**HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY**

SCHOOL OF INFORMATION COMMUNICATION TECHNOLOGY

# COMPUTER ARCHITECTURE LAB FINAL REPORT

Group Members:
Đoàn Tiến Dũng - 2022 0072
Bùi Hoàng Việt – 2022 6073

Project:

Checking the syntax of an instruction (5)

Digital lock (10)

# Exercise 10: Digital lock

## 1. Requirements:

Using the key matrix and 7-segment LEDs to program the electronic lock application with the following
requirements:

+ The password to unlock contains n digits (digits from 0 to 9, with at least 4 digits) pre-set in memory.

+ To open the door, the user enters the password by pressing the numeric buttons on the key matrix, ending with button F. If the password is correct, the lock opens, LEDs display ON. If the password is incorrect, LEDs display OF(F).

+ If the password is entered incorrectly more than 3 times, the lock will be suspended for 1 minute. During this time, all buttons will not work.

+ The user can change the password by pressing button A, then pressing the numeric buttons corresponding to the old password, then pressing button F. If the old password is correct, the system will ask for a new password. The user enters the new password by pressing the numeric buttons, then pressing button F to finish. The new password is overwritten with the old password in memory.

## 2. Workflow:

1. The system prompts the user to enter a password. The user enters digits from 0-9 and presses F to finish entering the password.

   If the password is correct: the LEDs display "ON" (unlocking the door).

   If the password is incorrect: the LEDs display "OFF" (door remains locked).

   If the password is entered incorrectly more than 3 times, the system will be suspended for 1 minute, during which no keys will work.

2. The user presses A to initiate the password change process.

   The user must enter the old password (using the numeric keys).

   If the old password is correct: the system prompts for a new password.
   The user enters the new password and presses F to confirm.
   The new password will overwrite the old password in memory.

If the user enters the wrong password more than 3 times, the system will lock out for 1 minute. During this period, all keys will be unresponsive.

## 3. Source code:

```
.eqv IN_ADDRESS_HEXA_KEYBOARD 0xFFFF0012
.eqv OUT_ADDRESS_HEXA_KEYBOARD 0xFFFF0014
.eqv SEVENSEG_LEFT 0xFFFF0011        # Address of the LED on the left
# Bit 0 = segment a
# Bit 1 = segment b
# ...
# Bit 7 = dot sign
.eqv SEVENSEG_RIGHT 0xFFFF0010       # Address of the LED on the right

.data
password: .byte 1, 2, 3, 4        # Mật khẩu ban đầu
buffer_password: .space 32
len: .word 4
buffer: .space 32                   # Lưu mật khẩu tối đa 16 ký tự
index: .word 0                      # Chỉ số hiện tại trong buffer, nơi lưu mật khẩu nhập vào
wrong_attempts: .word 0             # Biến đếm số lần nhập sai mật khẩu

msg_enter_password: .asciz "Please enter your password:\n"
msg_unlock_success: .asciz "Unlock successful! Welcome.\n"
msg_enter_old_password: .asciz "Please enter your old password:\n"
msg_enter_new_password: .asciz "Please enter your new password:\n"
msg_password_updated: .asciz "Password has been successfully updated.\n"
msg_password_wrong: .asciz "Incorrect password. Please try again.\n"
msg_lock_suspended: .asciz "Too many incorrect attempts. Lock is suspended for 1 minute.\n"
msg_press_A_to_change: .asciz "Press A to change the password.\n"
msg_short_password: .asciz "Password is too short. It must be at least 4 characters long.\n"

.text
main:
   li t1, IN_ADDRESS_HEXA_KEYBOARD   # Input address for row assignment
   li t2, OUT_ADDRESS_HEXA_KEYBOARD  # Output address for reading key pressed

start:
   la s0, buffer                  # Điểm bắt đầu của buffer
   la s1, index                   # Địa chỉ lưu chỉ số

loop_main:
   # Display the message to enter the password
   la a0, msg_enter_password
   jal print_message

   # Read and check password
   jal READ_PASSWORD
   jal CHECK_PASSWORD
```

```
    # If password is correct, show LED
    beqz a0, handle_password_incorrect  # Nếu a0 = 0, mật khẩu sai

    la t6, wrong_attempts
    sw zero, 0(t6)                    # Khôi phục số lần nhập sai về 0
    la a0, msg_unlock_success
    jal print_message
    jal LED_ON

    la a0, msg_press_A_to_change
    jal print_message
    j wait_for_A

end_main:
    li a7, 10
    ecall

handle_password_incorrect:
    la t6, wrong_attempts
    lw t4, 0(t6)
    addi t4, t4, 1
    sw t4, 0(t6)

# Nếu đã nhập sai quá 3 lần, tạm ngưng chương trình
    li t6, 3
    bge t4, t6, lock_suspended

    la a0, msg_password_wrong
    jal print_message

    jal LED_OFF
    j loop_main

lock_suspended:
    # Hiển thị thông báo tạm ngưng khóa
    la a0, msg_lock_suspended
    jal print_message

# Tạm dừng mọi nút bấm trong 1 phút (60 giây)
wait_1_minute:
    li a0, 60000                    # Giả lập thời gian đợi 1 phút
    li a7, 32
    ecall

    # Sau 1 phút, khôi phục lại số lần nhập sai
    la t6, wrong_attempts
    sw zero, 0(t6)
    j loop_main
```

