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

Final Project Report

<u>IT3280E - Assembly Language and Computer Architecture Lab</u>

Instructor: M.S. Le Ba Vui

Teaching Assistant: Nguyen Trung Kien

Students:

Name	**Student ID**
Phung Duc Dat	20226025
Ha Minh Duc	20226027

Ha Noi, 2024

Table of content

Part I: Typing Test

I. Problem Description

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

- Given a sample text, hardcoded in the source code.

- Use the timer to create measurement intervals. This is the time between two consecutive interruptions.

- The user enters text from the keyboard, the program will count the number of correct characters and display it with LEDs.

- The program also needs to estimate typing speed as the number of words per unit of time.

II. Method and Algorithms

1. Setup

```
.eqv SEVENSEG_LEFT 0xFFFF0011  # Simulated left 7-segment display
.eqv SEVENSEG_RIGHT 0xFFFF0010 # Simulated right 7-segment display
.eqv KEY_CODE 0xFFFF0004          # Simulated keyboard input code
.eqv KEY_READY 0xFFFF0000         # Keyboard input ready flag

.data
num_arr: .byte 63, 6, 91, 79, 102, 109, 125, 7, 127, 111  # 7-segment values for digits from 0 to 9
string: .asciz "Computer Architecture Lab"                  # Char count: 25
msg_display_time: .asciz "\nElapsed time: "
msg_display_wpm: .asciz "(s) \nAverage typing speed: "
msg_wpm: .asciz " words/minute\n"
msg_continue: .asciz "Do you want to continue to test typing speed?"
```

We declare some necessary variables, messages and an array for displaying correct character count.

2. Initialize data

```
.text
.globl main
main:
        li a7, 30       # Load syscall 30 to obtain current time in ms
        ecall
        mv t5, a0               # Save the start time in t5

        li s0, 0        # Correct character count
        li s1, 0        # Word count
        li s2, 32       # Initialize previous character to space
        la a1, string           # Load string base address
```

At the beginning of the main procedure, we retrieve current time using ***syscall 30*** and store it in ***t5***. Then we initialize some variables before continuing as shown above in the source code

3. Input handling

3.1 Main loop

```
# Main Loop
loop:
        li t0, KEY_READY        # KEY_READY address
        lb t1, 0(t0)            # Read KEY_READY
        bnez t1, keyboard_intr  # If key is ready, handle input
        j loop          # Repeat loop
```

The loop is used to check if we are ready to handle input using the Keyboard and Display MMIO Simulator tool

3.2 Keyboard interrupt handling

```
# Keyboard Interrupt Handling
keyboard_intr:
        li t0, KEY_CODE # KEY_CODE address
        lb t2, 0(t0)            # Load key pressed
        beqz t2, loop           # Skip if no key pressed

        # Check for backspace key
        li t4, 8          # ASCII for backspace
        beq t2, t4, handle_backspace # Handle backspace input

        # Check for enter key
        li s3, 10         # ASCII for newline/enter
        beq t2, s3, display_result # If user press enter, display testing result

        # Compare with current character of hardcoded string
        lb t3, 0(a1)            # Load current string character
        beq t2, t3, handle_correct_char # If match, count as correct
        j handle_space          # Handle space key input
```

If a key is pressed, its value will be stored in ***t2***. If it is an enter key the testing process will be stopped and testing results will be displayed using ***display_result*** procedure. The 3 procedures: ***handle_backspace***, ***handle_correct_char***, ***handle_space*** and are used to handle specific types of input character.

3.3 Special input cases

a, Backspace

```
handle_backspace:
        la t6, string           # Base address of the string
        beq a1, t6, loop        # If at start of string, do nothing and comeback to loop
        addi a1, a1, -1         # Move string pointer back a character

# Check if the character being backspaced matches the current character in the string
# If it does, the correct character count is decremented to ensure correct char count
check_correct_count:
        lb t3, 0(a1)            # Reload the string character
        beq t3, s2, decrement_correct_count # Decrease correct char count

# Decrease the correct char count when a backspaced character was counted as
correct
decrement_correct_count:
        addi s0, s0, -1         # Decrement correct character count
        j loop
```

There are 4 cases that might happen when the backspace key is pressed

+ We are currently at the start of the string. If we press backspace nothing should happen. Here we load the base address of the string to ***t6***, if ***t6*** is equal to ***a1*** then we are at the start of the string, the program will just jump back to ***loop*** to read input. Else we will decrease the string pointer ***a1*** back a character.

+ After inputting a correct character, we use backspace to delete it and input it again. In this case we need to make sure that the correct character count is correctly calculated. The ***check_correct_count*** procedure checks if the current string character is the same as our previously inputted character. If they are the same then we need to decrease the correct character count by 1 by the ***decrement_correct_count*** procedure to make sure it's correctly calculated.

+ After inputting a correct character, we use backspace to delete it and input an incorrect character. In this case, the correct character count will be decreased by 1 and won't be increased again since the new input is wrong.

