HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION
COMMUNICATION TECHNOLOGY

COMPUTER ARCHITECTURE LAB
FINAL REPORT

Group Members:
Ngô Anh Tú - 20226005
Phạm Trường Sang – 20225996

Exercise 3: Infix and postfix expressions

1. Requirements:

Create a program that can calculate an expression by evaluating the postfix expression.

Requirements:

1. Enter an infix expression from the console, for example: 9 + 2 + 8 * 6

2. Print it in the postfix representation, for example: 9 2 + 8 6 * +

3. Calculate and display the result to the console screen.

The operand must be an integer between 0 and 99.

Operators include addition, subtraction, multiplication, division, division with remainder and parenthesis.

2. Method:

1. The user can enter an infix expression into 'RUN I/O'.
2. The system will process the infix expression and convert it to postfix (display the postfix result), then evaluate the postfix expression to obtain the result (display the result).
3. The system will continue looping for the user to input new infix expressions. If the user inputs 'e', the loop will stop.

3. Algorithm:

Convert Infix to Postfix:

1. Scan the infix expression **from left to right**.

2. If the scanned character is an operand, put it in the postfix expression and print it out.

3. Otherwise, do the following

    - If the precedence of the current scanned operator is higher than the precedence of the operator on top of the stack, or if the stack is empty, or if the stack contains a '(', then push the current operator onto the stack.

    - Else, pop all operators from the stack that have precedence higher than or equal to that of the current operator. After that push the current operator onto the stack.

4. If the scanned character is a '**(**', push it to the stack.

5. If the scanned character is a '**)**', pop the stack and output it until a '**(**' is encountered, and discard both the parenthesis.

6. Repeat steps **2-5** until the infix expression is scanned.

7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.

8. Finally, print the postfix expression.

Evaluate Postfix to Result:

1. Scan the postfix from **left to right**

2. If the scanned character is an **operand**, put it into the stack.

3. If the scanned character is an **operator,** pop 2 operand from stack and calculate it, then push the new result into the stack.
4. Finally, print the result.

4. Source code: (read from left to right, then next page)

```
.data
input_message1: .asciz "Enter Infix Expression: "
input_message2: .asciz "Loop "
postfix_message: .asciz "Postfix Expression with respect to: "
message1: .asciz "Invalid input. Please enter your expression again!\n"
message2: .asciz "Operand's value is out of range. Please enter your expression again!\n"
message3: .asciz "Mismatched parenthesis. Please enter your expression again!\n"
message4: .asciz "Missing operand to perform the operator. Please check your expression and enter again\n"
block: .asciz "-------------------------"
output_mes1: .asciz "The result is: "
space: .asciz " "
buffer: .space 100
postfix_expression: .space 100
all: .align 2
operator_stack: .space 40
operand_stack: .word 40
.text
main:
        li t6, 0            # Store round
        beqz t6, input                      # First loop, no need to reset
reset: #reset all value to 0
        li a0, 0
        li a1, 0
        li a2, 0
        li a3, 0
        li s0, 0
        li s1, 0
        li s2, 0
        li s3, 0
        li t0, 0
        li t1, 0
        li t2, 0
        li t3, 0
        li t4, 0
input:
```

```
li a7, 4
    la a0, block #print ----------------------
    ecall
    li a7, 11 #print '\n'
    li a0, '\n'
    ecall
# Print the current loop
    li a7, 4
    la a0, input_message2
    ecall
    li a7, 1
    mv a0, t6
    ecall
    li a7, 11
    li a0, '\n'
    ecall
        li a7, 4
        la a0, input_message1
        ecall
        li a7, 8
        la a0, buffer
        li a1, 100
        ecall

        li s11, 'e'
        lb s10, 0(a0)
        beq s11,s10, exit
# ----------------------------------------------------------
# Infix to Postfix Procedure:
# 1.Scan the infix expression from left to right.
# 2.If the scanned character is an operand, put it in the postfix expression.
# 3.Otherwise, do the following:
#  -If the precedence of the current scanned operator is higher than the precedence of the operator on top of the stack,
#         or if the stack is empty, or if the stack contains a '(', then push the current operator onto the stack.
#  -Else, pop all operators from the stack that have precedence higher than or equal to that of the current operator.
#        After that push the current operator onto the stack.
# 4.If the scanned character is a ‘(‘, push it to the stack.
# 5.If the scanned character is a ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
# 6.Repeat steps 2-5 until the infix expression is scanned.
# 7.Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
# 8. Finally, print the postfix expression.
# ---------------------------------------------------------------

# ---------------------------------------------------------------
# Infix to Postfix:
# Register a1 : Postfix Expression
# Register a2 : Register to store Operators' value to check OR store some values in order to check operand
# Register s0 : Infix Expression
# Register s1 : Temporary store the address of Operator Stack  to check whether if it's empty or not
# Register s3 :        Operator Stack
# Register t0 : Store the current value of operand
# Register t1 : Current char
# Register t2 : Temporary hold the digit of operand to store into Postfix Expression
# Register t3 : Indicates the number of open brackets '(' is in the stack
# Register t4 : Store the operator in the top of stack
# ---------------------------------------------------------------
ifx2pfx:
# Message:
        li a7, 4
        la a0, postfix_message
        ecall
# Infix to Prefix
        la s0, buffer
        la a1, postfix_expression
        la s3, operator_stack
        li t0, -1
loop:
lb t1, 0(s0)     #take each char of the string
        li a2, 58
        bge t1, a2, invalid #if ascii > 58 -> out range
        beqz t1, bf_eval        # If encounter '\0' then move to Evaluation Phase
# Check Operand
        li a2, 48
        bge t1, a2, check_operand #if ascii > 48, <58 -> is operand
        li a2, -1
        beq t0, a2, find_operator     # If we don't encounter number then find  and the value of t0 is not equal -1 then find operator
        jal store_and_print             # Else encounter number (value of t0 is not equal -1) then store it to Postfix and print it.
                                        # before finding operator
find_operator:
        # Check Space
        li a2, 32
        beq t1, a2, next # If encounter space then move to next character
        # Addition
        li a2, 43
        beq t1, a2, handle_first_precedence
        # Subtraction
        li a2, 45
        beq t1, a2, handle_first_precedence
        #  Multiplication
        li a2, 42
        beq t1, a2, handle_second_precedence
        # Division
        li a2, 47
        beq t1, a2, handle_second_precedence
        # Division with remainder
        li a2, 37
        beq t1, a2, handle_second_precedence
        # Open round brackets
        li a2, 40
        beq t1, a2, push_bracket
        # Closed round brackets
```

```
li a2, 41
        beq t1, a2, handle_closed_bracket
        # Encounter newline then move to next char
        li a2, 10
        beq t1, a2, next
        # If cannot find any operator then invalid
        j invalid
next:
        addi s0, s0, 1 #next char
        j loop
invalid: #if invalid print error commet and run loop again
        li a7, 11
        li a0, '\n'
        ecall
        li a7, 4
        la a0, message1
        ecall
        li a7, 4
    la a0, block
    ecall
    li a7, 11
    li a0, '\n'
    ecall
    addi t6, t6, 1
        j reset

# Check whether if operand is out of range or not
check_operand:
        li a2, -1
        bgt t0, a2, new_oper            # If t0 = -1 then set t0 to 0
        li t0, 0
new_oper:
        li a2, 10
        mul t0, t0, a2 #if the number has 2 digit -> mul 10
        li a2, 100
        bge t0, a2, out_of_range # If value of t0 is greater than or equal 100 then print out the message.
        addi t1, t1, -48 # Transfer value of t1 to range from 0 to 9
```

```
        add t0, t0, t1  #t0 = current num (0-9)
        j next

out_of_range: #Print error commet and run loop again
        li a7, 11
        li a0, '\n'
        ecall
        li a7, 4
        la a0, message2
        ecall
        li a7, 4
    la a0, block
    ecall
    li a7, 11
    li a0, '\n'
    ecall
    addi t6, t6, 1
        j reset
store_and_print:
        # Print operand
        li a7, 1
        mv a0, t0
        ecall
        li a7, 4
        la a0, space
        ecall

        # Store operand to Postfix
        li a2, 10
        div t2, t0, a2 # t2 = t0 / 10
        beqz t2, one_digit
        addi t2, t2, 48 # Change to value in Ascii
        sb t2, 0(a1)
        addi a1, a1, 1
        j one_digit
one_digit:
        rem t2, t0, a2 # t2 = t0 % 10
        addi t2, t2, 48 # Change to value in Ascii
        sb t2, 0(a1)
        addi a1, a1, 1

        li a2, 32
        sb a2, 0(a1) # Store space
```

```
addi a1, a1, 1
	li t0, -1 # Reset value of t0 to -1 after storing and printing
	jr ra
# Handling operator
	# Handling first precedence
handle_first_precedence:
	la s1, operator_stack
	beq s3, s1, push_operator # If stack is empty, then push operator
	lw t4, 0(s3)
	li a2, 40
	beq t4, a2, push_operator # If the top of stack is '(', then push operator
#	bnez t3,  push_operator # If there is a '(', then push operator

	# Else pop all operators in stack and then push new operator
pop_operator_loop:
	beq s3, s1, push_operator  # If stack is empty now then push new operator
	lw t4, 0(s3)
	li a2, 40
	beq t4, a2, push_operator # If the top of stack is '(', then push operator
	jal store_and_print_operator
	addi s3, s3, -4
	j pop_operator_loop

	# Handling second precedence
handle_second_precedence:
	la s1, operator_stack
	beq s3, s1, push_operator # If stack is empty, then push operator
	lw t4, 0(s3)
	li a2, 40
	beq t4, a2, push_operator # If the top of stack is '(', then push operator
#	bnez t3,  push_operator # If there is a '(', then push operator

	# Else pop all operators in stack and then push new operator
pop_loop:
beq s3, s1, push_operator  # If stack is empty now then push new operator
	lw t4, 0(s3)
	li a2, 40
	beq t4, a2, push_operator # If the top of stack is '(', then push operator
	# If the operator in the top of stack has predence less than current operator then push operator to the stack
	li a2, 43
	beq t4, a2, push_operator # If the top of stack is '-', then push operator
	li a2, 45
	beq t4, a2, push_operator # If the top of stack is '+', then push operator

	# Else
	jal store_and_print_operator
	addi s3, s3, -4
	j pop_loop

push_operator:
	sw t1, 4(s3)
	addi s3, s3, 4
	j next
push_bracket:
	sw t1, 4(s3)
	addi t3, t3, 1 # Increase number of open bracket in stack by 1
	addi s3, s3, 4
	j next

# Handling closed bracket
handle_closed_bracket:
	beqz t3, syntax_error

pop_bracket_loop:
	lw t4, 0(s3)
	li a2, 40
	beq t4, a2, free_bracket # If encounter open bracket then free
	jal store_and_print_operator
	addi s3, s3, -4
	j pop_bracket_loop
free_bracket:
	addi s3, s3, -4
	addi t3, t3, -1 # Decerement number of open bracket in stack by 1
	j next


store_and_print_operator:
	# Print operator
	li a7, 11
	mv a0, t4
	ecall
	li a7, 4
	la a0, space
	ecall

	# Store operator to Postfix
	sb t4, 0(a1)
	addi a1, a1, 1
	li a2, 32
	sb a2, 0(a1) # Store space
	addi a1, a1, 1
	jr ra
# Before Evaluating
bf_eval:
# First check if there exist mismatch open bracket or not
	bnez t3, syntax_error
# Store and print operand if remain
	li a2, -1
	beq t0, a2, pop_all # If there is no operand that hasn't been stored and printed then move to store and print operators remain phase
	# Print operand
	li a7, 11
	mv a0, t0
	ecall
	li a7, 4
	la a0, space
	ecall
# Store operand to Postfix
li a2, 10
	div t2, t0, a2 # t2 = t0 / 10
beqz t2, one_digit_case
addi t2, t2, 48 # Change to value in Ascii
	sb t2, 0(a1)
	addi a1, a1, 1

one_digit_case:
	rem t2, t0, a2 # t2 = t0 % 10
	addi t2, t2, 48 # Change to value in Ascii
	sb t2, 0(a1)
	addi a1, a1, 1

	li a2, 32
	sb a2, 0(a1) # Store space
	addi a1, a1, 1
	li t0, -1 # Reset value of t0 to 0 after storing and printing
# Store and print operators remain
pop_all:
	la s1, operator_stack
	beq s3, s1, eval  # If stack is empty now then move to evaluation
	lw t4, 0(s3)
	li a2, 40
	beq t4, a2, pop_next
	jal store_and_print_operator
pop_next:
	addi s3, s3, -4
	j pop_all
```

