s2,     2
    j       after_process

process_equal:
    # Check if there's a pending operation
    beq     s7,     zero,   display_current

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

    # Print space
    li      a0,     ' '
    li      a7,     11
    ecall
```

```
    # Perform final calculation based on stored operator
    li      s11,    10
    beq     s4,     s11,    final_add
    li      s11,    11
    beq     s4,     s11,    final_sub
    li      s11,    12
    beq     s4,     s11,    final_mul
    li      s11,    13
    beq     s4,     s11,    final_div
    li      s11,    14
    beq     s4,     s11,    final_mod
    j       display_result
final_add:
    add     s5,     s5,     s3
    j       after_final_calc

final_sub:
    sub     s5,     s5,     s3
    j       after_final_calc

final_mul:
    mul     s5,     s5,     s3
    j       after_final_calc

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

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

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

```
# Newline
li      a0,     '\n'
li      a7,     11
ecall

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

# Reset flags
li      s7,     0       # Clear pending operation flag
li      s4,     15      # Mark calculation complete
add     s3,     zero,   s5  # Save result for next calculation
j       sleep
```

1.3.5 After Processing Code

a, handle_number
Determines the type of input based on `s2`:

- `1`: Number input → Jump to `handle_number`.
- `2`: Operator input → Jump to `handle_operator`.

Processes numerical inputs:

- If the calculator is reset (`s4 = 15`), clear all values.
- Updates the current number by multiplying the previous value by 10 and adding the new digit.
- Takes the last two digits for display (`s3 % 100`).
- Displays the updated number using system calls and displays it on the 7-segment LED.

b, handle_operator
Processes operator inputs:

- If no number is entered (`s6 = 0`), jumps to an error.
- If a pending operation exists (`s7 = 1`), performs the pending operation (e.g., addition, subtraction).

- Otherwise, stores the current number (`s3`) as the first operand and the operator (`s1`) for the next calculation.

c, Pending Operation Handling

- Executes the pending operation (`do_pending_add`, `do_pending_sub`, etc.).
- Updates the result (`s5`) and displays it temporarily.
- Resets `s3` for the next number input.

```
    after_process:
    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_calculator
    j       continue_number

reset_calculator:
    # Reset calculator for new computation
    li      s3,     0
    li      s4,     0
    li      s5,     0

continue_number:
    # Compute new number (previous * 10 + new digit)
    li      s11,    10
    mul     s3,     s3,     s11
    add     s3,     s3,     s1

    # Take two last digit
    li      t0,     100
    rem     a3,     s3,     t0   # a3 = s3 % 100

    j       display_number
```

```
display_number:
    # Display number
    add     a0,     zero,   s1
    li      a7,     1
    ecall
    add     a0,     zero,   s3
    jal     render
    j       sleep

handle_operator:
    # Check if an operand is entered
    beq     s6,     zero,   error_no_operand

    # If pending operation exists, compute it first
    beq     s7,     zero,   store_for_next

    # Perform pending operation
    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 temporary result
    add     a0,     zero,   s5
    jal     render
    j       store_current_op

store_for_next:
    # Store first operand for next operation
    add     s5,     zero,   s3

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

    # Display 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 current number
    j       sleep

display_current:
    # If there is no pending operation, 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
```