+ After inputting an incorrect character, we use backspace to delete it and input a new character. In this case, the correct character count will be decreased by 1 and won't be increased again since the input is already wrong in the first place.

b, Correct character

```
handle_correct_char:
        addi s0, s0, 1          # Increment correct character count
        j handle_space
```

This procedure is used to handle correctly inputted characters. The correct character count stored in ***s0*** will be increased by 1 and the program will jump to ***handle_space*** procedure to handle space character input

c, Space character

```
# Handle space input
handle_space:
        li t4, 32       # ASCII for space
        beq t2, t4, check_prev_space # If space is pressed, jump to check
        j check_new_word

# Check if previour character is also a space
# Preventing incorrect word count by ignoring consecutive spaces
check_prev_space:
        li t4, 32
        beq s2, t4, update
        j update

# Check if we reach a new word
check_new_word:
        li t4, 32               # ASCII for space
        beq s2, t4, increment_word_count # If previous char is space, new word starts
        j update

increment_word_count:
        addi s1, s1, 1          # Increment word count

# Update previous character and move to next character
update:
        mv s2, t2               # Update previous character
        addi a1, a1, 1          # Move to next character of the string
        j loop
```

- Logic: We update word count if and only if the current character(stored in ***t2***) is not a space character and the previous character(stored in ***s2***) is a space character.

- ***handle_space***: If the space key is pressed the program jumps to ***check_prev_space*** else jumps to ***check_new_word***.

- ***check_prev_space***: The program jumps to the ***update*** procedure to move to the next character in the string and take input again whether the previous character is a space character or not.

- ***check_new_word***: If the previous character is a space and the current is not(according to ***handle_space***) we increase the word count by 1 then update

- Some edge cases:

+ If we press space then press enter, the word count should still be 0. In the ***main*** procedure we initialize the previous character ***s2*** to be the space character, this guarantees that if we just press space and press enter to end the testing process the word count is still 0 according to the logic implemented in ***check_prev_space*** procedure.

+ If the input ends with a space, the word count shouldn't be changed. The logic implemented in ***check_prev_space*** also cover this since whether the previous character is a space character or not, the program will always jump to ***update*** and the word count remain unchanged

4. Displaying results

```
# Display Testing Results
display_result:
        li a7, 30       # Load syscall 30 to obtain current time in ms
        ecall
        mv t6, a0               # Save the end time in t6

        li a7, 4
        la a0, msg_display_time
        ecall           # Print "Elapsed time: "

        li a7, 1
        sub s4, t6, t5          # Test time = End time - Start time
        li s5, 1000
        div s4, s4, s5          # Divide milliseconds by 1000 to obtain seconds
        mv a0, s4
        ecall           # Print elapsed time

        li a7, 4
        la a0, msg_display_wpm
        ecall           # Print "Average typing speed: "

        li t0, 60       # Multiplier for words per minute
        mul t1, s1, t0
        div a0, t1, s4          # Calculate speed (words per minute)
        li a7, 1
        ecall           # Print average typing speed in wpm

        li a7, 4
        la a0, msg_wpm
        ecall           # Print "words/minute"

        jal ra, display_led   # Display correct character count on 7-segment LEDS
        j continue

# Display Correct Character Count on 7-Segment LEDS
display_led:
        li t0, 10
        div t1, s0, t0          # Divide correct char count by 10 to obtain tens digit
        rem t2, s0, t0          # Remainder of s0 / 10 is the unit digit

        la a0, num_arr
        add a0, a0, t1
        lb a1, 0(a0)            # Load value of tens digit
        li t0, SEVENSEG_LEFT
        sb a1, 0(t0)            # Display on left 7-segment

        la a0, num_arr
        add a0, a0, t2
        lb a1, 0(a0)            # Load value of unit digit
        li t0, SEVENSEG_RIGHT
        sb a1, 0(t0)            # Display on right 7-segment

        ret
```

The ***display_result*** procedure is used to display testing results and jump to ***continue*** procedure to display the confirm dialog

4.1 Display time related results

a, Elapsed time

- First we will use ***syscall 30*** again to retrieve current time in milliseconds and store it in ***t6***

- Elapsed time in seconds is obtained by subtracting the end time store in ***t6*** and start time store in ***t5*** then divided by 1000.

- This value is stored in ***s4*** and then is printed to the Run I/O window.

b, Typing speed (words/minute)

- Since we already have elapsed time, we can use it to calculate average typing speed in words/minute

- Word count stored in ***s1*** is multiplied by 60 and then divided by elapsed time stored in ***s4***

- This value is then printed to the Run I/O window.

4.2 Display correct character count

The display_led procedure displays the correct character count using 2 7-segment LEDS. We will need to calculate the tens digit and unit digit to correctly display them.