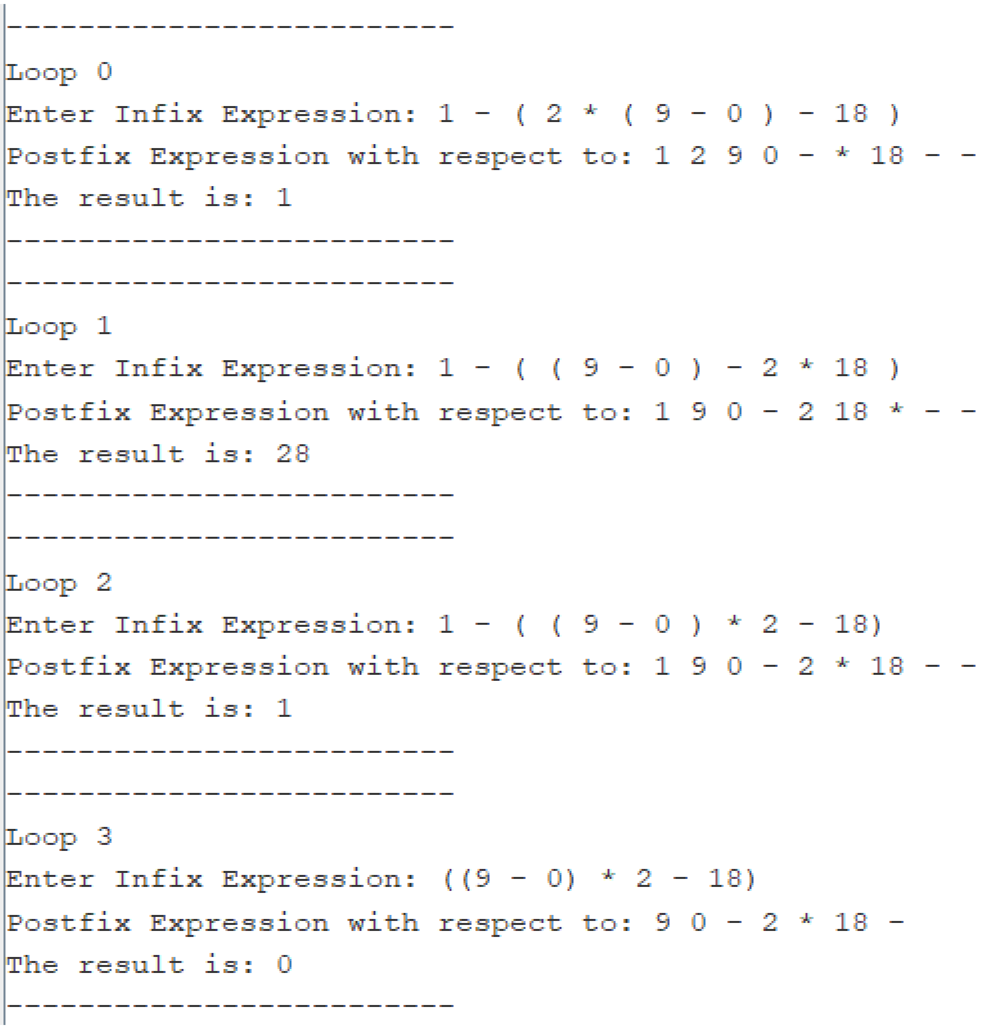
```
# Evaluation Phase
# Algorithm:
# Iterate the expression from left to right and keep
on storing the operands into a stack.
# Once an operator is received, pop the two
topmost elements and evaluate them and push
the result in the stack again.

# -------------------------------------------------------------
# Evaluation Procedure:
# Input: a1 - Postfix Expression
# Register using:
# Register a1 : Postfix Expression
# Register a2 : Register to store Operators' value to
check OR store some values in order to check
operand
# Register s0 : Iterator through Postfix Expression
# Register s1 : Temporary store the address of
Operator Stack  to check whether if it's empty or
not
# Register s2 :          Operand Stack
# Register s3 : Store the begining address of
Operand Stack
# Register t0 : Store the current value of operand
# Register t1 : Current char
# Register t2 : Holds value to perform operator
# Register t3 : Holds value to perform operator
# Register t4 : Holds the result of expression
# -------------------------------------------------------------
eval:
        la s0, postfix_expression
        la s2, operand_stack
        la s3, operand_stack
eval_loop:
        beq s0, a1, print_result      # Finish
        lb t1, 0(s0)
        # Check Operand
        li a2, 48
        bge t1, a2, handle_operand          # If
encounter number then ...
        li a2, -1
        bgt t0, a2, add_operand      # Else if value of
t0 is not equal -1 which means value of t1 now is
value of operand
                                                  # then add it to stack
                                                        # Else t0 = 0
# Addition
        li a2, 43
        beq t1, a2, addition
        # Subtraction
        li a2, 45
        beq t1, a2, subtraction
        #  Multiplication
        li a2, 42
        beq t1, a2, multiplication
        # Division
        li a2, 47
        beq t1, a2, division
        # Division with remainder
        li a2, 37
        beq t1, a2, div_rem
        # If encounter space then move to next
char
next_char:
        addi s0, s0, 1
        j eval_loop

# Handling operand when evaluating
handle_operand:
        li a2, -1
        bgt t0, a2, new_operand      # If t0 = -1 then
set t0 to 0
        li t0, 0
new_operand:
        li a2, 10
        mul t0, t0, a2
        addi t1, t1, -48 # Transfer value of t1 to
range from 0 to 9
        add t0, t0, t1
        j next_char
add_operand:
        sw t0, 4(s2)
        addi s2, s2, 4
        li t0, -1          # Reset value of t0
        j next_char
```

```
# Handling operator when evaluating
addition:
        lw t2, 0(s2)
        addi s2, s2, -4
        beq s2, s3, miss_operand_error    # If there is only one operand in the stack the issue an error
        lw t3, 0(s2)
        addi s2, s2, -4
        add t2, t2, t3 # Perform additon
        sw t2, 4(s2) # Store result back into stack
        addi s2, s2, 4
        j next_char
subtraction:
        lw t2, 0(s2)
        addi s2, s2, -4
        beq s2, s3, miss_operand_error              # If there is only one operand in the stack the issue an error
        lw t3, 0(s2)
        addi s2, s2, -4
        sub t2, t3, t2 # Perform subtraction
        sw t2, 4(s2) # Store result back into stack
        addi s2, s2, 4
        j next_char
multiplication:
        lw t2, 0(s2)
        addi s2, s2, -4
        beq s2, s3, miss_operand_error              # If there is only one operand in the stack the issue an error
        lw t3, 0(s2)
        addi s2, s2, -4
        mul t2, t2, t3 # Perform multiplication
        sw t2, 4(s2) # Store result back into stac
        addi s2, s2, 4
        j next_char
division:
        lw t2, 0(s2)
        addi s2, s2, -4
        beq s2, s3, miss_operand_error              # If there is only one operand in the stack the issue an error
        lw t3, 0(s2)
        addi s2, s2, -4
div t2, t3, t2 # Perform division
        sw t2, 4(s2) # Store result back into stack
        addi s2, s2, 4
        j next_char
div_rem:
        lw t2, 0(s2)
        addi s2, s2, -4
        beq s2, s3, miss_operand_error              # If there is only one operand in the stack the issue an error
        lw t3, 0(s2)
        addi s2, s2, -4
        rem t2, t3, t2 # Perform division with remainder:
        sw t2, 4(s2) # Store result back into stack
        addi s2, s2, 4
        j next_char

# Print result
print_result:
        lw t4, 0(s2) # Take result
        li a7, 11
        li a0, '\n'
        ecall
        li a7, 4
        la a0, output_mes1
        ecall
        li a7, 1
        mv a0, t4
        ecall
        li a7, 11
    li a0, '\n'
    ecall
    li a7, 4
    la a0, block
    ecall
    li a7, 11
    li a0, '\n'
    ecall
    addi t6, t6, 1                  # Increment round
        j reset

syntax_error:
        li a7, 11
    li a0, '\n'
    ecall
        li a7, 4
        la a0, message3
        ecall
    li a7, 4
    la a0, block
    ecall
    li a7, 11
    li a0, '\n'
    ecall
    addi t6, t6, 1
        j reset

miss_operand_error:
        li a7, 11
    li a0, '\n'
    ecall
        li a7, 4
        la a0, message4
        ecall
    li a7, 4
    la a0, block
    ecall
    li a7, 11
    li a0, '\n'
    ecall
    addi t6, t6, 1
        j reset
exit:
        li a7, 10
        ecall
```

- All explanations have been written in comments within the source code.