1.3.6 Display 7-segment LED Function

- The `display` function displays the 7-segment LED representation of an input integer.
- Always shows the tens and units digits.
- Stack operations:
  - Save the program addresses into a stack in the label `display7Seg_store`.
  - After completing tasks, retrieve the saved addresses, restore the registers, and close the stack.
- The main task:
  - At the label `display7Seg_do`, divide the number to be displayed by 10 to get the units digit, store it, and call the `show_digit` function.
  - Similarly, divide the number to get the tens digit, store it, and call `show_digit`.
- Access the predeclared array `NUMS_OF_7SEG` to fetch the 7-segment LED display code corresponding to the number.
- Send the code to the 7-segment LED address for display.

```
# Display function — display number on 7—segment LED
display7Seg:
display7Seg_store:
    addi    sp, sp, —24               # Expand the stack
    sw      ra, 20(sp)                # Save the return address
    sw      s0, 16(sp)                # Save the value of register s0
    sw      a0, 12(sp)                # Save the value of parameter a0 (integer
       to display)
    sw      a1, 8(sp)                 # Save the value of parameter a1 (7—
       segment LED address)
    sw      t0, 4(sp)                 # Save the value of register t0
    sw      t1, 0(sp)                 # Save the value of register t1
```

```
display7Seg_do:
    li      t0, 10                  # Load the value 10 into register t0
    mv      t1, a0                  # Copy the value of parameter a0 into
       register t1
    rem     a0, a0, t0              # Get the remainder of a0 divided by 10 (
       units digit)
    li      a1, SEVENSEG_RIGHT      # Set the address of the right 7-segment
       LED into a1
    jal     ra, show_digit          # Call the function show_digit to display
       the digit

    div     t1, t1, t0              # Get the integer division of t1 by 10
    rem     a0, t1, t0              # Get the remainder of t1 divided by 10 (
       tens digit)
    li      a1, SEVENSEG_LEFT       # Set the address of the left 7-segment
       LED into a1
    jal     ra, show_digit          # Call the function show_digit to display
       the digit

display7Seg_load:
    lw      t1, 0(sp)               # Load the value of register t1 from the
       stack
    lw      t0, 4(sp)               # Load the value of register t0 from the
       stack
    lw      a1, 8(sp)               # Load the value of parameter a1 from the
       stack
    lw      a0, 12(sp)              # Load the value of parameter a0 from the
       stack
    lw      s0, 16(sp)              # Load the value of register s0 from the
       stack
    lw      ra, 20(sp)              # Load the return address from the stack
    addi    sp, sp, 24              # Shrink the stack
    jr      ra                      # Return to the caller

# Show_digit function - display a single digit on the 7-segment LED
show_digit:
    # Save registers
    addi    sp,     sp,     -12
    sw      ra,     8(sp)
```

```
    sw      t0,     4(sp)
    sw      t1,     0(sp)

    # Fetch corresponding LED code and display
    la      t0,     VALUE_7SEGMENT
    slli    t1,     a0,     2       # Multiply by 4 for offset
    add     t0,     t0,     t1
    lw      t0,     0(t0)
    sb      t0,     0(a1)

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

1.3.7 Error Handling

The program manages two primary error cases:

- **Missing Operand Error (`error_no_operand`):** If an operator is entered without a preceding number, the program displays an error message and halts, waiting for the next input.
- **Division by Zero Error (`error_div_zero`):** If a division or modulo operation is attempted with zero as the divisor, an error message is shown, and the program pauses for the next input.

```
error_no_operand:
    # Display error for missing number input
    la      a0,     message1
    li      a7,     4
    ecall
    j       sleep