- To obtain the tens digits we simply divide correct character count stored in ***s0*** by 10

- The unit digit is the remainder of ***s0***/10

- With these 2 values we simply retrieve 2 given 7-segment values from the ***num_arr*** and display them in 2 7-segment LEDS

5. Continue to test or end program

```
continue:
        li a7, 50
        la a0, msg_continue
        ecall
        beq a0, zero, main # If Yes(a0=0), continue to test typing speed
        li a7, 10 # If No(a0 = 1), end the program
        ecall
```

- After the ***display_result*** procedure ends, the program jumps to ***continue*** procedure

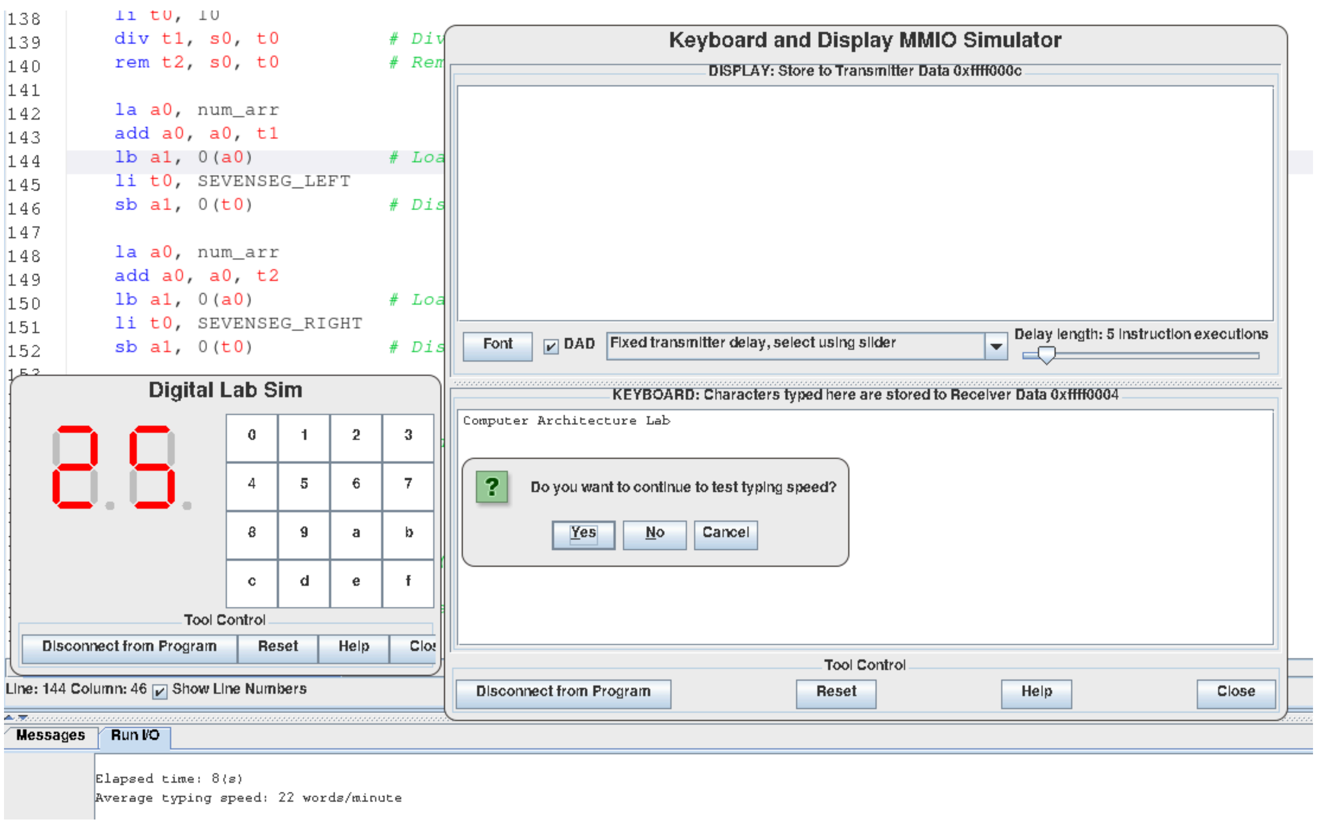
- We use ***syscall 50*** to display a confirm dialog to see if the user wants to continue the typing test or not.

+ If the user wants to continue by pressing Yes(***a0*** = 0), the program will jump back to ***main***, starting the testing process again from the start

+ If the user want to quit by pressing No(***a0*** = 1), the program will quit using ***syscall 10***

III. Simulation results

1. All characters are correct


Keyboard and Display MMIO Simulator
DISPLAY: Store to Transmitter Data 0xffff000c
Font
DAD
Fixed transmitter delay, select using slider
Delay length: 5 instruction executions
KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004
Tool Control
Disconnect from Program
Reset
Help
Close
Digital Lab Sim
0 1 2 3 4 5 6 7 8 9 a b c d e f
Messages
Run I/O

```
li t0, 10
div t1, s0, t0          # Div
rem t2, s0, t0          # Rem

la a0, num_arr
add a0, a0, t1
lb a1, 0(a0)            # Loa
li t0, SEVENSEG_LEFT
sb a1, 0(t0)            # Dis

la a0, num_arr
add a0, a0, t2
lb a1, 0(a0)            # Loa
li t0, SEVENSEG_RIGHT
sb a1, 0(t0)            # Dis
```

Line: 144 Column: 46 ✓ Show Line Numbers

```
Computer Architecture Lab
```

Do you want to continue to test typing speed?