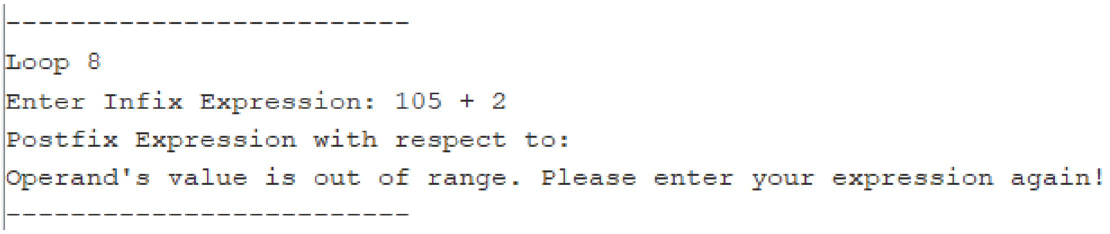
5. Simulation result:

Normal case:



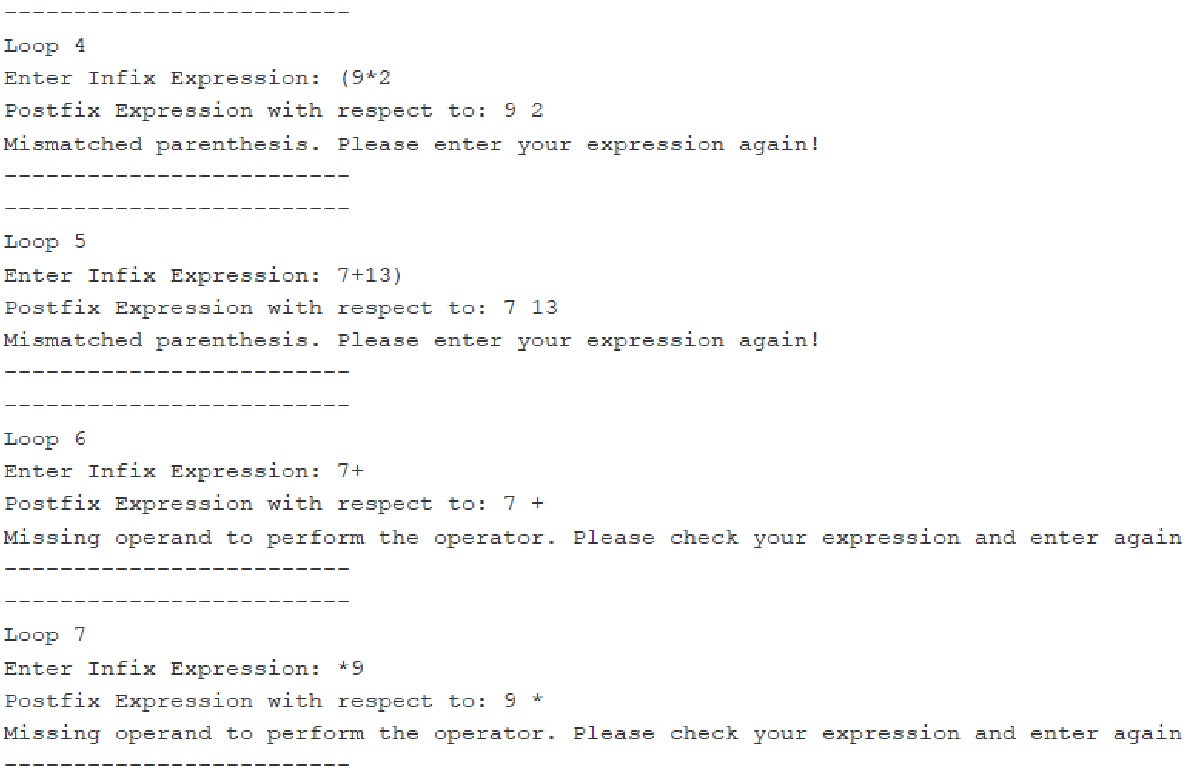
```
-------------------------

Loop 0
Enter Infix Expression: 1 - ( 2 * ( 9 - 0 ) - 18 )
Postfix Expression with respect to: 1 2 9 0 - * 18 - -
The result is: 1
-------------------------
-------------------------
Loop 1
Enter Infix Expression: 1 - ( ( 9 - 0 ) - 2 * 18 )
Postfix Expression with respect to: 1 9 0 - 2 18 * - -
The result is: 28
-------------------------
-------------------------
Loop 2
Enter Infix Expression: 1 - ( ( 9 - 0 ) * 2 - 18)
Postfix Expression with respect to: 1 9 0 - 2 * 18 - -
The result is: 1
-------------------------
-------------------------
Loop 3
Enter Infix Expression: ((9 - 0) * 2 - 18)
Postfix Expression with respect to: 9 0 - 2 * 18 -
The result is: 0
-------------------------
```

Out of range:



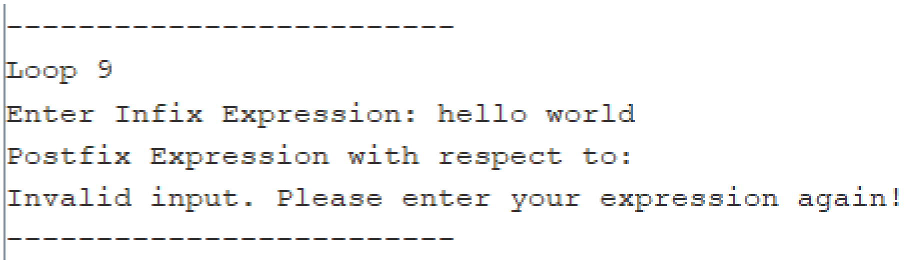
```
-------------------------
Loop 8
Enter Infix Expression: 105 + 2
Postfix Expression with respect to:
Operand's value is out of range. Please enter your expression again!
-------------------------
```

Missing operand:



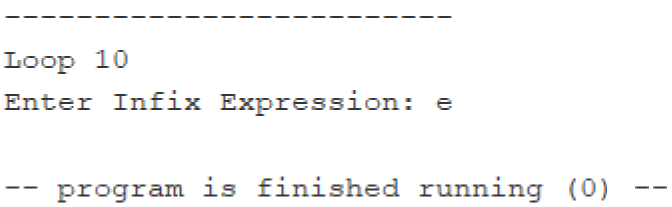
```
-------------------------
Loop 4
Enter Infix Expression: (9*2
Postfix Expression with respect to: 9 2
Mismatched parenthesis. Please enter your expression again!
-------------------------
-------------------------
Loop 5
Enter Infix Expression: 7+13)
Postfix Expression with respect to: 7 13
Mismatched parenthesis. Please enter your expression again!
-------------------------
-------------------------
Loop 6
Enter Infix Expression: 7+
Postfix Expression with respect to: 7 +
Missing operand to perform the operator. Please check your expression and enter again
-------------------------
-------------------------
Loop 7
Enter Infix Expression: *9
Postfix Expression with respect to: 9 *
Missing operand to perform the operator. Please check your expression and enter again
-------------------------
```

Invalid input:



```
-------------------------
Loop 9
Enter Infix Expression: hello world
Postfix Expression with respect to:
Invalid input. Please enter your expression again!
-------------------------
```

Exit the program:



```
-------------------------
Loop 10
Enter Infix Expression: e

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

Exercise 15: Numbers memory game

1. Requirements:

+ Research about the system call to generate a random number.
+ Show 4 random numbers in the DISPLAY window. These numbers have at least 3 digits.
+ After counting down from 5 seconds, clear these numbers.
+ The user will enter these numbers into the KEYBOARD window, separated by spaces, and ending with the Enter key.
+ The program will determine if the numbers entered are correct. If correct, the game will move to the next round and end the game if not. The numbers entered do not have to be in the same order with the original numbers.

2. Method:

1. Generate 4 different random numbers displayed on the "Keyboard and Display MMIO," along with a countdown timer showing 5 seconds decreasing gradually.
2. After 5 seconds, the numbers disappear and are replaced by 16 other numbers (including the 4 original numbers and 12 other random numbers).
3. After the user enters the numbers and presses Enter, the system will check whether the entered numbers match the initial 4 numbers:
   - If correct, the system will display "Correct" and repeat the process.
   - If incorrect, the system will display "Wrong" and stop.

3. Algorithm:

1. Generate and display 4 random numbers and store them into **numbers**.
2. Countdown timer (5 seconds).
3. Generate and display 16 random numbers (including the 4 initial numbers).
4. Store the input numbers into **user_answer** (string)**,** then extracting to number into **answer_numbers_stack** and check:
    - If each number exists in **numebers**, set that number to **-1** (in **numbers**) and continue checking the next number in **answer_numbers_stack**. If all 4 numbers are correct, print **"Correct"** and repeat the program.
    - If a number does not exist in **numbers**, print **"Wrong"** and stop the program.

Some function:

Print_number function(): Description: Prints the number in a0 to the display. (a0 in here is all the numbers need to print).
check_dup function(): Description: Checking whether the number generated duplicated or not.
check_pos_dup function(): Description: Checking whether the positon of number generated in the matrix duplicated or not. (the position here is the position will display the number in 16 noise number).
write_char function(): Description: Writes the character in t5 to the display. (t5 in here is all the characters need to print).
Moving cursor function(): Description: Move cursor to position (x, y). (Move to the position need to display).
clear_display function(): Description: Clears the display window.
Delay_one_second function(): Description: delay the time counter 1s.
Extracting answer function(): Store the whole string of input (user answer) to user_answer.

Extracting number function(): Etracting to each number and put it into answer_numbers_stack.
Checking number function(): Checking each number against **numbers.**

Handle exception:

Handle out of range issues.

Handle invalid issues.

Show out too many inputs issue message. (more than 4 numbers)

Show out the less input issue message. (less than 4 numbers)

4. Source code:

```
.eqv RECEIVER_CONTROLLER   0xFFFF0000      # ASCII code from keyboard, 1 byte
.eqv RECEIVER_DATA     0xFFFF0004    # =1 if has a new keycode ?
                                     # Auto clear after lw
.eqv TRANSMITTER_CONTROLLER   0xFFFF0008   # ASCII code to show, 1 byte
.eqv TRANSMITTER_DATA    0xFFFF000C # =1 if the display has already to do
                                     # Auto clear after sw
.eqv SPACE 32           # ASCII for space character
.eqv NEWLINE 10         # ASCII for newline character
.data
numbers:    .space 16        # Space to store the 4 random numbers (4 bytes each)
position:   .space 16              # Space to store the position of the random numbers in the matrix.
user_answer: .space 64        # Space to store user's answer
answer_numbers_stack: .space 40# Space to store the numbers extracted from answer
countdown_msg: .asciz "Time remain: "
```

```
wrong_msg:   .asciz "Wrong! Game Over."
error_msg1: .asciz "Wrong input syntax. Please enter your answers separated by spaces, and ending with the Enter key."
error_msg2: .asciz "Your input is out of range. Please enter your answers between 1 and 999."
error_msg3: .asciz "Enter 4 numbers only!"
```

```
correct_msg: .asciz "Correct! Moving to next round."
```

```
.text
# ------------------------------------------------------------
# Main game loop
# Register used:
# Register a1: Store RECEIVER_CONTROLLER address
# Register a2: Store RECEIVER_DATA address
# Register s1: Store TRANSMITTER_CONTROLLER address
# Register s2: Store TRANSMITTER_DATA address
# Register t1: Address of Array to store numbers to remember
# Register t6: Store round counter
#-------------------------------------------------------
   li t6, 0
main:
   li  a1, RECEIVER_CONTROLLER
li  a2, RECEIVER_DATA
   li  s1, TRANSMITTER_CONTROLLER
   li  s2, TRANSMITTER_DATA

   jal gen_and_disp_nums
   jal countdown_phase
   jal clear_display
   jal gen_sixteen_nums
   jal receive_and_check
   beqz t6, exit                    # If detecting t6 equal 0 then end game
   j main                           # Else, next round
```

```
# Step 1: Generate and display random numbers
# ------------------------------------------------------------
# Generate and display random numbers function
# Register used:
# Register a0: Store generated number
# Register a1: Store RECEIVER_CONTROLLER address
# Register a2: Store RECEIVER_DATA address
# Register s1: Store TRANSMITTER_CONTROLLER address
# Register s2: Store TRANSMITTER_DATA address
# Register s3: Address of Array to store position numbers to remember in the matrix
# Register s6: Store the x position of the cursor
# Register s7: Store the y position of the cursor
# Register s8: Indicate that the generated number existed or not based on check_pos_dup function
# Register t1: Address of Array to store numbers to remember
# Register t2: Store the number needs to be generated
# Register t6: Store round counter
#-------------------------------------------------------
gen_and_disp_nums:
# Store return address
   addi sp, sp, -16
   sw ra, 0(sp)
   sw s6, 4(sp)
   sw s7, 8(sp)
   sw a1, 12(sp)

   beqz t6, four_number_generation    # First loop, no need to clear
   jal clear_display       # Clear the display

four_number_generation:
   li s6, 0                          # x = 0
   li s7, 0                          # y = 0
   jal move_cur_pos         # Set the start position at the point (0, 0)

   la t1, numbers          # Address to store numbers
   li t2, 4             # Generate 4 numbers
   la s3, position              # Address to store the position of 4 corrected numbers in the matrix
```