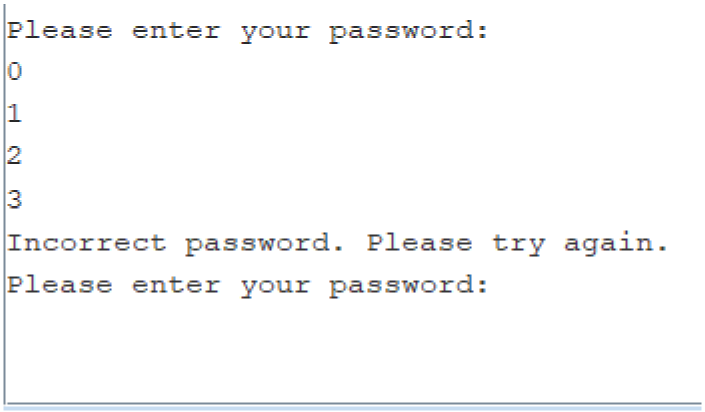
```
print_message:
    addi sp, sp, -4
    sw a0, 0(sp)                    # Lưu giá trị của a0 (địa chỉ chuỗi)
    li a7, 4                        # Syscall: Print string
    ecall                           # In chuỗi
    lw a0, 0(sp)
    addi sp, sp, 4
    jr ra

back_to_program:
    jr ra

# ------------------------------------------------------------
# Function LED_ON: Bật LED (Hiển thị tất cả các segment)
# ------------------------------------------------------------
LED_ON:
    li t0, SEVENSEG_LEFT            # Địa chỉ LED trái
    li a0, 0x7F                     # LED = 8 (Hiển thị tất cả các segment)
    sb a0, 0(t0)                    # Ghi giá trị vào LED trái

    li t0, SEVENSEG_RIGHT           # Địa chỉ LED phải
    li a0, 0x7F                     # LED = 8 (Hiển thị tất cả các segment)
    sb a0, 0(t0)                    # Ghi giá trị vào LED phải

    jr ra                           # Quay lại gọi hàm

# ------------------------------------------------------------
# Function LED_OFF: Tắt LED (Tắt tất cả các segment)
# ------------------------------------------------------------
LED_OFF:
    li t0, SEVENSEG_LEFT            # Địa chỉ LED trái
    li a0, 0x00                     # Tắt LED (Tất cả các segment = 0)
    sb a0, 0(t0)                    # Ghi giá trị vào LED trái

    li t0, SEVENSEG_RIGHT           # Địa chỉ LED phải
    li a0, 0x00                     # Tắt LED (Tất cả các segment = 0)
    sb a0, 0(t0)                    # Ghi giá trị vào LED phải

    jr ra                           # Quay lại gọi hàm

# ------------------------------------------------------------
# Function READ_PASSWORD : Read password input from keypad
# ------------------------------------------------------------
READ_PASSWORD:
    li t5, 0x0
    la s0, buffer
    la s1, index
    sw t5, 0(s1)
loop:
    li t3, 0x1                      # Quét từ row 0x1 (row đầu tiên)
```

```
scan_rows:
    sb t3, 0(t1)                    # Gán giá trị hàng hiện tại
    lb a0, 0(t2)                    # Đọc mã nút bấm
    beqz a0, next_row                   # Không có nút nào được bấm -> kiểm tra hàng tiếp theo

    li t4, 0x88                     # Kiểm tra nếu nút bấm là F
    andi a1, a0, 0xFF
    beq a1, t4, back_to_program       # Nếu nút F được bấm, quay lại luồng chương trình để kiểm tra mật khẩu

    addi sp, sp, -4
    sw ra, 0(sp)
    jal decode
    lw ra, 0(sp)
    addi sp, sp, 4

    # Kiểm tra nếu mã phím không hợp lệ
    li t4, 0xFF
    beq a0, t4, invalid_key           # Nếu a0 = 0xFF, gọi đến nhãn invalid_key

    sb a0, 0(s0)                    # Lưu mã nút bấm vào buffer
    addi s0, s0, 1                  # Tăng địa chỉ buffer
    lw t5, 0(s1)                    # Đọc chỉ số hiện tại
    addi t5, t5, 1                  # Tăng chỉ số
    sw t5, 0(s1)                    # Lưu lại chỉ số mới

    addi sp, sp, -4
    sw ra, 0(sp)
    jal print_number
    lw ra, 0(sp)
    addi sp, sp, 4

    sb zero, 0(t2)

    li a0, 100                      # Sleep 100ms (debounce)
    li a7, 32
    ecall

next_row:
    slli t3, t3, 1                  # Chuyển sang row tiếp theo (0x1 -> 0x2 -> 0x4 -> 0x8)
    li t4, 0x10                     # Hết tất cả các row (sau 0x8)
    blt t3, t4, scan_rows             # Tiếp tục quét nếu còn row
    j loop                          # Quay lại vòng lặp chính

decode:                             # Chuyển đổi số nhập vào từ vị trí nhận được khi nhập
    andi a1, a0, 0xFF

    li t4, 0x11
    li t6, 0x0
    beq a1, t4, save
```

```
   li t4, 0x21
   li t6,0x1
   beq a1, t4, save

   li t4, 0x41
   li t6, 0x2
   beq a1, t4, save

   li t4, 0x81
   li t6, 0x3
   beq a1, t4, save

   li t4, 0x12
   li t6, 0x4
   beq a1, t4, save

   li t4, 0x22
   li t6, 0x5
   beq a1, t4, save

   li t4, 0x42
   li t6, 0x6
   beq a1, t4, save

   li t4, 0x82
   li t6, 0x7
   beq a1, t4, save

   li t4, 0x14
   li t6, 0x8
   beq a1, t4, save

   li t4, 0x24
   li t6, 0x9
   beq a1, t4, save

   # Nếu không khớp bất kỳ phím nào, trả về 0xFF để báo lỗi
   li a0, 0xFF              # Đặt a0 = 0xFF khi không khớp phím nào
   jr ra                    # Quay lại

invalid_key:
   sb zero, 0(t2)
   j loop

save:
   add a0, zero, t6
   jr ra

print_number:
   addi sp, sp, -4
   sw a0, 0(sp)
   li a7, 1                    # Syscall: Print integer
   ecall                       # In giá trị trong a0

   # In ký tự xuống dòng
   li a0, 10                   # Mã ASCII của '\n'
   li a7, 11                   # Syscall: Print character
   ecall

   lw a0, 0(sp)
   addi sp, sp, 4
   jr ra

# ------------------------------------------------------------
# Function CHECK_PASSWORD: Compare entered password with stored password
# ------------------------------------------------------------
CHECK_PASSWORD:
   la s0, buffer               # Bắt đầu đọc buffer
   la s1, index                # Bắt đầu đọc mật khẩu lưu trữ
   la s2, password
   lw t5, 0(s1)

   addi sp, sp, -4
   sw t5, 0(sp)

   la t6, len
   lw t6, 0(t6)                # Đặt chiều dài mật khẩu tối thiểu

   bne t5, t6, password_incorrect    # Nếu số ký tự nhập vào khác số kí tự của mật khẩu -> sai

compare_loop:
   beqz t5, password_correct         # Nếu không còn ký tự, mật khẩu đúng
   lb a0, 0(s0)                # Lấy ký tự từ buffer
   lb a1, 0(s2)                # Lấy ký tự từ password
   bne a0, a1, password_incorrect    # Nếu khác nhau, mật khẩu sai
   addi s0, s0, 1              # Tăng địa chỉ buffer
   addi s2, s2, 1              # Tăng địa chỉ password
   addi t5, t5, -1             # Giảm số lượng ký tự còn lại
   j compare_loop                  # Quay lại so sánh ký tự tiếp theo

password_correct:
   li a0, 1
   j recover

password_incorrect:
   li a0, 0
   j recover

recover:
   lw t5, 0(sp)
   sw t5, 0(s1)
   addi sp, sp, 4
   j back_to_program

# wait for change password
wait_for_A:
   li t3, 0x1                    # Quét từ row 0x1 (row đầu tiên)
scan_rows_A:
   sb t3, 0(t1)                    # Gán giá trị hàng hiện tại
   lb a0, 0(t2)                    # Đọc mã nút bấm
   beqz a0, next_row_A               # Không có nút nào được bấm -> kiểm tra hàng tiếp theo

   li t4, 0x44                    # Mã phím A
   andi a1, a0, 0xFF
   beq a1, t4, change_password        # Nếu phím A được bấm, xử lý đổi mật khẩu

   sb zero, 0(t2)

   li a0, 100                     # Sleep 100ms (debounce)
   li a7, 32
   ecall

next_row_A:
   slli t3, t3, 1                   # Chuyển sang hàng tiếp theo
   li t4, 0x10                    # Hết tất cả các hàng
   blt t3, t4, scan_rows_A           # Tiếp tục quét nếu còn hàng
   j wait_for_A                    # Quay lại quét hàng đầu tiên

change_password:
   # Chuyển đến hàm đổi mật khẩu
   jal LED_OFF
   jal CHANGE_PASSWORD
   j start                        # Quay lại vòng lặp chính

# -----------------------------------------------------------
# Function CHANGE_PASSWORD: Handle changing the password
# -----------------------------------------------------------
CHANGE_PASSWORD:
   li a2, 0                       # Số lần nhập sai mật khẩu
   li a3, 3                                           # Cho phép nhập mật khẩu sai tối đa bao nhiêu lần
   addi sp, sp -4
   sw ra, 0(sp)
retry_old_password:
   # Hiển thị thông báo nhập mật khẩu cũ
   la a0, msg_enter_old_password
   jal print_message

   # Đọc mật khẩu cũ từ người dùng
   jal READ_PASSWORD
   # Kiểm tra mật khẩu cũ
   jal CHECK_PASSWORD
   beqz a0, change_password_wrong     # Nếu a0 = 0, mật khẩu sai

update_new_password:
   # Hiển thị thông báo nhập mật khẩu mới
   la a0, msg_enter_new_password
   jal print_message

   # Đọc mật khẩu mới từ người dùng
   jal READ_PASSWORD

   la s0, buffer                    # Địa chỉ của buffer chứa mật khẩu mới
   la s1, index
   lw t5, 0(s1)                     # Lấy chiều dài mật khẩu mới từ index

   li t6, 4
   blt t5, t6, password_too_short

   # Lưu mật khẩu mới vào vùng nhớ `password`
   la s2, password                  # Địa chỉ của password cũ
   la t6, len
   sw t5, 0(t6)                     # Cập nhật chiều dài mới vào len

copy_new_password:
   beqz t5, password_updated        # Nếu không còn ký tự, cập nhật xong
   lb a0, 0(s0)                     # Lấy từng ký tự từ buffer
   sb a0, 0(s2)                     # Ghi vào vùng nhớ `password`
   addi s0, s0, 1                   # Tăng địa chỉ buffer
   addi s2, s2, 1                   # Tăng địa chỉ password
   addi t5, t5, -1                  # Giảm số lượng ký tự còn lại
   j copy_new_password              # Lặp lại cho ký tự tiếp theo

password_updated:
   # Hiển thị thông báo mật khẩu đã cập nhật thành công
   la a0, msg_password_updated
   jal print_message
   lw ra, 0(sp)
   addi sp, sp, 4
   jr ra

password_too_short:
   la a0, msg_short_password        # Hiển thị thông báo mật khẩu quá ngắn
   jal print_message
   j update_new_password            # Yêu cầu nhập lại mật khẩu mới

change_password_wrong:
   addi a2, a2, 1
   bge a2, a3, lock_user
   la a0, msg_password_wrong
   jal print_message

   j retry_old_password           # Yêu cầu nhập lại

lock_user:
   # Hiển thị thông báo khóa
   la a0, msg_lock_suspended
   jal print_message

   li a0, 60000                   # Giả lập thời gian đợi 1 phút
   li a7, 32
   ecall

   # Sau 1 phút, khôi phục lại số lần nhập sai
   li a2, 0
   j retry_old_password           # Yêu cầu nhập lại sau khi hết khóa
```