Yes No Cancel

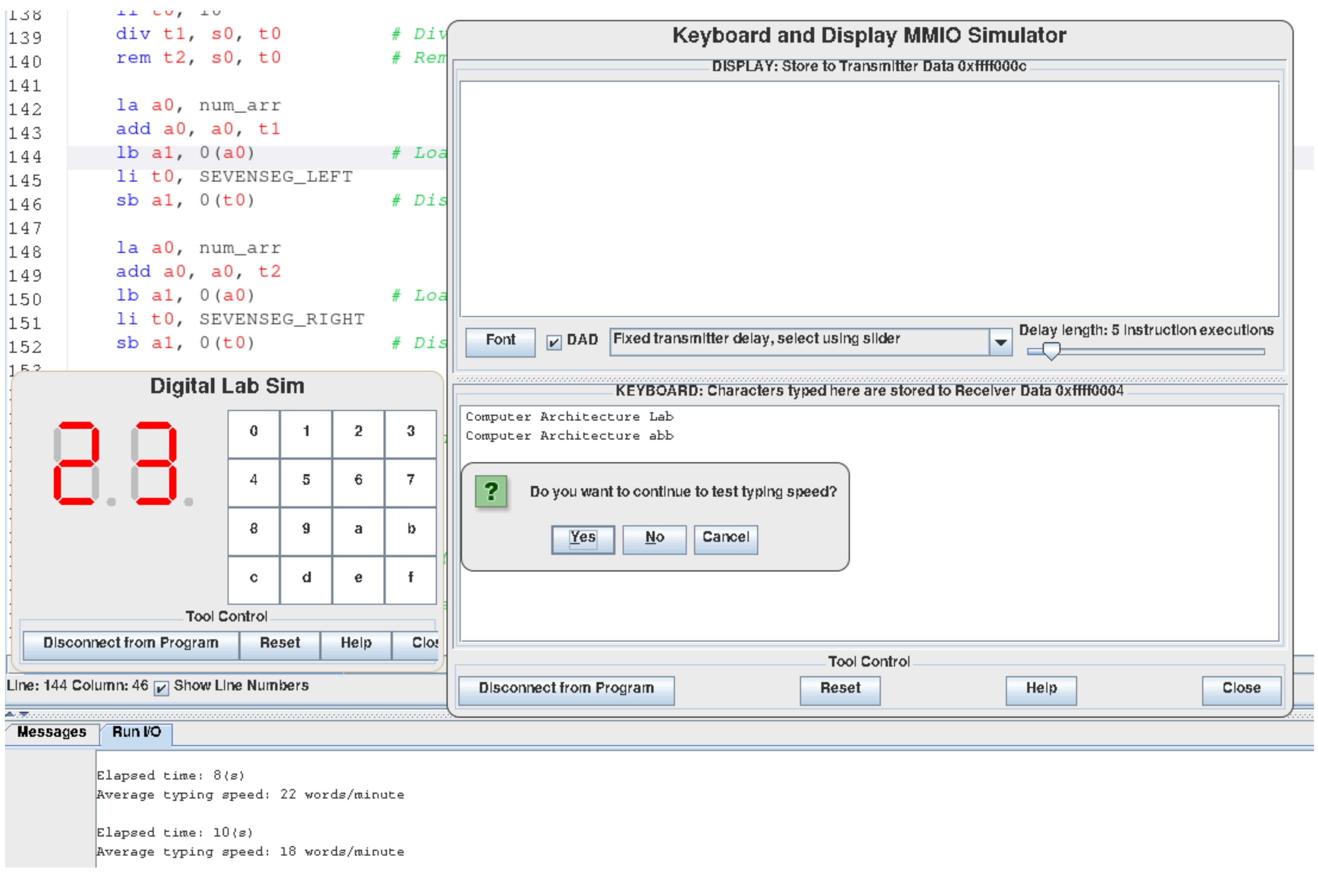
```
Elapsed time: 8(s)
Average typing speed: 22 words/minute
```

- The correct character count is 25(same as hardcoded string)

- The program display a confirm dialog after displaying results

- word count=3, elapsed time=8(s) => typing speed =3 x 60 / 8 = 22 words/minute

2. 2 incorrect characters



```
div t1, s0, t0          # Div
rem t2, s0, t0          # Rem

la a0, num_arr
add a0, a0, t1
lb a1, 0(a0)            # Loa
li t0, SEVENSEG_LEFT
sb a1, 0(t0)            # Dis

la a0, num_arr
add a0, a0, t2
lb a1, 0(a0)            # Loa
li t0, SEVENSEG_RIGHT
sb a1, 0(t0)            # Dis
```

Keyboard and Display MMIO Simulator

DISPLAY: Store to Transmitter Data 0xffff000c

DAD ✓ Fixed transmitter delay, select using slider — Delay length: 5 instruction executions

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

```
Computer Architecture Lab
Computer Architecture abb
```

Do you want to continue to test typing speed?