```
four_number_generation_loop:
   li a1, 999            # Generate random numbers in range (0-999)
   li a7, 42
   ecall
   jal check_dup                    # Check the duplication with the new generated number
   bnez s8, four_number_generation_loop # If s8 = 1 then generate again, else store the new one
   sw a0, 0(t1)           # Store random number in memory
   addi t1, t1, 4
   jal print_number        # Display the number
   li a0, SPACE            # Print space
   jal write_char
# Generate the position of new generated number in the matrix
pos_rand:
   li a1, 15
   li a7, 42
   ecall
   jal check_pos_dup
   bnez s8, pos_rand        # Check the duplication with the positon of new generated number in the matrix
   sw a0, 0(s3)
   addi s3, s3, 4
   addi t2, t2, -1          # Decrement the total numbers remained to generate
   bnez t2, four_number_generation_loop

# Load return address

   lw a1, 12(sp)
   lw s7, 8(sp)
   lw s6, 4(sp)
   lw ra, 0(sp)
   addi sp, sp, 16
   jr ra                             # Finish generating numbers step
```

```
# Description: Checking whether the number generated duplicated or not
# ------------------------------------------------------------
# check_dup function:
# Register used:
# Register a0: Store number to be checked (No need to save register a0 to stack because we don't modify it)
# Register s0: Store begining of the Arrray
# Register t1: Address of Array to store numbers to remember
# Register t2: The number stored in the Array
# Return: register s8 indicating that there is a duplication (s8 = 1) or not (s8 = 0)
#-------------------------------------------------------
check_dup:
# Load return address
   addi sp, sp, -16
   sw ra, 0(sp)
   sw s0, 4(sp)
   sw t1, 8(sp)
   sw t2, 12(sp)
   li s8, 0                 # Initialize s8 = 0
   la s0, numbers
   beq s0, t1, out_dup_loop  # If there aren't any numbers stored then store it
check_dup_loop:
   addi t1, t1, -4
   lw t2, 0(t1)                        # Load each number stored in the Array
   beq t2, a0,   exist_dup                   # If duplicate then set s8 = 1
   beq s0, t1, out_dup_loop  # If there is no duplication then s8 remains 0
   j check_dup_loop
exist_dup:
   li s8, 1
out_dup_loop:
   lw t2, 12(sp)
   lw t1, 8(sp)
   lw s0, 4(sp)
   lw ra, 0(sp)
   addi sp, sp, 16
```

```
# Description: Checking whether the positon of number generated in the matrix duplicated or not
# ------------------------------------------------------------
# check_pos_dup function:
# Register used:
# Register a0: Store number to be checked (No need to save register a0 to stack because we don't modify it)
# Register s0: Store begining of the Arrray of position
# Register s3: Address of Array to store the position of numbers to remember
# Register t2: The position of the number stored in the Array
# Return: register s8 indicating that there is a duplication (s8 = 1) or not (s8 = 0)
#------------------------------------------------------
check_pos_dup:
# Load return address
   addi sp, sp, -16
   sw ra, 0(sp)
   sw s0, 4(sp)
   sw t2, 8(sp)
   sw s3, 12(sp)
   li s8, 0                    # Initialize s8 = 0
   la s0, position
   beq s0, s3, out_pos_dup_loop      # If there aren't any numbers stored then store it
check_pos_dup_loop:
   addi s3, s3, -4
   lw t2, 0(s3)                          # Load positon of each number stored in the Array
   beq t2, a0,   exist_pos_dup                         # If duplicate then set s8 = 1
   beq s0, s3, out_pos_dup_loop      # If there is no duplication then s8 remains 0
   j check_pos_dup_loop
exist_pos_dup:
   li s8, 1
out_pos_dup_loop:
   lw s3, 12(sp)
   lw t2, 8(sp)
   lw s0, 4(sp)
   lw ra, 0(sp)
```

```
   jr ra
   jr ra
# Step 2: Countdown timer (fix 5 seconds)
# ------------------------------------------------------------
# Print countdown function
# Register used:
# Register a0: Store the countdown message
# Register a1: Store RECEIVER_CONTROLLER address
# Register a2: Store RECEIVER_DATA address
# Register s1: Store TRANSMITTER_CONTROLLER address
# Register s2: Store TRANSMITTER_DATA address
# Register t4: Store the number seconds to countdown
# Register t5: Store each character in countdown message
# Register t6: Store round counter
# Register s6: Store the x position of the cursor
# Register s7: Store the y position of the cursor
#------------------------------------------------------

countdown_phase:
# Store return address
   addi sp, sp, -12
   sw ra, 0(sp)
   sw s6, 4(sp)
   sw s7, 8(sp)

   li t4, 5                    # Set up the numbers of seconds to remember fixed at 5 seconds
   li s6, 0                    # x = 0
   li s7, 1                    # y = 1

   jal move_cur_pos
# Reset the coordination after modifying in the move_cur_pos function
   li s6, 0                    # x = 0
   li s7, 1                    # y = 1
   la a0, countdown_msg
print_cd_msg:
   lb t5, 0(a0)
   beqz t5, countdown_loop
   jal write_char
   addi a0, a0, 1
```

```
  addi sp, sp, 16
```

```
  addi s6, s6, 1          # Move cursor to the next position to write character
  j print_cd_msg
countdown_loop:
  mv a0, t4
  jal print_number        # Display countdown number
  jal delay_one_second
  jal move_cur_pos
  addi t4, t4, -1
  bgez t4, countdown_loop

# Load return address
  lw s7, 8(sp)
  lw s6, 4(sp)
  lw ra, 0(sp)
  addi sp, sp, 12
  jr ra

# Description: Prints the number in a0 to the display
# -----------------------------------------------------------
# Print_number function:
# Register used:
# Register a0: Store number to be printed out
# Register a1: Store RECEIVER_CONTROLLER address
# Register a2: Store RECEIVER_DATA address
# Register s1: Store TRANSMITTER_CONTROLLER address
# Register s2: Store TRANSMITTER_DATA address
# Register t3: Copy the value of generated number to print
# Register t4: Holds digit 10
# Register t5: Store the character to write out
#-----------------------------------------------------

print_number:
# Store return address
  addi sp, sp, -16
  sw ra, 0(sp)
  sw s6, 4(sp)
  sw s7, 8(sp)
  sw t4, 12(sp)
```

```
mv t3, a0            # Copy the number to t3
  li t4, 100
  div t5, t3, t4          # Extract hundred digit
  beqz t5, just_two_digit           # If t5 = 0 move to print 2-digit number
  addi t5, t5, 48
  jal write_char                # Else write
two_digit:
  rem t3, t3, t4                # t3 = t3 % 100
  li t4, 10
  div t5, t3, t4          # Extract tenth digit
  addi t5, t5, 48
  jal write_char
  j one_digit

just_two_digit:
  rem t3, t3, t4                # t3 = t3 % 100
  li t4, 10
  div t5, t3, t4          # Extract tenth digit
  beqz t5, one_digit      # If t5 = 0 move to print 1-digit number
  addi t5, t5, 48
  jal write_char

one_digit:
  rem t3, t3, t4                # t3 = t3 % 10
  mv t5, t3
  addi t5, t5, 48
  jal write_char

print_done:
  li t5, SPACE
  jal write_char                # Print
  addi s6, s6, 1

# Load return address
  lw t4, 12(sp)
  lw s7, 8(sp)
  lw s6, 4(sp)
  lw ra, 0(sp)
  addi sp, sp, 16
  jr ra
```

```
# Description: Writes the character in t5 to the display
# -----------------------------------------------------------
# write_char function:
# Register used:
# Register s1: Store TRANSMITTER_CONTROLLER address
# Register s2: Store TRANSMITTER_DATA address
# Register t0: Store the status of Ready Bit
# Register t5: Store the character to write out
#----------------------------------------------------
write_char:
# Store return address
   addi sp, sp, -12
   sw ra, 0(sp)
   sw s6, 4(sp)
   sw s7, 8(sp)

   li s1, TRANSMITTER_CONTROLLER
   lw t0, 0(s1)          # Load the control register
   beqz t0, write_char      # Wait until Ready bit is 1
   li s2, TRANSMITTER_DATA          # Transmitter Data Register address
   sw t5, 0(s2)          # Write the character to the display
# Load return address
   lw s7, 8(sp)
   lw s6, 4(sp)
   lw ra, 0(sp)
   addi sp, sp, 12
   jr ra
# Description: Move cursor to position (x, y)
# -----------------------------------------------------------
# Moving cursor function:
# Register used:
# Register a5: Store the ASCII form feed
# Register s6: Store the x position of the cursor
# Register s7: Store the y position of the cursor
# Register s1: Store TRANSMITTER_CONTROLLER address
# Register s2: Store TRANSMITTER_DATA address
# Register t0: Store the status of Ready Bit
# Register t5: Store the character to write out
```

```
move_cur_pos:
# Store return address
   addi sp, sp, -12
   sw ra, 0(sp)
   sw s6, 4(sp)
   sw s7, 8(sp)

   li a5, 7
   slli s6, s6, 20
   slli s7, s7, 8
# Modify the value to load to the right position
   or a5, a5, s6
   or a5, a5, s7
cur_wait:
   lw t0, 0(s1) # Wait until Ready bit is 1
   beq t0, zero, cur_wait
   sw a5, 0(s2)  # write the form to the ASCII code(Bell or FF)

# Load return address
   lw s7, 8(sp)
   lw s6, 4(sp)
   lw ra, 0(sp)
   addi sp, sp, 12

   jr ra

# Description: Clears the display window
# -----------------------------------------------------------
# clear_display function:
# Register used:
# Register a5: Store the ASCII form feed
# Register s1: Store TRANSMITTER_CONTROLLER address
# Register s2: Store TRANSMITTER_DATA address
# Register t0: Store the status of Ready Bit
#----------------------------------------------------
clear_display:
   addi sp, sp, -4
   sw ra, 0(sp)

   li a5, 12        # ASCII form feed
```

```
#-------------------------------------------------------
wait_clear:
   li s1, TRANSMITTER_CONTROLLER
   lw t0, 0(s1)     # Wait until Ready bit is 1
   beq t0, zero, wait_clear
   li s2, TRANSMITTER_DATA
   sw a5, 0(s2)  # write the form to the ASCII code(Bell or FF)

   lw ra, 0(sp)
   addi sp, sp, 4

   jr ra


# -----------------------------------------------------------
# Delay_one_second function:
#------------------------------------------------------------
delay_one_second:
   addi sp, sp, -16
   sw ra, 0(sp)
   sw a0, 4(sp)
   sw s6, 8(sp)
   sw s7, 12(sp)

   li a7, 32
   li a0, 1000
   ecall

   # Load return address
   lw s7, 12(sp)
   lw s6, 8(sp)
   lw a0, 4(sp)
   lw ra, 0(sp)
   addi sp, sp, 16
   jr ra
```

```
# Step 3: Generate 16 nunmbers
# -----------------------------------------------------------
# Generate and display noise numbers function
# Register used:
# Register a0: Store generated number
# Register a1: Store RECEIVER_CONTROLLER address
# Register a2: Store RECEIVER_DATA address
# Register s1: Store TRANSMITTER_CONTROLLER address
# Register s2: Store TRANSMITTER_DATA address
# Register t1: Store the rows needs to be generated
# Register t2: Store the columns needs to be generated
# Register t6: Store round counter
# Register s6: Store the x position of the cursor
# Register s7: Store the y position of the cursor
# Register s8: Store the current position in the matrix (convert to 0-15) s8 = 16 - t1 * 4 - t2
# Register s9: Indicate whether the current position is the position of generated number in the matrix or not
#-----------------------------------------------------------

gen_sixteen_nums:
sixteen_number_generation:
# Store return address
   addi sp, sp, -8
   sw ra, 0(sp)
   sw a1, 4(sp)

   li s6, 0                              # x = 0
   li s7, 0                              # y = 0
   jal move_cur_pos         # Set the start position at the point (0, 0)

   li t1, 4              # Initialize 4 rows
sixteen_number_generation_loop1:
   beqz t1,  end_gen_loop
   li t2, 4
sixteen_number_generation_loop2:
   jal check_stored_pos                    # Check whether current position is the position of stored number or not
   bnez s9, print_stored_number
```