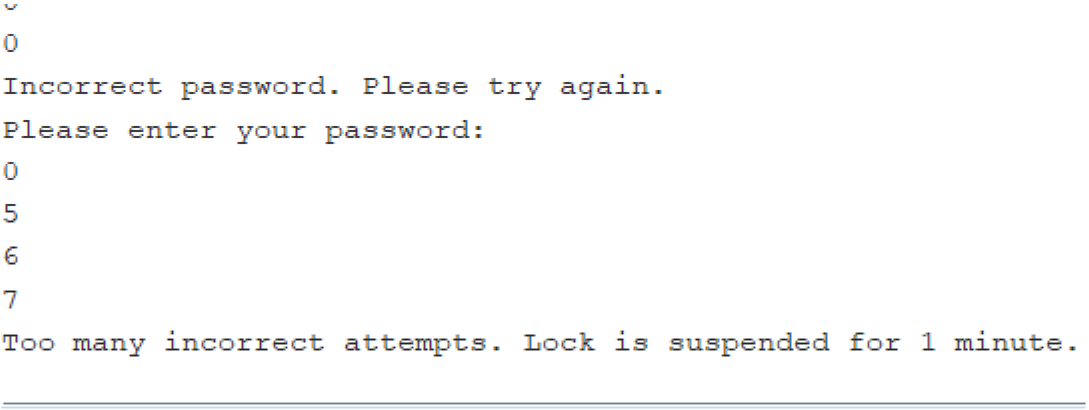
## 4. Result:

- Wrong password:



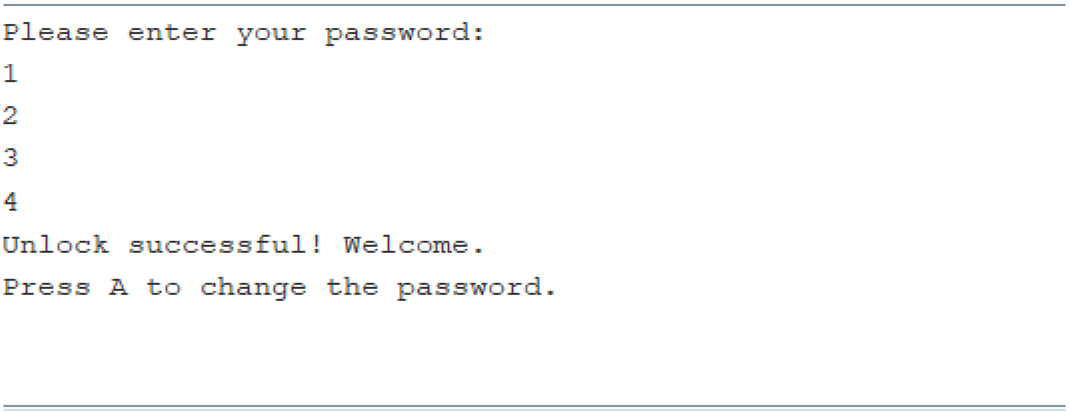
```
Please enter your password:
0
1
2
3
Incorrect password. Please try again.
Please enter your password:
```

- Wrong password 3 times:

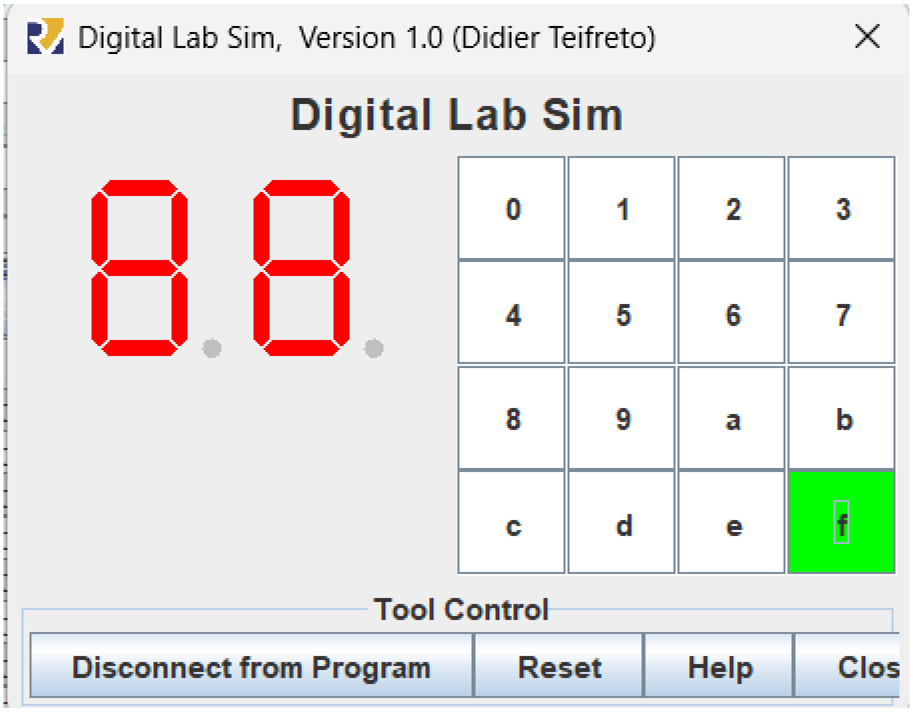


```
0
Incorrect password. Please try again.
Please enter your password:
0
5
6
7
Too many incorrect attempts. Lock is suspended for 1 minute.
```