Line: 144 Column: 46 ✓ Show Line Numbers

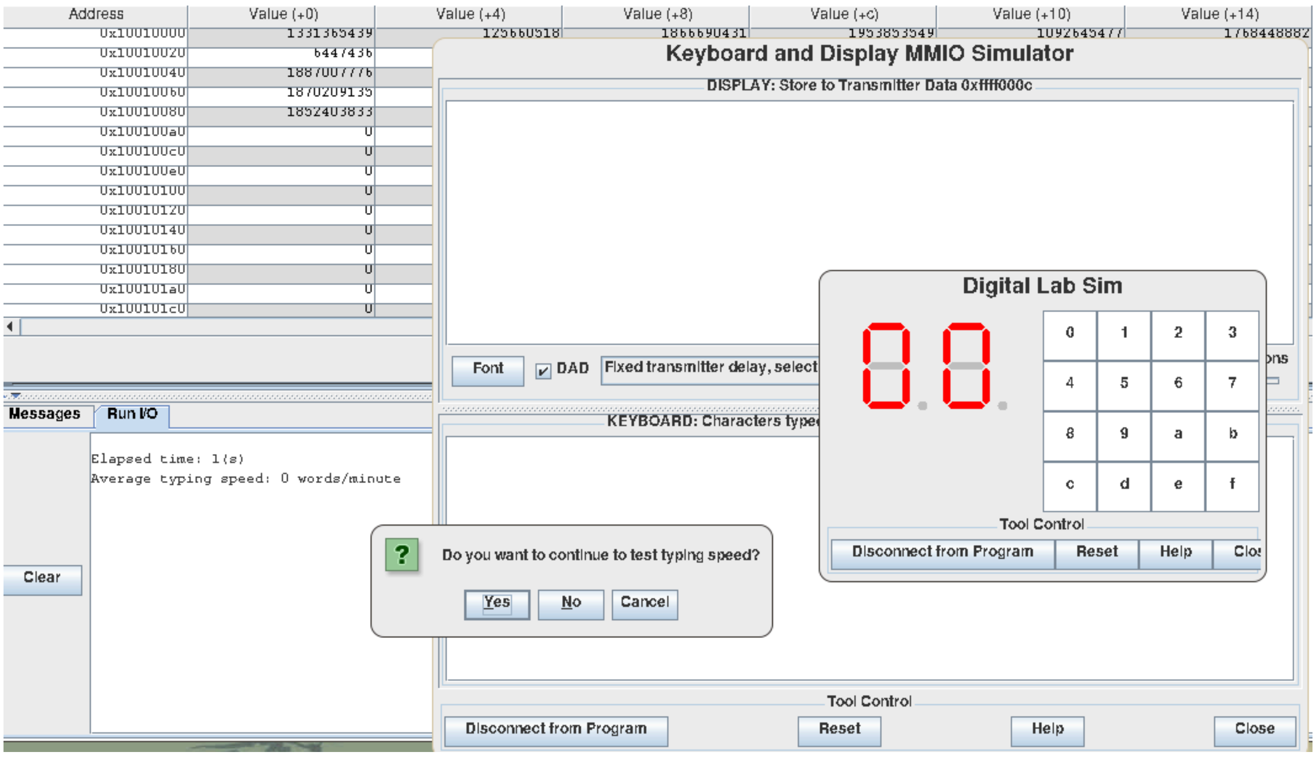
```
Elapsed time: 8(s)
Average typing speed: 22 words/minute

Elapsed time: 10(s)
Average typing speed: 18 words/minute
```

- The correct character count is now 23 because there are 2 incorrect characters(ab compares to La in the hardcoded string)

- word count=3, elapsed time=10(s) => typing speed=3 x 60 / 10 = 18 words/minute

3. Empty input



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)
0x10010000	1331365439	1256660518	1866690431	1953853549	1092645477	1768448882
0x10010020	6447436					
0x10010040	1887007776					
0x10010060	1870209135					
0x10010080	1852403833					
0x100100a0	0					
0x100100c0	0					
0x100100e0	0					
0x10010100	0					
0x10010120	0					
0x10010140	0					
0x10010160	0					
0x10010180	0					
0x100101a0	0					
0x100101c0	0					

```
Elapsed time: 1(s)
Average typing speed: 0 words/minute
```

Do you want to continue to test typing speed?

Keyboard and Display MMIO Simulator
DISPLAY: Store to Transmitter Data 0xffff000c
Font
DAD
Fixed transmitter delay, select
KEYBOARD: Characters typed
Tool Control
Disconnect from Program
Reset
Help
Close
Digital Lab Sim
0 1 2 3
4 5 6 7
8 9 a b
c d e f
Messages
Run I/O
Clear
Yes
No
Cancel

If the input is empty then typing speed is 0 since there is no word, correct character count will also be 0

4. Special cases

There are some special cases mentioned in the previous part that can’t be shown directly because they include more steps. You can test it yourself with the provided source code below