```
  li a1, 999            # If the current position is not
```

```
  li a7, 42             # Then generate random numbers in range (0-999)
  ecall
print_stored_number:
  jal print_number        # Display the number
  li a0, SPACE            # Print space
  jal write_char
  addi t2, t2, -1         # Decrement the total columns remained to generate
  addi s6, s6, 1
  bnez t2, sixteen_number_generation_loop2
  addi t1, t1, -1                    # Decrement the total rows remained to generate
  li s6, 0                           # Reset s6 to 0
  addi s7, s7, 1                     # And move to next rows
  jal move_cur_pos
  j sixteen_number_generation_loop1
end_gen_loop:
# Load return address
  lw a1, 4(sp)
  lw ra, 0(sp)
  addi sp, sp, 8
  jr ra                         # Finish number generation step

# Description: Checking whether the current positon is the position of number generated in the matrix or not
# ------------------------------------------------------------
# check_stored_pos function:
# Register used:
# Register s0: Store begining of the Arrray of position
# Register s1: Store begining of the Arrray of stored number
# Register s3: Address of Array to store the position of numbers to remember
# Register s6: Store the x position of the cursor
# Register s7: Store the y position of the cursor
# Register s8: Store the current position in the matrix (convert to 0-15) s8 = s7 * 4 + s6
# Register t1: Address of Array to store numbers to remember
```

```
# Register t3: The numbers of number need to check
# Return: register a0 : Stored number to print out if s9 = 1
#         register s9 : Indicate whether the current position is the position of generated number in the matrix or not
#-----------------------------------------------------
check_stored_pos:
# Store return address
  addi sp, sp, -36
  sw ra, 0(sp)
  sw s0, 4(sp)
  sw s1, 8(sp)
  sw s3, 12(sp)
  sw s8, 16(sp)
  sw t1, 20(sp)
  sw t2, 24(sp)
  sw t3, 28(sp)
  sw a1, 32(sp)


  slli s8, s7, 2 # s8 = 4 * s7
  add s8, s8, s6         # s8 = 4 * s7 + s6
  la s0, position
  la a1, numbers
  li s9, 0
  addi a1, a1, 16
check_stored_pos_loop:
  addi s3, s3, -4
  addi a1, a1, -4
  lw t2, 0(s3)                        # Load positon of each number stored in the Array
  beq t2, s8,   exist_stored_number_pos               # If match then set s9 = 1
  beq s0, s3, out_stored_pos_loop # If there is no duplication then s8 remains 0
  j check_stored_pos_loop
exist_stored_number_pos:
  li s9, 1
  lw a0, 0(a1)
```

```
# Register t2: The position of the number stored in the Array
```

```
out_stored_pos_loop:
# Load return address
   lw a1, 32(sp)
   lw t3, 28(sp)
   lw t2, 24(sp)
   lw t1, 20(sp)
   lw s8, 16(sp)
   lw s3, 12(sp)
   lw s1, 8(sp)
   lw s0, 4(sp)
   lw ra, 0(sp)
   addi sp, sp, 36
   jr ra

 # Step 4: Receive and check the answer
# -------------------------------------------------------------
# Receiving answer function
# Register used:
# Register a0: Store the user's answer
# Register a1: Store RECEIVER_CONTROLLER address
# Register a2: Store RECEIVER_DATA address
# Register s1: Store TRANSMITTER_CONTROLLER address
# Register s2: Store TRANSMITTER_DATA address
# Register s0: Store the ASCII code of NEWLINE and BACKSPACE
# Register t0: Store the status of Ready Bit
# Register t1: Store the character read from Receiver Data
# Register t2: Store the begining Address of user_answer
#-------------------------------------------------------------
receive_and_check:
   addi sp, sp, -8
   sw ra, 0(sp)
   sw t2, 4(sp)

   la a0, user_answer
   la t2, user_answer
wait_ans:
   lw t0, 0(a1)
```

```
receive_ans:
   lw t1, 0(a2)
   li s0, BACKSPACE
   beq s0, t1, handle_backspace                  # If encounter Backspace then...
   li s0, NEWLINE
   sb t1, 0(a0)             # Store to the string
   addi a0, a0, 1
   beq t1, s0, extract_ans       # If character read is NEWLINE then move to checking function
   j wait_ans                          # Else continue to receive the answer

handle_backspace:
   beq a0, t2, wait_ans                      # If user answer is empty and encounter backspace then do nothing
   addi a0, a0, -1                     # Else, move back
   j  wait_ans
back_exit:
   mv t6, zero
   lw t2, 4(sp)
   lw ra, 0(sp)
   addi sp, sp, 8
   jr ra
back_next:
   li a7, 32
   li a0, 1000
   ecall
   addi t6, t6, 1
   lw ra, 0(sp)
   addi sp, sp, 4
   jr ra
# -------------------------------------------------------------
# Extracting answer function
# Register used:
# Register a0: Store the user's answer
# Register a1: Store RECEIVER_CONTROLLER address
# Register a2: Store RECEIVER_DATA address
# Register s1: Store TRANSMITTER_CONTROLLER address
# Register s2: Store TRANSMITTER_DATA address
# Register t0: Store the number taken from answer
```

```
    beq t0, zero, wait_ans
```

```
# Register t3: Store the character read from answer
# Register t4: Holds some extra values to check
# Register s11: Indicates the number of answers can be input
#-------------------------------------------------------------
extract_ans:
    la a0, user_answer
    la t1, answer_numbers_stack
    li t0, -1                    # Initialize the number extracted from answer to -1
    li s11, -4                   # Can just input 4 numbers
extract_loop:
    lb t3, 0(a0)
    li t4, 57
    bgt t3, t4, invalid          # Invalid
    li t4, NEWLINE
    bgtz s11, more_input_issue
    beq t3, t4, check_numbers          # If reach the newline then move to checking phase
    li t4, 48
    bge t3, t4, extract_number # If encouter the digit then move to extract number function
    li t4, SPACE
    bne t3, t4, invalid    # If not digit nor Newline or Space then it's invalid
    li t4, -1
    beq t0, t4, extract_next      # If encounter Space and extract a number then store it
    sw t0, 0(t1)             # Else move to next char
    addi s11, s11, 1             # Decrement the numbers remain need to be input
    li t0, -1                    # Reset t0 to -1
    addi t1, t1, 4
extract_next:
    addi a0, a0, 1
    j extract_loop

# -------------------------------------------------------------
# Extracting number function
# Register used:
# Register t0: Store the number taken from answer
```

```
# Register t1: Stack to store numbers taken from answer for furthur checking
```

```
# Register t4: Holds some extra values (changing frequently)
#-------------------------------------------------------------
extract_number:
    li t4, -1
    bgt t0, t4, next_nb            # If t0 = -1 then we found a new number then set t0 = 0
    mv t0, zero
next_nb:
    li t4, 10
    mul t0, t0, t4
    li t4, 1000
    bge t0, t4, out_of_range              # If exceed 1000 then signal an issue
    addi t3, t3, -48
    add t0, t0, t3
    j extract_next

# -------------------------------------------------------------
# Checking number function
# Register used:

# Register s0: Store the Address of Array to store numbers to remember
# Register t0: Store the number to be checked
# Register t1: Stack to store numbers taken from answer for checking
# Register t2: Store the begining Address of the stack
# Register t3: Store the correct numbers in the Array
# Register t4: Holds some extra values (changing frequently)
# Register t5: Iterators through Array of corrected numbers
# Register s11: Indicates the number of answers can be input
#-------------------------------------------------------------
check_numbers:
    li t4, -1
    beq t0, t4, check_each_number             # If there is no numbers left to push into stack then start to check
    sw t0, 0(t1)                   # Else, store it
```

```
# Register t3: Store the character read from answer
```

```
   addi s11, s11, 1                # Increment the numbers of answer by 1
   bnez s11,  less_input_issue
```

```
check_each_number:
   la s0, numbers
   la t2, answer_numbers_stack
   addi t2, t2, -4
   beq t1, t2, finish_check                # Finish checking all the extracted numbers
check_phase:
   li t5, 0
check_loop:
   li t4, 4
   beq t5, t4, check_next                  # If the answer doesn't match then check the next number
   lw t0, 0(t1)
   lw t3, 0(s0)
   beq t0, t3, mdf_arr                     # If the answer match then modify the array and move to the next answer
   add s0, s0, t4
   addi t5, t5, 1                          # Else continue
   j check_loop
check_next:
   addi t1, t1, -4                         # Next one
   j check_each_number

mdf_arr:
   li t4, -1
   sw t4, 0(s0)                    # Replace the matched numbers in the Array by -1 to avoid the duplicated answer.
   j check_next

# ------------------------------------------------------------
# Description: Handle out of range issue
#-------------------------------------------------------------
out_of_range:
   jal clear_display
   li s6, 0                # x = 0
   li s7, 0                # y = 1

   jal move_cur_pos
   la a0, error_msg2
```

```
beqz t5, back_exit      # Finsh the game
   jal write_char
   addi a0, a0, 1
   addi s6, s6, 1          # Move cursor to the next position to write character
   j print_oor_msg
# ------------------------------------------------------------
# Description: Handle invalid issue
#-------------------------------------------------------------
invalid:
   jal clear_display
   li s6, 0                # x = 0
   li s7, 0                # y = 1

   jal move_cur_pos
   la a0, error_msg1
   li t4, NEWLINE
print_invalid_msg:
   lb t5, 0(a0)
   beqz t5, back_exit  # Finsh the game
   jal write_char
   addi a0, a0, 1
   addi s6, s6, 1          # Move cursor to the next position to write character
   j print_invalid_msg

# ------------------------------------------------------------
# Description: Conclude the result based on the Array
# Register used:
# Register s0: Store the Address of Array to store numbers to remember
# Register t0: Store the correct numbers in the Array
# Register t1: Iterators through Array of corrected numbers
# Register t4: Holds some extra values (changing frequently)
#-------------------------------------------------------------
finish_check:
   la s0, numbers
   li t1, 0
final_check:
  li t4, NEWLINE
print_oor_msg:
  lb t5, 0(a0)
```