error_div_zero:
    # Display error for division by zero
    la      a0,     message2 # Display for error
    li      a7,     4
    ecall
    j       sleep
```

1.4 Output

Case 1: Separate computation

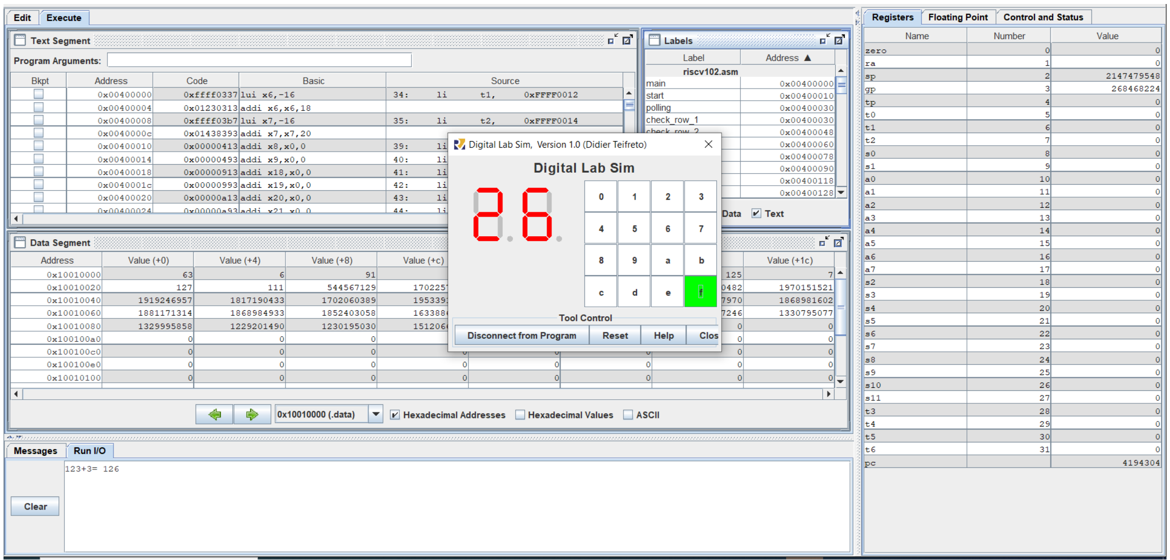

Figure 1: The two last digits of the computation, 26, are displayed on the LED.

Case 2: Complex computation

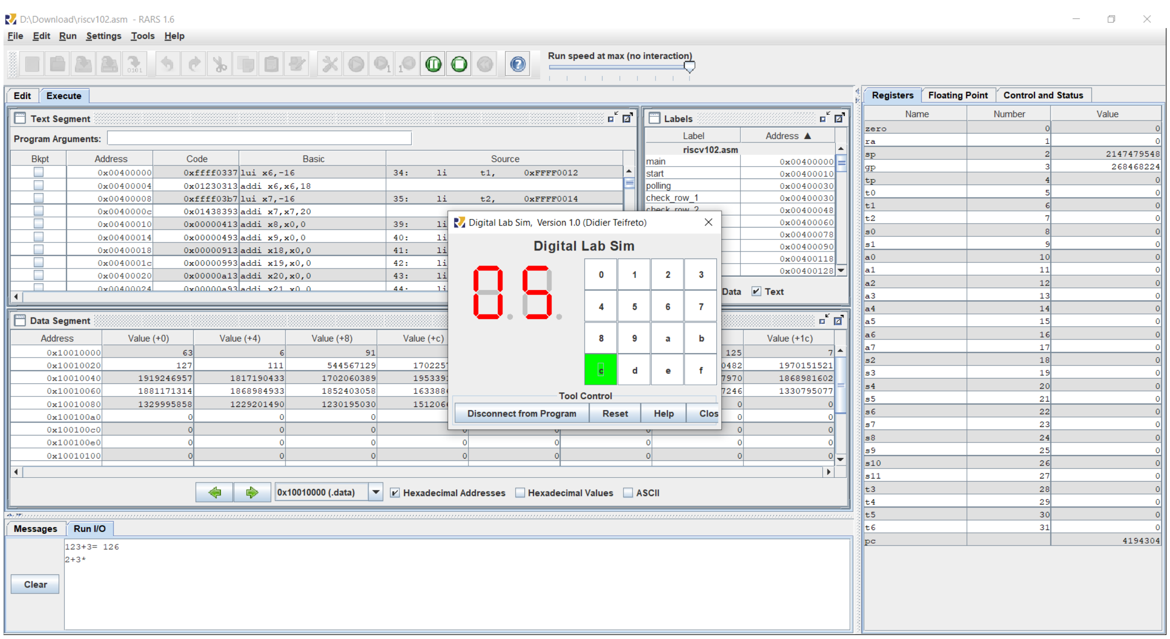

Figure 2: Step 1: 2 + 3 = 5

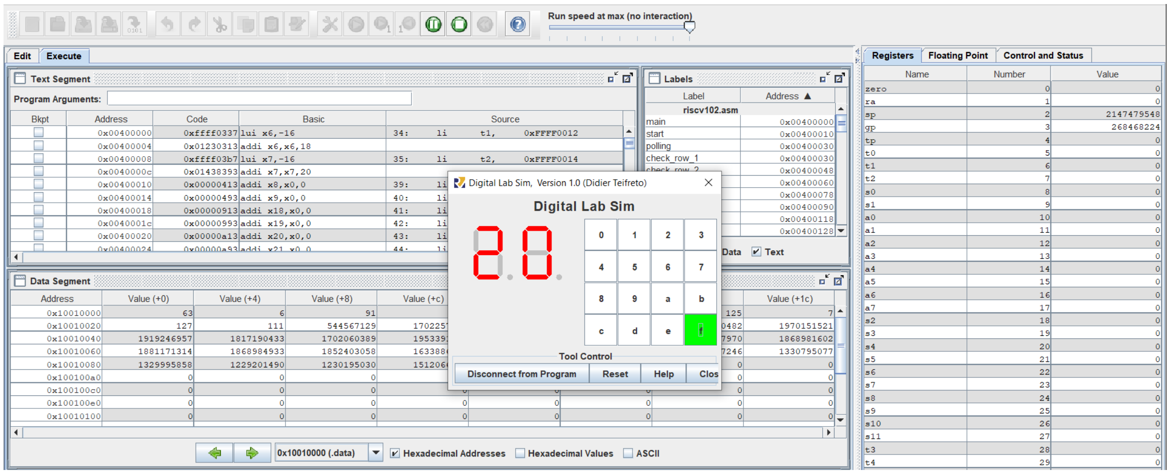

Figure 3: Step 2: (2+3)*4 = 20

Case 3: Error Handling

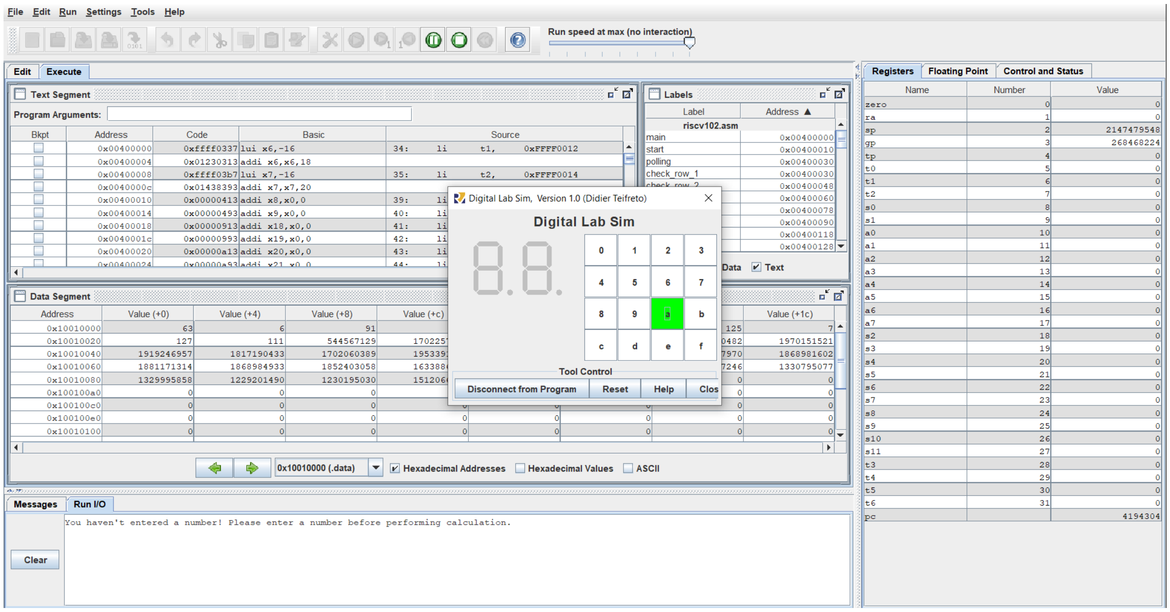

Figure 4: If an operator is entered without a number, the program displays an error message

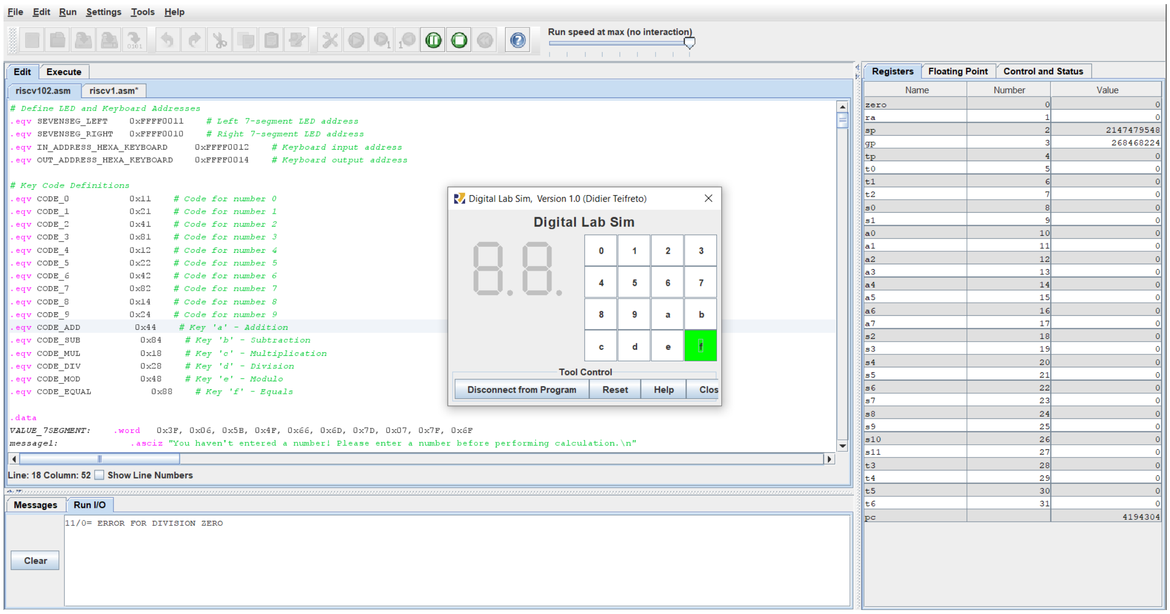

Figure 5: Error for division with zero

2 Read the BMP file and display on the Bitmap Display

2.1 Description

Read and display the BMP image file on the Bitmap Display. The maximum resolution of the image is 512x512.

- Research about the system calls to open and read file.
- Research about the BMP file format.
- The program lets the user enter the path of a BMP image file from Run I/O.

2.2 Idea

The program reads and displays a BMP image file with a maximum resolution of 512×512 pixels on a bitmap display. The main steps of the program are as follows:

1. **Input BMP file path:**
    - Print a prompt asking the user to enter the file path.