IV. Source code

```
.eqv SEVENSEG_LEFT 0xFFFF0011  # Address of left 7-segment led
.eqv SEVENSEG_RIGHT 0xFFFF0010 # Address of right 7-segment led
.eqv KEY_CODE 0xFFFF0004          # Simulated keyboard input code
.eqv KEY_READY 0xFFFF0000         # Keyboard input ready flag

.data
num_arr: .byte 63, 6, 91, 79, 102, 109, 125, 7, 127, 111  # 7-segment values for digits
from 0 to 9
string: .asciz "Computer Architecture Lab"                  # Char count: 25
msg_display_time: .asciz "\nElapsed time: "
msg_display_wpm: .asciz "(s) \nAverage typing speed: "
msg_wpm: .asciz " words/minute\n"
msg_continue: .asciz "Do you want to continue to test typing speed?"

.text
.globl main
main:
        li a7, 30       # Load syscall 30 to obtain current time in ms
        ecall
        mv t5, a0               # Save the start time in t5

        li s0, 0        # Correct character count
        li s1, 0        # Word count
        li s2, 32       # Initialize previous character to space
        la a1, string           # Load string base address

# Main Loop
loop:
        li t0, KEY_READY        # KEY_READY address
        lb t1, 0(t0)            # Read KEY_READY
        bnez t1, keyboard_intr  # If key is ready, handle input
        j loop          # Repeat loop

# Keyboard Interrupt Handling
keyboard_intr:
        li t0, KEY_CODE # KEY_CODE address
        lb t2, 0(t0)            # Load key pressed
        beqz t2, loop           # Skip if no key pressed
        # Check for backspace key
        li t4, 8          # ASCII for backspace
        beq t2, t4, handle_backspace # Handle backspace input

        # Check for enter key
        li s3, 10         # ASCII for newline/enter
        beq t2, s3, display_result # If user press enter, display testing result

        # Compare with current character of hardcoded string
        lb t3, 0(a1)            # Load current string character
        beq t2, t3, handle_correct_char # If match, count as correct
        j handle_space          # Handle space key input

# Input character match string character
handle_correct_char:
        addi s0, s0, 1          # Increment correct character count
        j handle_space

# Handle space input
handle_space:
        li t4, 32         # ASCII for space
        beq t2, t4, check_prev_space # If space is pressed, jump to check
        j check_new_word

# Check if previour character is also a space
# Preventing incorrect word count by ignoring consecutive spaces
check_prev_space:
        li t4, 32
        beq s2, t4, update
        j update

# Check if we reach a new word
check_new_word:
        li t4, 32               # ASCII for space
        beq s2, t4, increment_word_count # If previous char is space, new word starts
        j update

increment_word_count:
        addi s1, s1, 1          # Increment word count
# Update previous character and move to next character
update:
        mv s2, t2               # Update previous character
        addi a1, a1, 1          # Move to next character of the string
        j loop

# Correctly handle backspace key pressed
handle_backspace:
        la t6, string           # Base address of the string
        beq a1, t6, loop        # If at start of string, do nothing and comeback to loop
        addi a1, a1, -1         # Move string pointer back a character

# Check if the character being backspaced matches the current character in the string
# If it does, the correct character count is decremented to ensure correct char count
check_correct_count:
        lb t3, 0(a1)            # Reload the string character
        beq t3, s2, decrement_correct_count # Decrease correct char count

# Decrease the correct character count when a backspaced character was counted as
correct
decrement_correct_count:
        addi s0, s0, -1         # Decrement correct character count
        j loop

# Display Testing Results
display_result:
        li a7, 30       # Load syscall 30 to obtain current time in ms
        ecall
        mv t6, a0               # Save the end time in t6

        li a7, 4
        la a0, msg_display_time
        ecall           # Print "Elapsed time: "

        li a7, 1
        sub s4, t6, t5          # Test time = End time - Start time
        li s5, 1000
        div s4, s4, s5          # Divide milliseconds by 1000 to obtain seconds
        mv a0, s4
        ecall           # Print elapsed time

        li a7, 4
        la a0, msg_display_wpm
        ecall           # Print "Average typing speed: "
        li t0, 60       # Multiplier for words per minute
        mul t1, s1, t0
        div a0, t1, s4          # Calculate speed (words per minute)
        li a7, 1
        ecall           # Print average typing speed in wpm

        li a7, 4
        la a0, msg_wpm
        ecall           # Print "words/minute"

        jal ra, display_led   # Display correct character count on 7-segment LEDS
        j continue

# Display Correct Character Count on 7-Segment LEDS
display_led:
        li t0, 10
        div t1, s0, t0          # Divide correct char count by 10 to obtain tens digit
        rem t2, s0, t0          # Remainder of s0 / 10 is the units digit

        la a0, num_arr
        add a0, a0, t1
        lb a1, 0(a0)            # Load value of tens digit
        li t0, SEVENSEG_LEFT
        sb a1, 0(t0)            # Display on left 7-segment

        la a0, num_arr
        add a0, a0, t2
        lb a1, 0(a0)            # Load value of units digit
        li t0, SEVENSEG_RIGHT
        sb a1, 0(t0)            # Display on right 7-segment
        ret
# Show confirm dialog and continue or exit
continue:
        li a7, 50
        la a0, msg_continue
        ecall
        beq a0, zero, main # If Yes(a0=0), continue to test typing speed
        li a7, 10 # If No(a0 = 1), end the program
        ecall
```

Part II: Music Play

I. Problem Description

The task is to create a system that can play different musical scripts when a user interacts with a keyboard interface. The musical script consists of a string of values, each representing a specific note, its duration, instrument type, and volume. The system must allow the user to select which script to play by pressing keys on the keyboard, specifically keys 1 to 4. Pressing the "0" key will pause the music.

The program should be able to:

- Read inputs from the keyboard.
- Map key presses to specific musical scripts (song data).
- Play the chosen song by interpreting the four values in each set of the song data: pitch, duration, instrument type, and volume.
- Pause the music when the "0" key is pressed.