```
  li t4, 4
  beq t1, t4, corrected_result          # If all the numbers
in the Array are negative then it's corrected answer
```

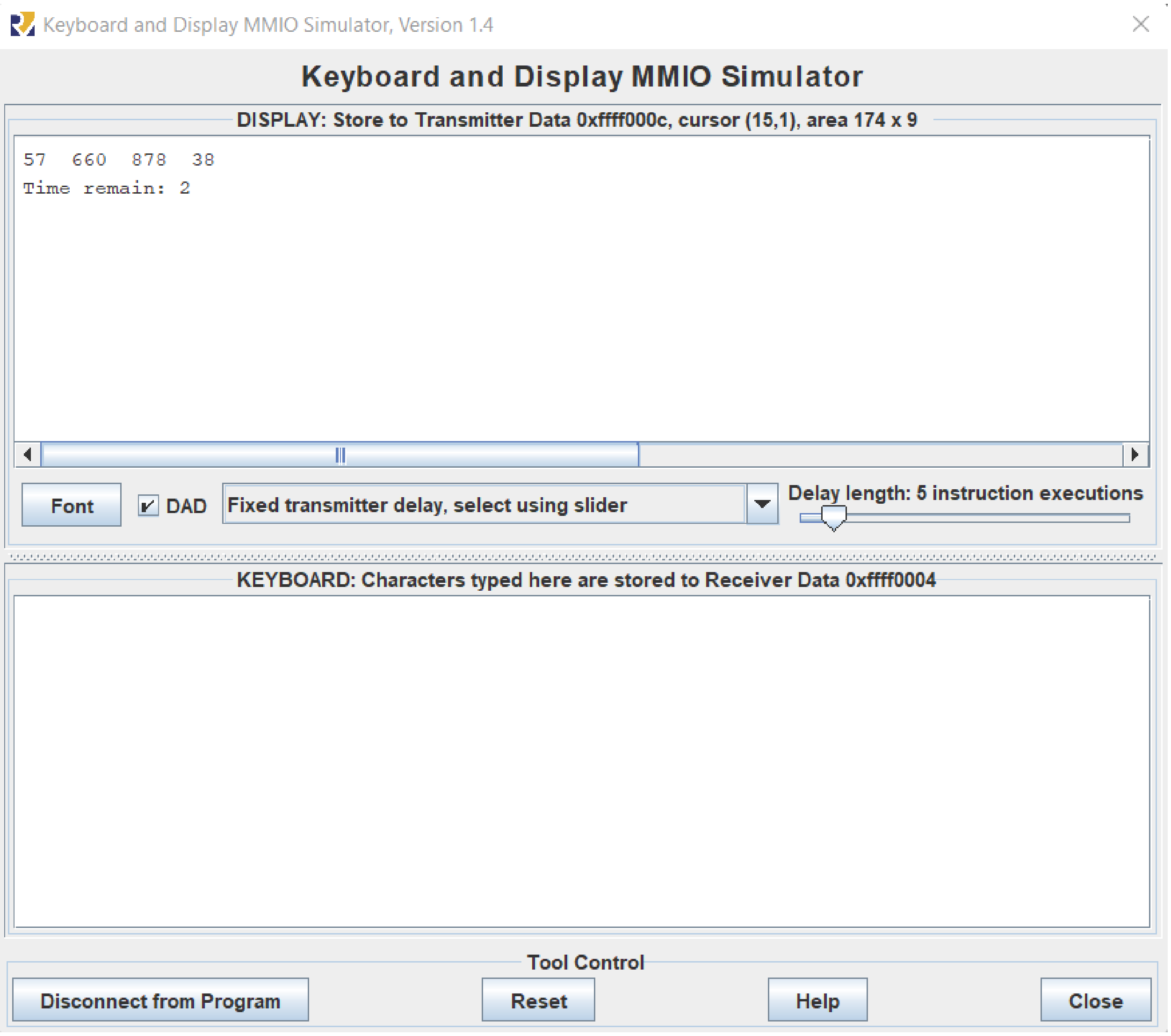
```
  lw t0, 0(s0)
  bge t0, zero, wrong_result            # If there is a
number that is greater or equal 0
                                        # which means
unmatched number then the answer is wrong
  add s0, s0, t4
  addi t1, t1, 1
  j final_check
# ------------------------------------------------------------
# Description: Show out the corrected message
#-------------------------------------------------------------
corrected_result:
  jal clear_display
  li s6, 0                  # x = 0
  li s7, 0                  # y = 1
  jal move_cur_pos
  la a0, correct_msg
  li t4, NEWLINE
print_corrected_msg:
  lb t5, 0(a0)
  beqz t5, back_next # Finsh the game
  jal write_char
  addi a0, a0, 1
  addi s6, s6, 1          # Move cursor to the next
position to write character
  j print_corrected_msg
# ------------------------------------------------------------
# Description: Show out the too many input issue
message
#-----------------------------------------------------------
more_input_issue:
  jal clear_display
  li s6, 0                  # x = 0
  li s7, 0                  # y = 1
  jal move_cur_pos
  la a0, error_msg3
print_more_input_issue:
  lb t5, 0(a0)
  beqz t5, back_exit  # Finsh the game
  jal write_char
  addi a0, a0, 1
```

```

# ------------------------------------------------------------
# Description: Show out the less input issue message
#-----------------------------------------------------------
less_input_issue:
  jal clear_display
  li s6, 0                  # x = 0
  li s7, 0                  # y = 1

  jal move_cur_pos
  la a0, error_msg3
print_less_input_issue:
  lb t5, 0(a0)
  beqz t5, back_exit  # Finsh the game
  jal write_char
  addi a0, a0, 1
  addi s6, s6, 1          # Move cursor to the next
position to write character
  j print_less_input_issue

# ------------------------------------------------------------
# Description: Show out the wrong message
#-------------------------------------------------------------
wrong_result:
  jal clear_display
  li s6, 0                  # x = 0
  li s7, 0                  # y = 1

  jal move_cur_pos
  la a0, wrong_msg
  li t4, NEWLINE
print_wrong_msg:
  lb t5, 0(a0)
  beqz t5, back_exit  # Finsh the game
  jal write_char
  addi a0, a0, 1
  addi s6, s6, 1          # Move cursor to the next
position to write character
  j print_wrong_msg
```

addi s6, s6, 1 # Move cursor to the next position to write character j print_more_input_issue	exit: li a7, 10 ecall

5. Simulation result:

Time decreases (5s -> 2s):



Keyboard and Display MMIO Simulator, Version 1.4

Keyboard and Display MMIO Simulator

DISPLAY: Store to Transmitter Data 0xffff000c, cursor (15,1), area 174 x 9

```
57  660  878  38
Time remain: 2
```

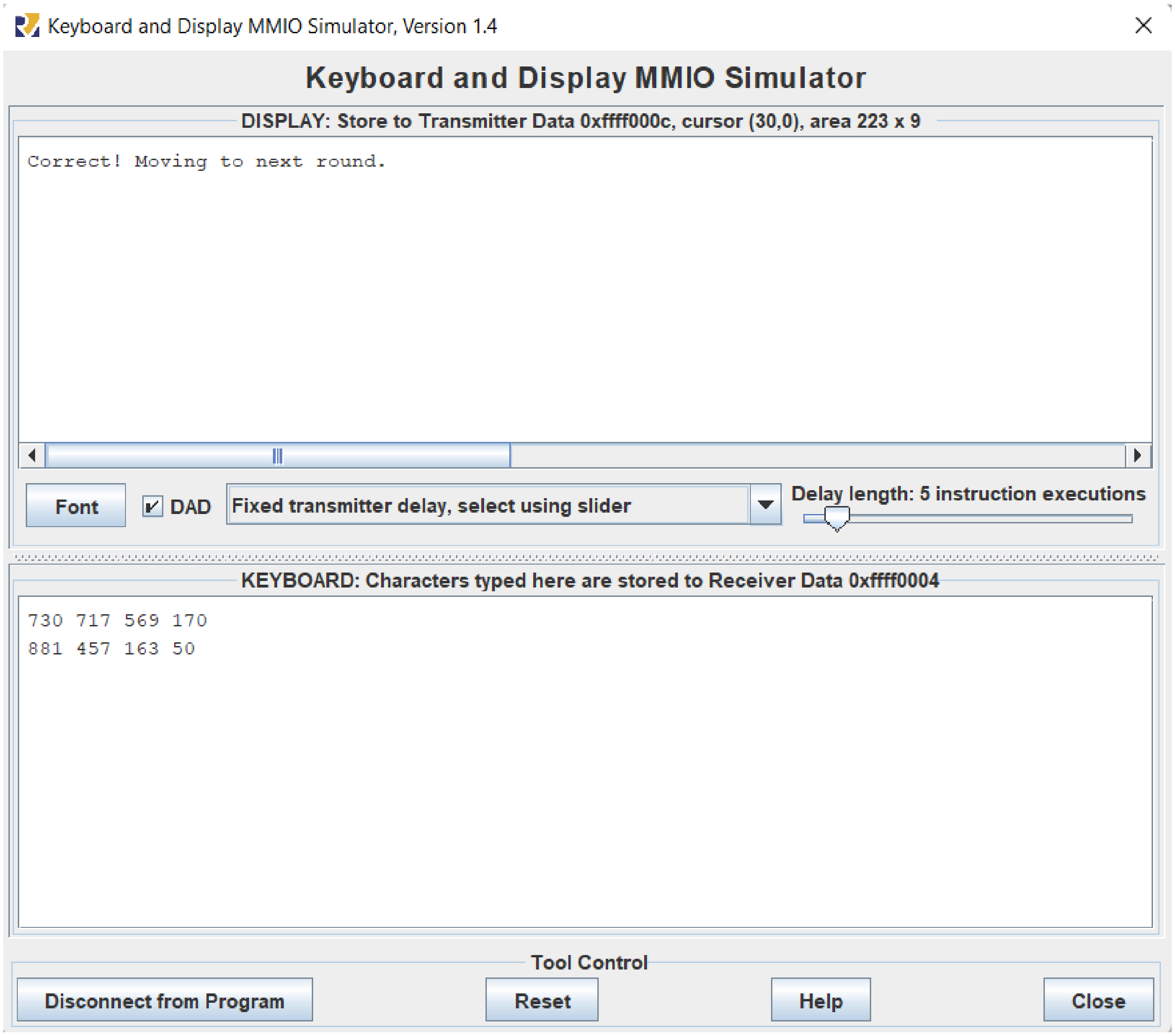
☑ DAD

Fixed transmitter delay, select using slider

Delay length: 5 instruction executions

KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004

Correct answer:



Keyboard and Display MMIO Simulator, Version 1.4
Keyboard and Display MMIO Simulator
DISPLAY: Store to Transmitter Data 0xffff000c, cursor (30,0), area 223 x 9

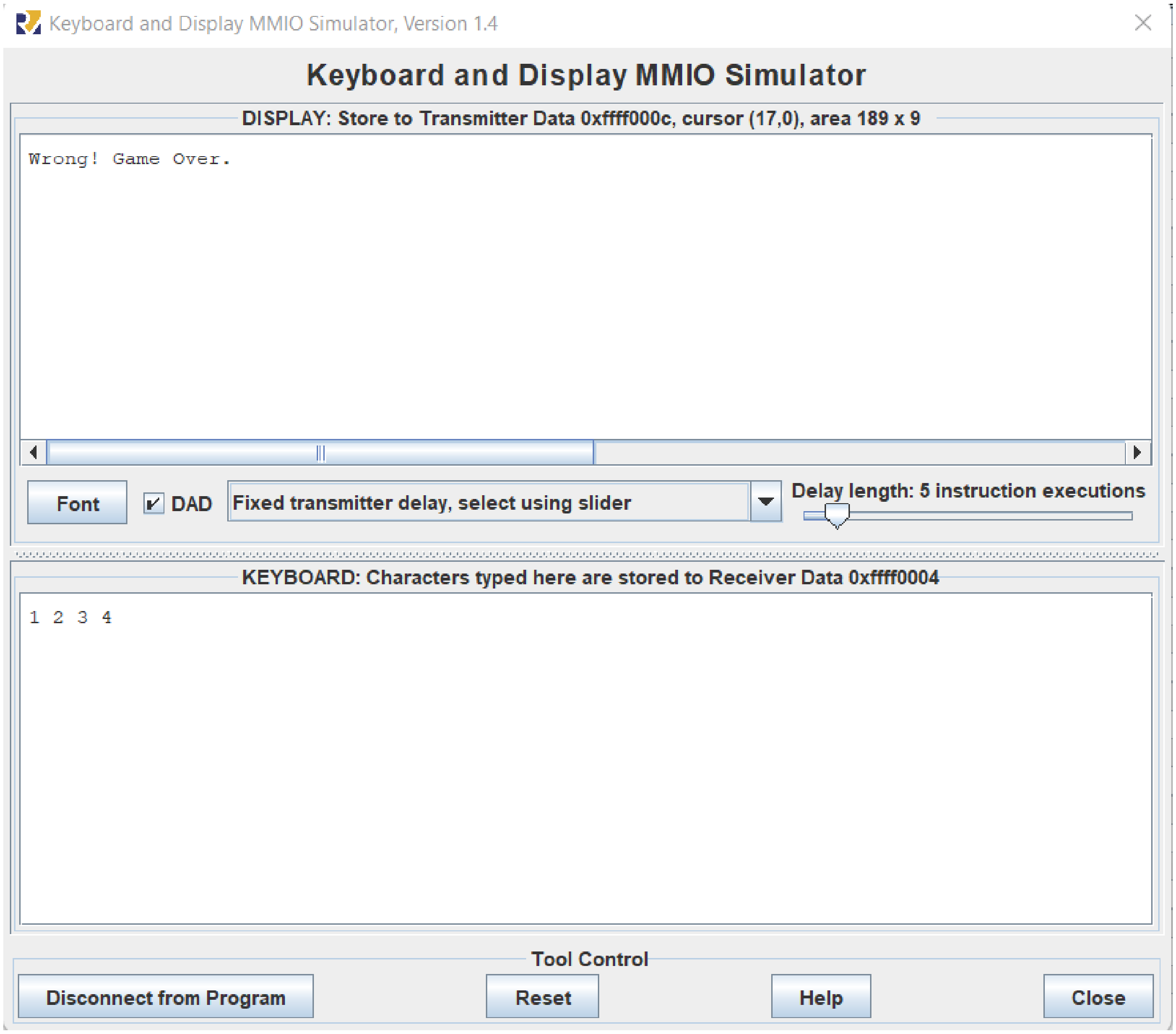
```
Correct! Moving to next round.
```

Font
☑ DAD
Fixed transmitter delay, select using slider
Delay length: 5 instruction executions
KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004