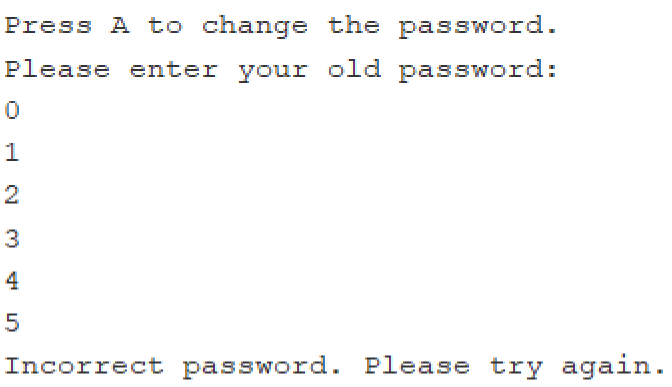
- Correct password:



```
Please enter your password:
1
2
3
4
Unlock successful! Welcome.
Press A to change the password.
```

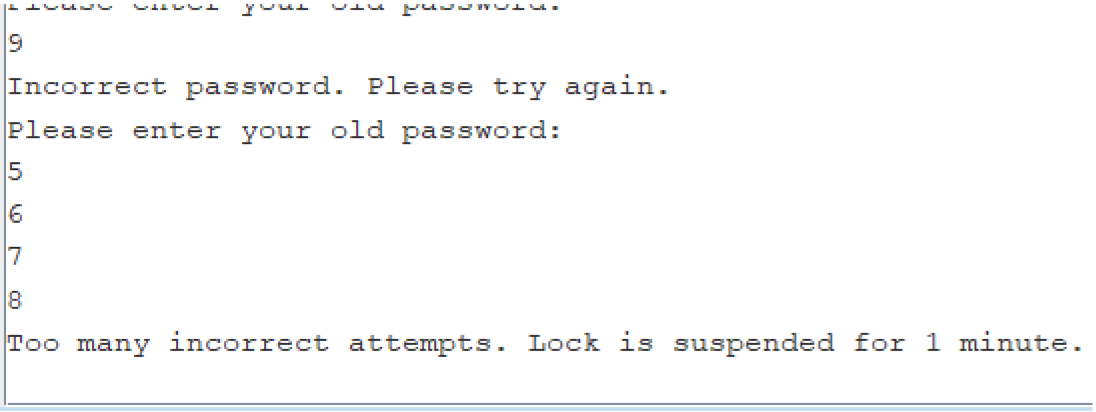


- Change password but enter wrong old password:



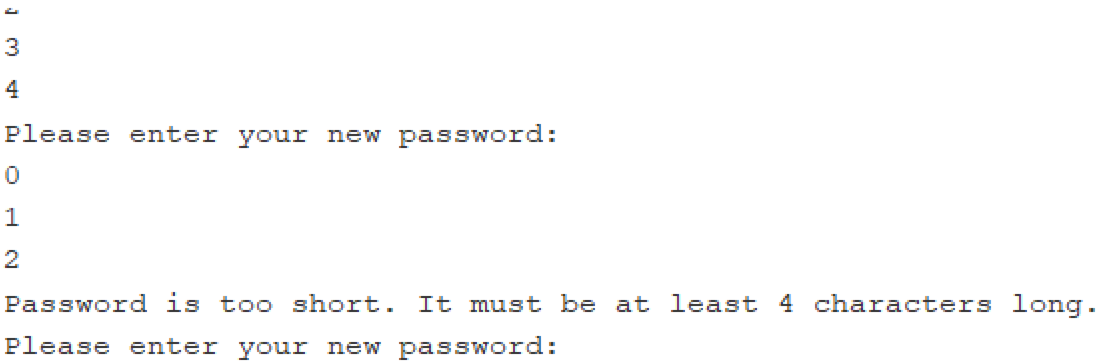
```
Press A to change the password.
Please enter your old password:
0
1
2
3
4
5
Incorrect password. Please try again.
```

- Enter wrong password 3 times:



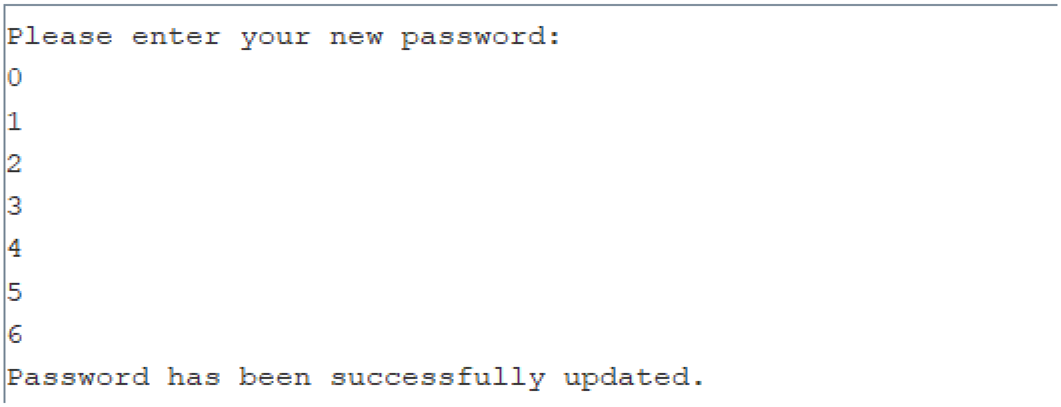
```
Please enter your old password:
9
Incorrect password. Please try again.
Please enter your old password:
5
6
7
8
Too many incorrect attempts. Lock is suspended for 1 minute.
```