    - Read the file path and remove the newline character.

2. **Open the file and verify the BMP format:**
   - Open the file in read-only mode.
   - Read the first 54 bytes of the file (BMP header).
   - Check the first two bytes to confirm the BMP format (‘’B’‘ and ‘’M’‘).
3. **Check image dimensions:**
   - Extract the image width and height from the header (offsets 18 and 22).
   - Verify that the image dimensions do not exceed 512×512 pixels.
4. **Read pixel data from the file:**
   - Determine the offset to the pixel data (stored at byte 10 in the header).
   - Calculate the row size, including padding, to ensure alignment to 4-byte boundaries.
   - Read all pixel data into a buffer.
5. **Display the image on the bitmap screen:**
   - Traverse the pixel data from bottom to top (BMP stores images in reverse row order).
   - Convert the pixel format from BGR to RGB.
   - Write the pixel data to the bitmap display memory.

2.3 Function

2.3.1 Data

- `MONITOR_SCREEN`: Memory address for displaying pixel data on the bitmap screen.
- `buffer`: A buffer of 256 bytes to store the BMP file path entered by the user.
- `big_buffer`: A large buffer (1.1 MB) to store the BMP file header and pixel data.
- `error_size`, `error_type`, and `error_open`: Strings to display error messages for invalid file formats, oversized images, or file opening issues.