```
730 717 569 170
881 457 163 50
```

Tool Control
Disconnect from Program
Reset
Help
Close

Wrong answer:



Keyboard and Display MMIO Simulator, Version 1.4

Keyboard and Display MMIO Simulator

DISPLAY: Store to Transmitter Data 0xffff000c, cursor (17,0), area 189 x 9

```
Wrong! Game Over.
```

☑ DAD

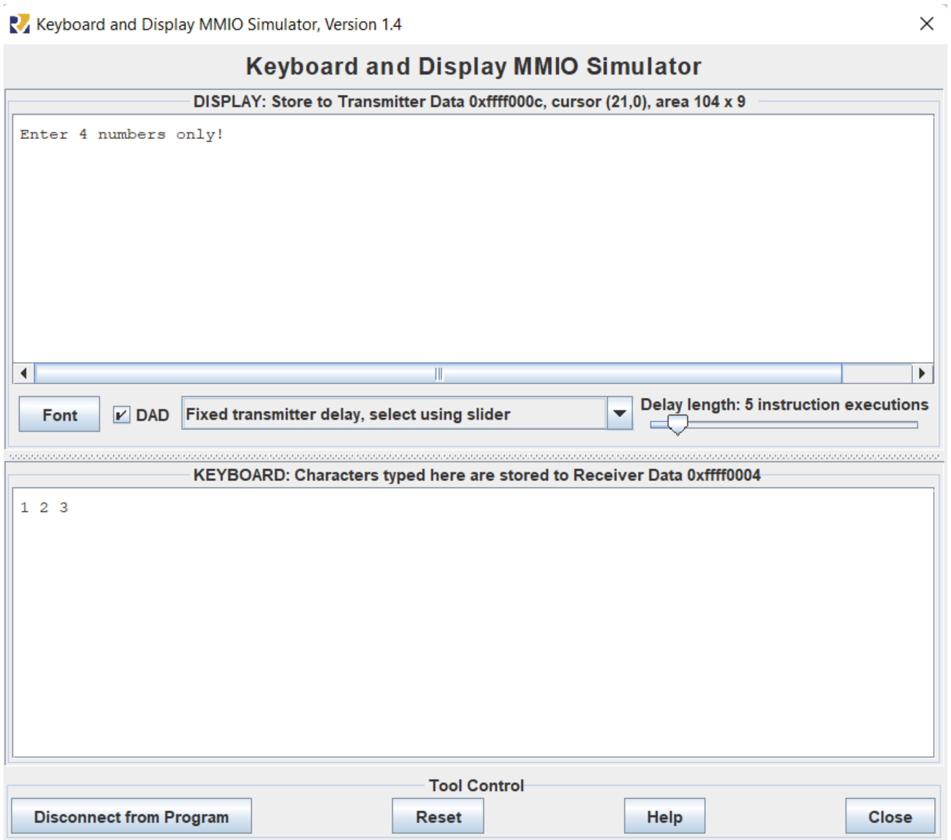
Fixed transmitter delay, select using slider

Delay length: 5 instruction executions

KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004

```
1 2 3 4
```

Too many or too few input numbers:



Keyboard and Display MMIO Simulator, Version 1.4

Keyboard and Display MMIO Simulator

DISPLAY: Store to Transmitter Data 0xffff000c, cursor (21,0), area 104 x 9

```
Enter 4 numbers only!
```

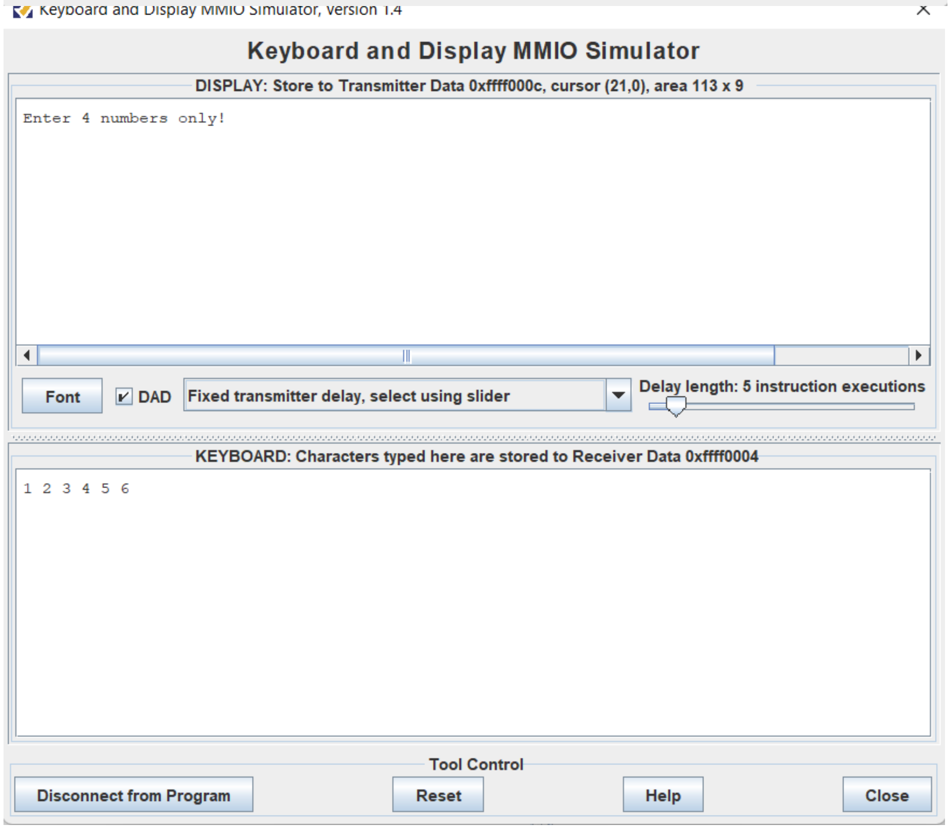
☑ DAD

Fixed transmitter delay, select using slider

Delay length: 5 instruction executions

KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004

```
1 2 3
```



Keyboard and Display MMIO Simulator, Version 1.4

Keyboard and Display MMIO Simulator

DISPLAY: Store to Transmitter Data 0xffff000c, cursor (21,0), area 113 x 9