- New password has less than 4 characters:



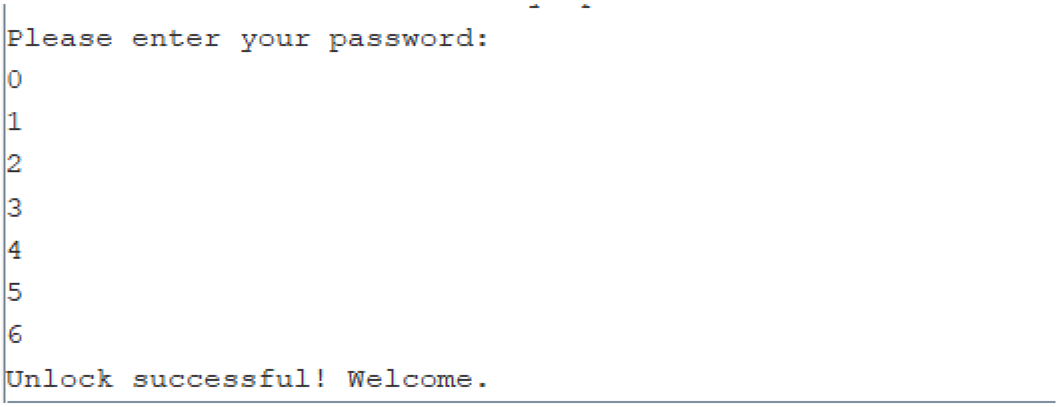
```
2
3
4
Please enter your new password:
0
1
2
Password is too short. It must be at least 4 characters long.
Please enter your new password:
```

- Change password success:

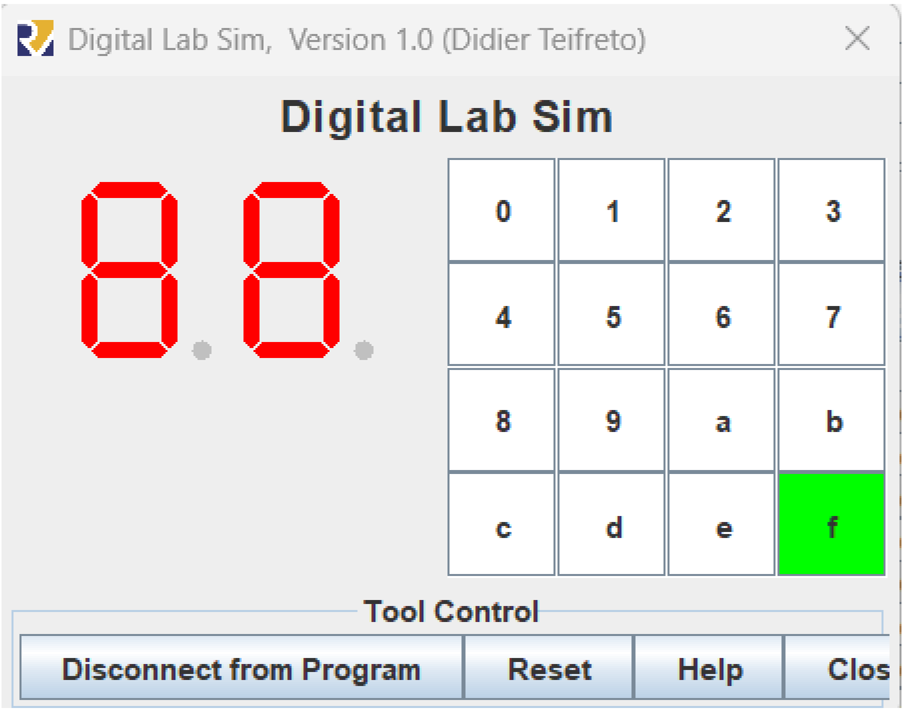


```
Please enter your new password:
0
1
2
3
4
5
6
Password has been successfully updated.
```