```
# Data definitions
prompt:        .asciz "Enter BMP file path: "
error_size:    .asciz "Error: Image size exceeds 512x512.\n"
error_type:    .asciz "Error: Not a BMP file.\n"
error_open:    .asciz "Error: Cannot open file. Check file path and permissions
    .\n"
```

```
buffer:         .space 256          # Buffer to hold the input file path
big_buffer:     .space 1100000      # Large buffer for header and pixel data

# Screen Address
.eqv MONITOR_SCREEN, 0x10010000
```

2.3.2 File Opening and Header Validation

Functionality:

- Open the BMP file specified by the user and validate its header.
- Ensure the file is a valid BMP by checking the first two bytes (`'B'` and `'M'`). [1]
- Read image width and height from the header and validate the dimensions.

Implementation Details:

- The syscall `open` is used to open the file in read-only mode.
- The syscall `read` reads the first 54 bytes of the BMP file into `big_buffer`. [2]
- The first two bytes are checked to ensure the file type is BMP.
- The image width and height are extracted from offsets 18 and 22, respectively, and checked against the 512×512 limit. [3]

```
# Open the BMP file
li a7, 1024             # Syscall: open
la a0, buffer           # File path
li a1, 0                # Read-only mode
ecall
mv t0, a0               # File descriptor
blt t0, zero, open_file_error

# Read BMP header (54 bytes)
li t1, 54
la a1, big_buffer
mv a0, t0
mv a2, t1
li a7, 63               # Syscall: read
ecall
blt a0, t1, error       # Error if less than 54 bytes read
```

```
# Check BMP signature ('B' and 'M')
la t2, big_buffer
lbu t3, 0(t2)
lbu t4, 1(t2)
li t5, 'B'
bne t3, t5, type_error
li t5, 'M'
bne t4, t5, type_error

# Extract width and height
addi t3, t2, 18
lw t4, 0(t3)          # t4 = width
addi t3, t2, 22
lw t5, 0(t3)          # t5 = height
li t6, 512
bgt t4, t6, size_error
bgt t5, t6, size_error
```

2.3.3 Pixel Data Processing and Display

Functionality:

- Extract pixel data from the BMP file based on the pixel data offset in the header.
- Display the image on the bitmap screen by traversing the pixel data in reverse row order (bottom-to-top).
- Convert pixel format from BGR (BMP standard) to RGB (display standard).

Implementation Details:

- The pixel data offset is read from byte 10 in the header.
- The file pointer is moved to the pixel data location using syscall `lseek`.
- The row size is calculated, including padding to align each row to a 4-byte boundary.
- Pixel values are read and displayed row-by-row in reverse order.

```
# Get pixel data offset
addi t3, t2, 10
lw t6, 0(t3)          # t6 = pixel data offset

# Move file pointer to pixel data
mv a0, t0
mv a1, t6
li a2, 0              # SEEK_SET
li a7, 62             # Syscall: lseek
ecall

# Calculate row size (padded)
li s10, 3
mul s7, t4, s10       # rowSize = width * 3 (3 bytes per pixel)
addi s7, s7, 3
li s9, -4
and s7, s7, s9        # Align to 4-byte boundary

# Read pixel data
mul s8, s7, t5        # Total size = rowSize * height
la a1, big_buffer
mv a0, t0
mv a2, s8
li a7, 63             # Syscall: read
ecall
```

2.3.4 Bottom-to-Top Display Loop

Functionality:

- Display pixel data row by row from bottom to top, as BMP stores rows in reverse order.
- Each pixel's BGR values are converted to RGB before displaying on the screen.

Implementation:

```
# Display pixels
li a3, MONITOR_SCREEN
mv s1, t5            # s1 = height
mv s2, t4            # s2 = width
```

```
loop_rows:
    addi s1, s1, -1        # s1--
    blt s1, zero, done     # If s1 < 0, exit
    mul s9, s1, s7         # Calculate row_start
    la t3, big_buffer
    add t3, t3, s9
    mv s4, s2              # s4 = width
    mv s5, t3              # s5 = row_start

loop_cols:
    beqz s4, next_row
    lbu t1, 0(s5)    # B
    lbu t2, 1(s5)    # G
    lbu s11, 2(s5)   # R
    slli s11, s11, 16
    slli t2, t2, 8
    or s11, s11, t2
    or s11, s11, t1
    sw s11, 0(a3)    # Write RGB to screen
    addi a3, a3, 4
    addi s5, s5, 3
    addi s4, s4, -1
    j loop_cols

next_row:
    j loop_rows

done:
    # Close file and exit
    mv a0, t0
    li a7, 57        # Syscall: close
    ecall
    li a0, 0
    li a7, 10        # Syscall: exit
    ecall
```

2.3.5 Error Handling

Error Cases:

- **Invalid File Format:** Display an error if the file does not start with `'B'` and `'M'`.

- **Image Size Exceeds Limit:** Display an error if the image dimensions exceed 512×512 pixels.
- **File Open Failure:** Display an error if the file cannot be opened.