II. Source Code

```
.eqv IN_ADDRESS_HEXA_KEYBOARD  0xFFFF0012         # Input address for
the keyboard (memory-mapped I/O)
.eqv OUT_ADDRESS_HEXA_KEYBOARD 0xFFFF0014        # Output address for
the keyboard (memory-mapped I/O)

.data

# Key mappings for selecting songs (1-4) or pausing (0)
key_map:
        .word 0x21, song1_data  # Key 1 -> song1
        .word 0x41, song2_data  # Key 2 -> song2
        .word 0x81, song3_data  # Key 3 -> song3
        .word 0x12, song4_data   # Key 4 -> song4
        .word 0x11, pause_song   # Key 0 -> pause music
        .word 0                  # End of key map

# Song data for each song script: [Pitch, Duration, Instrument Type, Volume]
song1_data:
        .word 60, 500, 5, 100
        .word 60, 500, 5, 100
        .word 62, 500, 5, 100
        .word 64, 500, 5, 100
        .word 65, 500, 5, 100
        .word 67, 500, 5, 100
        .word 69, 500, 5, 100
        .word 71, 500, 5, 100
        .word 72, 500, 5, 100
        .word 74, 500, 5, 100
        .word 76, 500, 5, 100
        .word 77, 500, 5, 100
        .word 79, 500, 5, 100
        .word 81, 500, 5, 100
        .word 83, 500, 5, 100
        .word 84, 500, 5, 100
        .word 72, 500, 5, 100
        .word 71, 500, 5, 100
        .word 69, 500, 5, 100
        .word 67, 500, 5, 100
        .word 65, 500, 5, 100
        .word 64, 500, 5, 100
        .word 62, 500, 5, 100
        .word 60, 500, 5, 100
        .word 0, 0, 0, 0             # End of song data

song2_data:
        .word 64, 500, 0, 100
        .word 64, 500, 0, 100
        .word 66, 1000, 0, 100
        .word 64, 1000, 0, 100
        .word 69, 1000, 0, 100
        .word 68, 2000, 0, 100
        .word 64, 500, 0, 100
        .word 64, 500, 0, 100
        .word 66, 1000, 0, 100
        .word 64, 1000, 0, 100
        .word 71, 1000, 0, 100
        .word 69, 2000, 0, 100
        .word 64, 500, 0, 100
        .word 64, 500, 0, 100
        .word 76, 1000, 0, 100
        .word 72, 1000, 0, 100
        .word 69, 1000, 0, 100
        .word 68, 1000, 0, 100
        .word 66, 1000, 0, 100
        .word 74, 500, 0, 100
        .word 74, 500, 0, 100
        .word 72, 1000, 0, 100
        .word 69, 1000, 0, 100
        .word 71, 1000, 0, 100
        .word 69, 2000, 0, 100
        .word 0, 0, 0, 0            # End of song data

song3_data:
        .word 60, 500, 7, 100
        .word 60, 500, 7, 100
        .word 62, 500, 7, 100
        .word 64, 500, 7, 100
        .word 65, 500, 7, 100
        .word 64, 500, 7, 100
        .word 62, 500, 7, 100
        .word 60, 500, 7, 100
        .word 60, 500, 7, 100
        .word 60, 500, 7, 100
        .word 62, 500, 7, 100
        .word 64, 500, 7, 100
        .word 65, 500, 7, 100
        .word 64, 500, 7, 100
        .word 62, 500, 7, 100
        .word 60, 500, 7, 100
        .word 60, 500, 7, 100
        .word 62, 500, 7, 100
        .word 64, 500, 7, 100
        .word 65, 500, 7, 100
        .word 64, 500, 7, 100
        .word 62, 500, 7, 100
        .word 60, 500, 7, 100
        .word 0, 0, 0, 0                # End of song data

song4_data:
        .word 60, 500, 4, 100
        .word 60, 500, 4, 100
        .word 67, 500, 4, 100
        .word 67, 500, 4, 100
        .word 69, 500, 4, 100
        .word 67, 500, 4, 100
        .word 65, 500, 4, 100
        .word 65, 500, 4, 100
        .word 64, 500, 4, 100
        .word 64, 500, 4, 100
        .word 62, 500, 4, 100
        .word 60, 500, 4, 100
        .word 67, 500, 4, 100
        .word 67, 500, 4, 100
        .word 65, 500, 4, 100
        .word 65, 500, 4, 100
        .word 64, 500, 4, 100
        .word 64, 500, 4, 100
        .word 62, 500, 4, 100
        .word 60, 500, 4, 100
        .word 0, 0, 0, 0                # End of song data

# Pause song label (no music data, just halts playback)
pause_song:

error_msg: .asciz "Phim khong hop le\n"       # Error message for invalid key press
paused_msg: .asciz "Music Paused\n"           # Message when music is paused

.text
.globl main
main:
        li s1, IN_ADDRESS_HEXA_KEYBOARD  # Load input address for the
keyboard
        li s2, OUT_ADDRESS_HEXA_KEYBOARD # Load output address for the
keyboard
        li t3, 0x01                     # Set initial bit for key press checking

polling:
        sb t3, 0(s1)                    # Write the polling bit to the input address
        lbu a0, 0(s2)               # Load the current key press (byte) from the
keyboard

        beqz a0, next_row                   # If no key is pressed, skip to next row

        li t1, 17                   # Define key code for "0" (pause key)
        beq a0, t1, pause_music             # If the key pressed is "0", pause music

        mv a0, a0                   # Store key value in a0 for further processing
        call find_song              # Call function to find corresponding song
        beqz a0, show_invalid_key           # If no song found, show an error message
        call play_song              # Call function to play the selected song

        li a0, 500                  # Short delay after each iteration
        li a7, 32
        ecall

next_row:
        slli t3, t3, 1              # Shift bit to check the next key
        li t4, 0x10                 # Check if all keys have been polled
        bne t3, t4, continue_polling        # If not all keys checked, continue polling
        li t3, 0x01                 # Reset polling bit to check from the start

continue_polling:
        j polling                   # Jump back to polling loop

show_invalid_key:
        li a7, 4
        la a0, error_msg            # Load error message for invalid key
        ecall
        j polling                   # Continue polling for next input

pause_music:
        li a7, 33                   # System call to pause the music
        ecall
        li a7, 4
        la a0, paused_msg           # Load "Music Paused" message
        ecall
        j polling                   # Return to polling for new input

# Function to find which song corresponds to the pressed key
find_song:
        la t0, key_map              # Load the address of the key_map
find_song_loop:
        lw t1, 0(t0)                # Load the key value from the key map
        beqz t1, not_found          # If no match found, return
        bne a0, t1, next_entry      # If key doesn't match, go to next entry
        lw a0, 4(t0)                # If match, load the corresponding song data
        ret

next_entry:
        addi t0, t0, 8              # Move to next key in the map
        j find_song_loop

not_found:
        li a0, 0                    # Return 0 if no song found
        ret

# Function to play the selected song
play_song:
        mv t0, a0                   # Load song data address
play_song_loop:
        lbu t1, 0(s2)               # Check if the key is still pressed (if not, stop)
        li t2, 0x00
        beq t1, t2, return          # If no key is pressed, exit

        lw t1, 0(t0)                # Get the pitch of the current note
        beqz t1, return             # If pitch is 0, end of song

        lw a0, 0(t0)                # Load pitch
        lw a1, 4(t0)                # Load duration
        lw a2, 8(t0)                # Load instrument type
        lw a3, 12(t0)               # Load volume

        li a7, 31                   # System call to play sound
        ecall

        addi t0, t0, 16             # Move to the next note in the song
        j play_song_loop            # Continue playing the next note

return:
        ret
```