```
Enter 4 numbers only!
```

☑ DAD

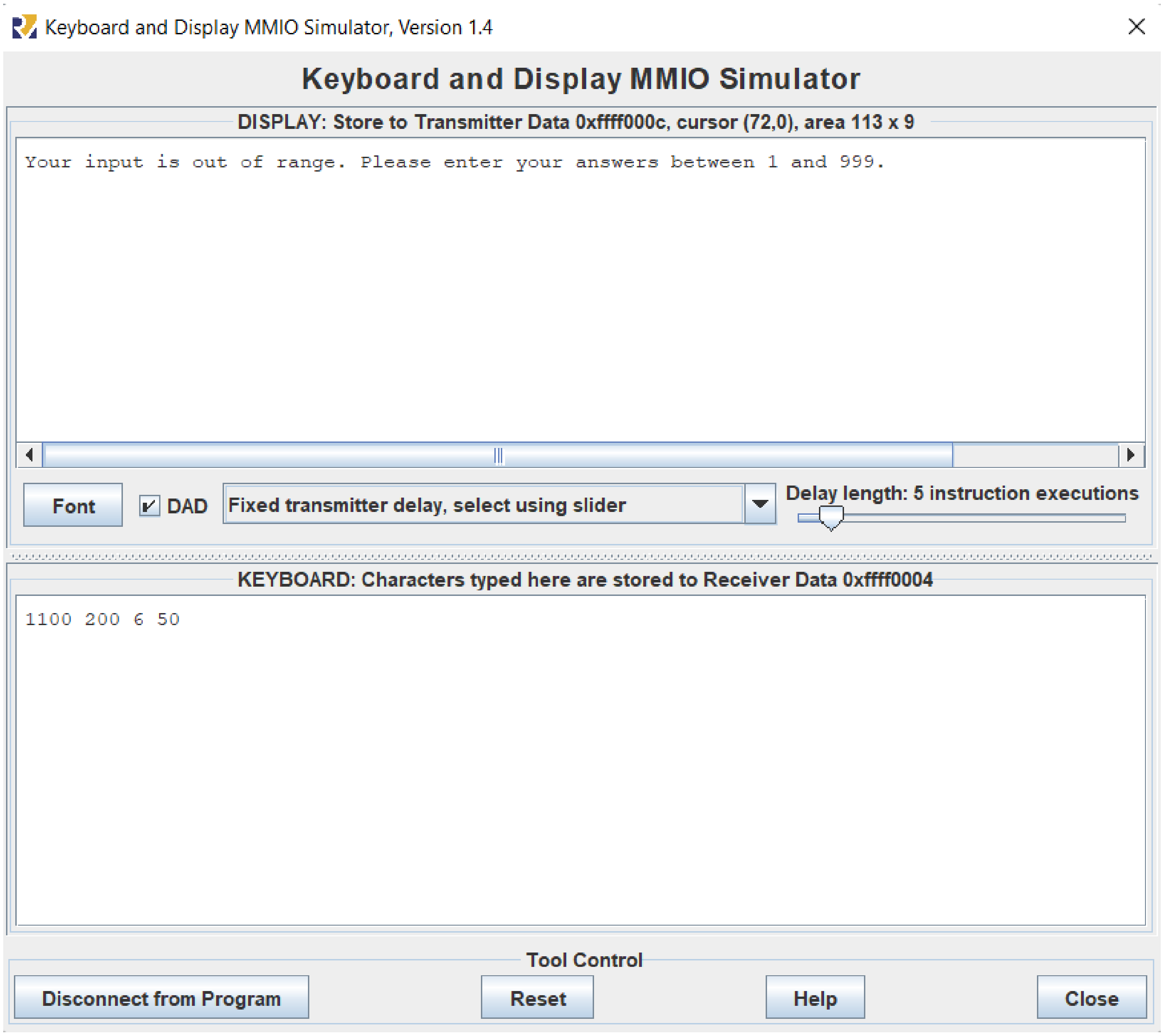
Fixed transmitter delay, select using slider

Delay length: 5 instruction executions

KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004

```
1 2 3 4 5 6
```

Out of range:



Keyboard and Display MMIO Simulator, Version 1.4
Keyboard and Display MMIO Simulator
DISPLAY: Store to Transmitter Data 0xffff000c, cursor (72,0), area 113 x 9

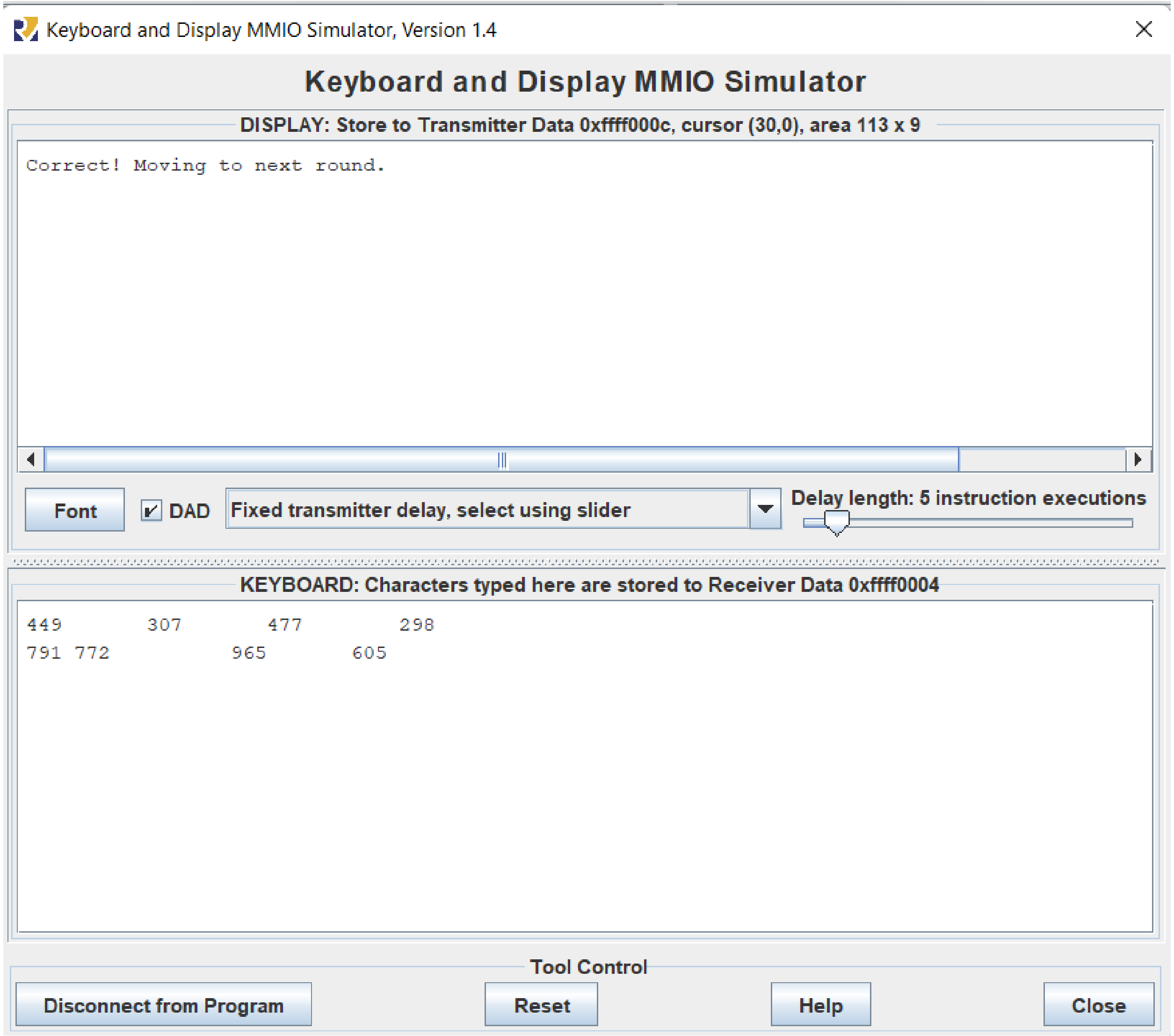
```
Your input is out of range. Please enter your answers between 1 and 999.
```

Font
☑ DAD
Fixed transmitter delay, select using slider
Delay length: 5 instruction executions
KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004

```
1100 200 6 50
```

Tool Control
Disconnect from Program
Reset
Help
Close

Many space: (still correct!!!)


Keyboard and Display MMIO Simulator, Version 1.4
Keyboard and Display MMIO Simulator
Font
Tool Control
Disconnect from Program
Reset
Help
Close

DISPLAY: Store to Transmitter Data 0xffff000c, cursor (30,0), area 113 x 9

```
Correct! Moving to next round.
```

☑ DAD

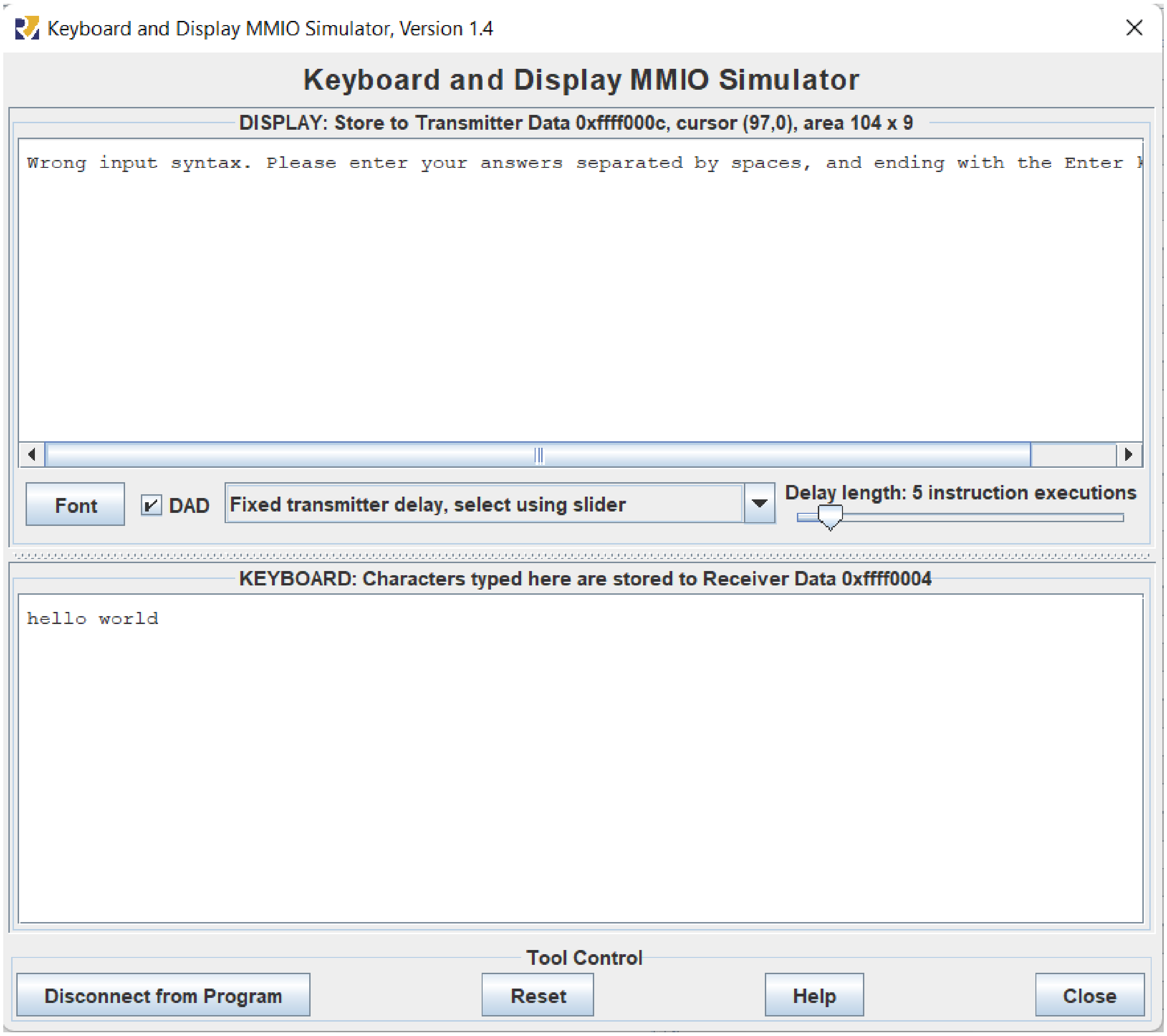
Fixed transmitter delay, select using slider

Delay length: 5 instruction executions

KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004

```
449       307       477        298
791 772          965       605
```

Invalid input:



Keyboard and Display MMIO Simulator, Version 1.4

Keyboard and Display MMIO Simulator

DISPLAY: Store to Transmitter Data 0xffff000c, cursor (97,0), area 104 x 9

```
Wrong input syntax. Please enter your answers separated by spaces, and ending with the Enter
```

☑ DAD

Fixed transmitter delay, select using slider

Delay length: 5 instruction executions

KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004

```
hello world
```