```
open_file_error:
    li a7, 4
    la a0, error_open
    ecall
    j error

size_error:
    li a7, 4
    la a0, error_size
    ecall
    j error

type_error:
    li a7, 4
    la a0, error_type
    ecall
    j error

error:
    li a0, 0
    li a7, 10
    ecall
```

2.4 Output

Case 1: 512 x 512 resolution

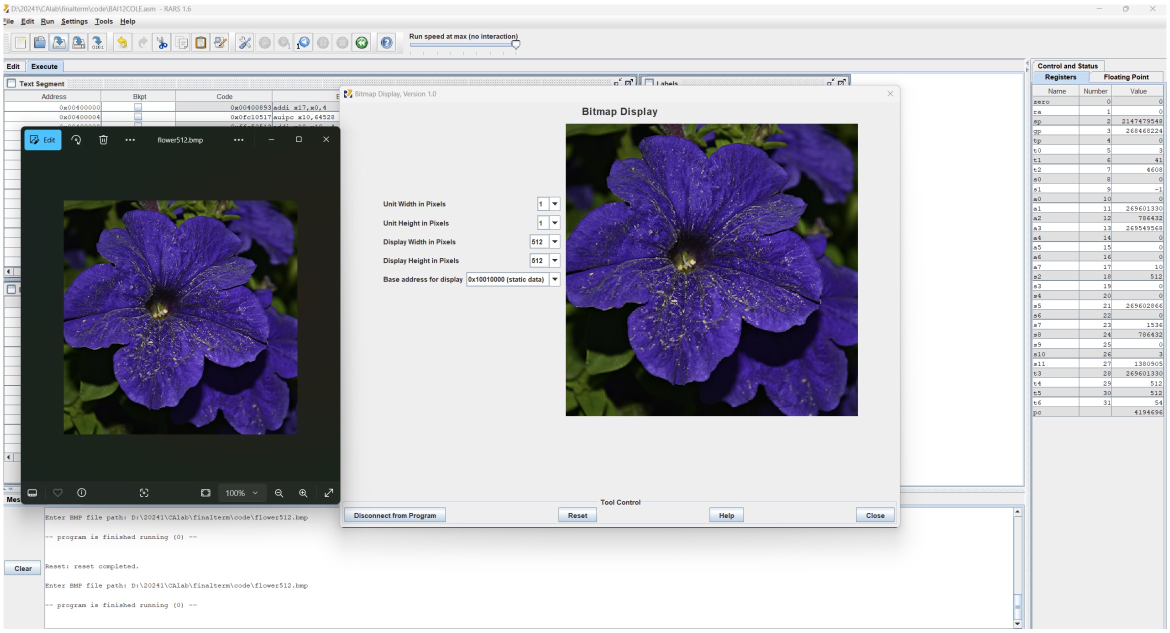

Figure 6: Displaying a BMP image with 512×512 resolution on the Bitmap Display

Case 2: 256 x 256 resolution

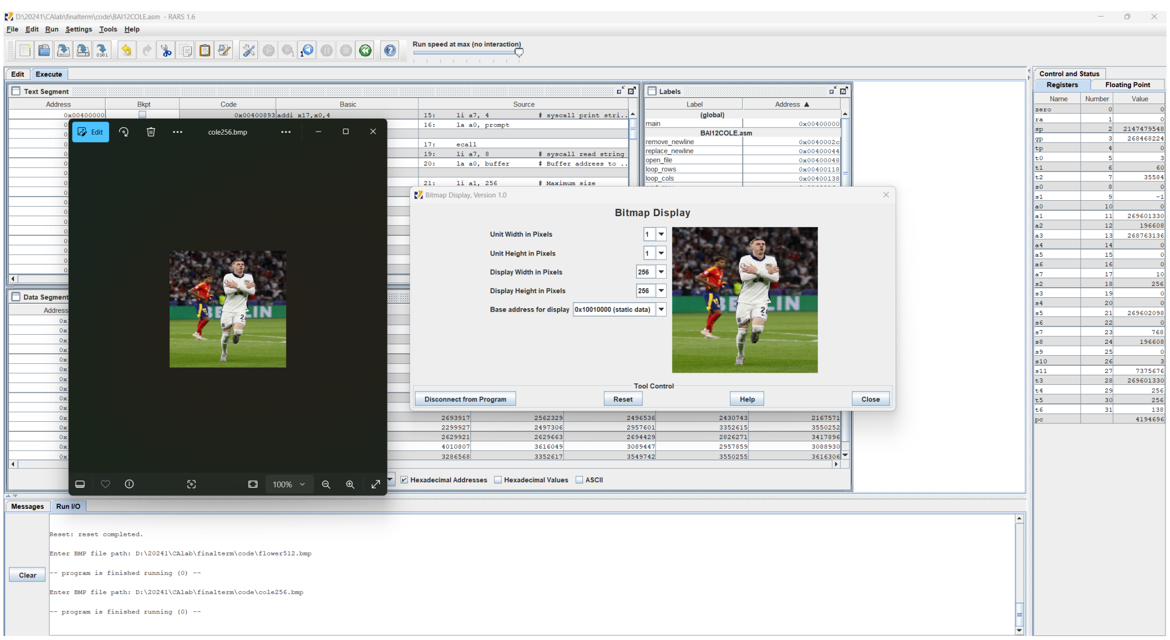

Figure 7: Displaying a BMP image with 256×256 resolution on the Bitmap Display

Case 3: Error Handling

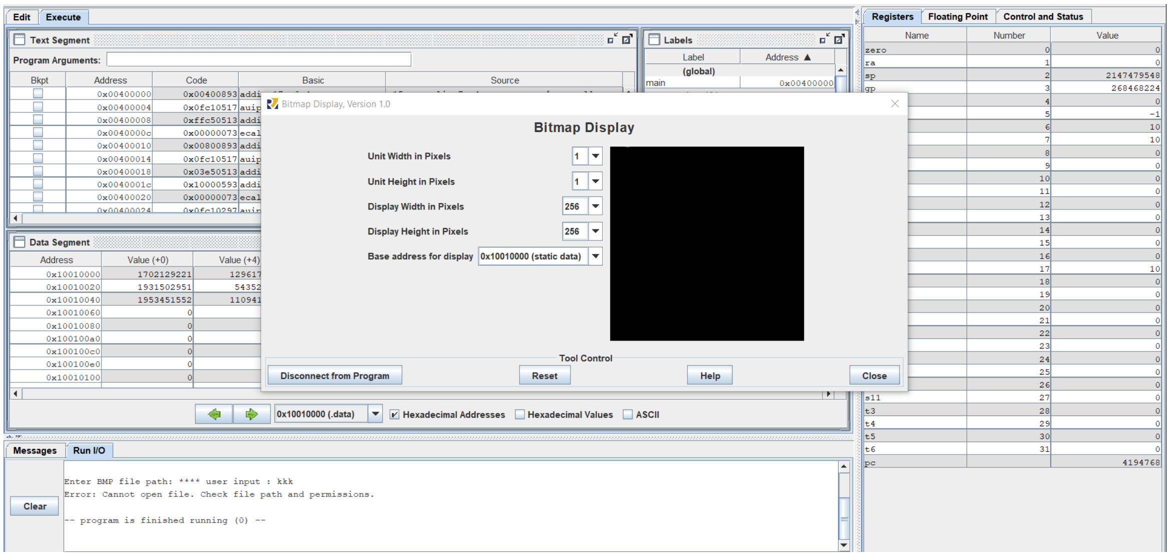