III. Method

The implementation relies on hardware interfacing using a memory-mapped keyboard system. The keyboard input is checked and processed to determine which song the user wishes to play or if they wish to pause the music. The program uses a polling method to continuously check for key presses.

1. Input Handling:

The system reads the user input from a memory-mapped I/O address, which is configured as:

- **IN_ADDRESS_HEXA_KEYBOARD (0xFFFF0012)**: Address for reading the keyboard input.
- **OUT_ADDRESS_HEXA_KEYBOARD (0xFFFF0014)**: Address for writing output to the keyboard or related peripherals.

The input is polled in a loop, where each key press corresponds to a specific action:

- Keys "1" to "4" map to different song scripts.
- Key "0" pauses the music.

The polling loop checks the value from the input address and processes the key press accordingly:

- If a valid key is pressed (1-4), the corresponding song is selected and played.
- If "0" is pressed, the music pauses.

2. Analysis of Execution

When the system is running:

- **Polling**: The keyboard is polled for input.
- **Key Identification**: Each key press is identified and mapped to a specific action:

➔ If the key corresponds to a song (1-4), the system retrieves and plays that song.
➔ If the key is "0", the music is paused.
➔ If an invalid key is pressed, an error message is displayed.

- **Music Playback**: After identifying the correct song, the system executes the sequence of song data (pitch, duration, instrument type, volume) to produce the desired sound using system calls.
- **Pausing Music**: When the "0" key is pressed, the music playback is halted and a pause message is shown.

3. Output

The output of the system consists of sound playback corresponding to the selected musical script and printed messages on the console. The key outputs are:

- **Error Message**: If an invalid key is pressed.
- **Paused Message**: If the music is paused using the "0" key.
- **Song Playback**: When the system successfully plays a selected song.

IV. Simulation Results

The program fulfills the requirements of the task. When the system runs, it waits for user input through the Digital Lab Sim. The program plays a song based on the key pressed (1-4) and pauses the music when the "0" key is pressed. Additionally, if an invalid key is pressed, an error message is shown, and the music continues to play without interruption.

Test Cases:

1. **Key 1: Plays Song 1.**
2. **Key 2: Plays Song 2.**
3. **Key 3: Plays Song 3.**
4. **Key 4: Plays Song 4.**
5. **Key 0: Pauses the music and displays a "Music Paused" message.**
6. **Invalid Key Press: Displays an error message "Phim khong hop le."**

The music system is responsive to user input and provides real-time feedback, allowing for a straightforward user experience in controlling the playback of different musical scripts.