Figure 9: Error when the system can not open the file

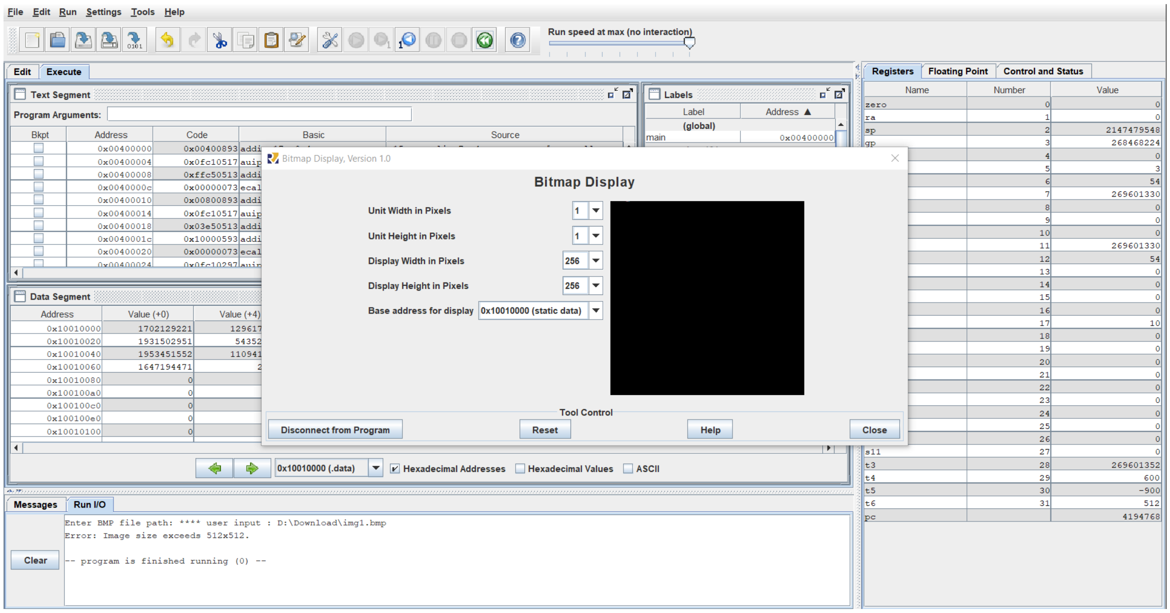

Figure 8: Error when the input image exceeds the limit size

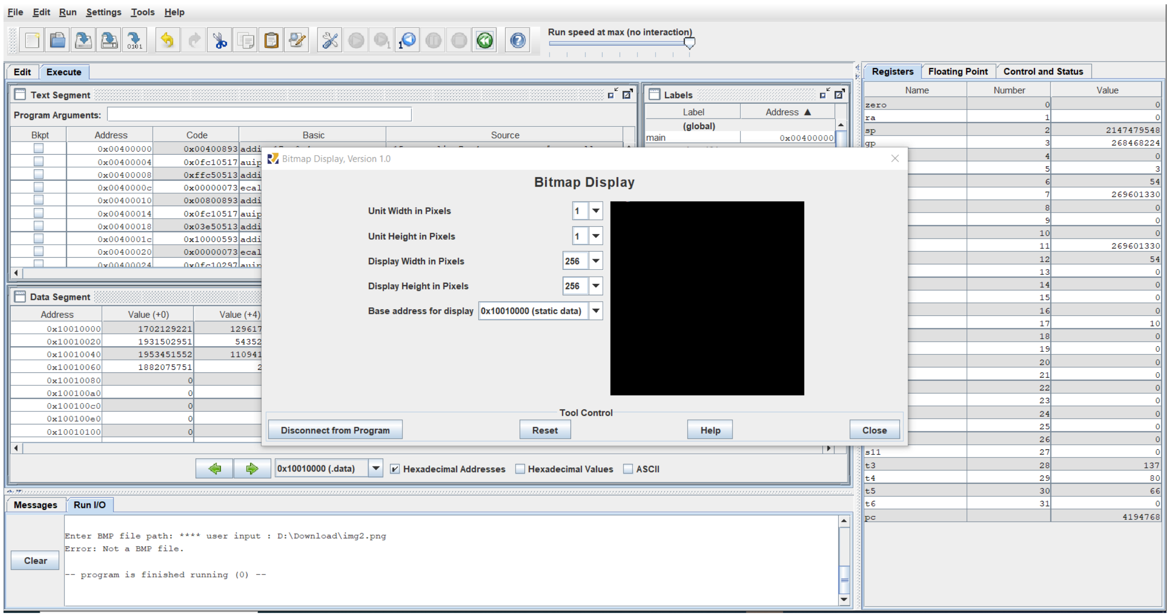

Figure 10: Error when the input image is not a BMP file

References

[1] Wikipedia contributors, “BMP file format — Wikipedia, The Free Encyclopedia.” https://en.wikipedia.org/wiki/BMP_file_format, 2024.

[2] “VGA - BMP File Structure.” http://www.ue.eti.pg.gda.pl/fpgalab/zadania.spartan3/zad_vga_struktura_pliku_bmp_en.html.

[3] “BMP File Format.” https://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/2003_w/misc/bmp_file_format/bmp_file_format.htm.