- Try new password:



```
Please enter your password:
0
1
2
3
4
5
6
Unlock successful! Welcome.
```


Digital Lab Sim,  Version 1.0 (Didier Teifreto)
Digital Lab Sim
0
1
2
3
4
5
6
7
8
9
a
b
c
d
e
f
Tool Control
Disconnect from Program
Reset
Help
Clos

# **Exercise 5:** Checking the syntax of an instruction

## 1. Requirements:

The processor's compiler checks the syntax of the instructions in the source code, whether they are correct or not, and then translates those instructions into machine code. Write a program that checks the syntax of any instruction (not including pseudo instructions) as follows: - Enter a line of instructions from the keyboard. For example, beq s1, 31, t4 - Check if the opcode is correct or not. In this example, beq is correct so the program should display “opcode: beq, correct” - Check if the operands are correct or not? In this example, s1 is correct, 31 is incorrect, and t4 does not need to be checked anymore. Tip: students should construct structures that can store the format of each instruction with the instruction name and type of operands.

## 2. Work flow

1. User enters a line of instructions from the keyboard. For example, beq s1, 31, t4
2. The code will check if the opcode is correct or not.
3. After that, check if the operands are correct or not.

## 3. Algorithm

This exercise mainly focuses on string processing. Separate the input instructions into 2 parts, opcode and operand, then save them in registers for processing.

Main implementation steps:

1. Function “main”: input value: wait until there is an input value and end if “Enter” is encountered.

2. Function “store_opcode” and “check_opcode”: Separate the input instruction and start checking the opcode. If it matches a valid opcode, it will continue processing.

3. Function “check_operand”: process operands, based on the instruction form to determine the type of operand to be checked.

## 4. Source code

```
#------------------------------------------------------------------------------------------------------------------------
# Steps to run this program:
# Step 1: Select Tools
# Step 2: Choose "Key Board and Display MMIO Simulator
# Step 3: Run the code
# Step 4: Select "connect to the program"
#------------------------------------------------------------------------------------------------------------------------
.eqv KEY_CODE 0xFFFF0004       # ASCII code from keyboard, 1 byte
.eqv KEY_READY 0xFFFF0000      # =1 if has a new keycode ?
                                     # Auto clear after lw
.eqv DISPLAY_CODE 0xFFFF000C   #  ASCII code to show, 1 byte
.eqv DISPLAY_READY 0xFFFF0008  # =1 if the display has

.data
instruction: .space 100
opcode: .space 20
operand: .space 20
base_address: .space 20
format_operand: .byte

# There are maximum 3 operands in a instruction.
# There are 3 types of operand. We will assign operand types as follows:
# no operand:0, integer: 1, register: 2, label: 3, fregister: 4, cregister: 5, dyn: 6, (base_address):7"

# List of instruction with their operand formats
register: .asciz "x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19 x20 x21 x22 x23 x24
x25 x26 x27 x28 x29 x30 x31 zero ra sp gp tp t0 t1 t2 s0 s1 a0 a1 a2 a3 a4 a5 a6 a7 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11
t3 t4 t5 t6 pc !"
aformat: .asciz "add****222 addi***221 and****222 andi***221 auipc**120 beq****223 bge****223
bgeu***223 blt****223 bltu***223 bne****223 div****222 divu***222 jal****230 jalr***221 mul****222
lui****120 mulh***222 mulhsu*222 mulhu**222 or*****222 ori****221 rem****222 remu***222
sll****222 slli***221 slt****222 slti***221 sltiu**221 sltu***222 sra****222 srai***221 srl****222
srli***221 sub****222 xor****222 xori***221 lb*****217 lbu****217 lh*****217 lhu****217 lw*****217
sb*****217 sh*****217 sw*****217 uret***000 wfi****000 ecall**000 ebreak*000 !"

# Float instruction
# There are maximum
fformat: .asciz "fadd.d****44446 fadd.s****44446 fclass.d**24000 fclass.s**24000 fcvt.d.s**44600
fcvt.d.w**42600 fcvt.d.wu*42600 fcvt.s.d**44600 fcvt.s.w**42600 fcvt.s.wu*46000 fcvt.w.d**24600
fcvt.w.s**24600 fcvt.wu.d*24600 fcvt.wu.s*14600 fdiv.d****44446 fdiv.s****44446 fence*****11000
fence.i***00000 feq.d*****24400 feq.s*****24400 fle.d*****24400 fle.s*****24400 flt.d*****24400
flt.s*****24400 fmadd.d***44446 fmadd.s***44446 fmax.d****44400 fmax.s****44400 fmin.d****44400
fmin.s****44400 fmsub.d***44446 fmsub.s***44446 fmul.d****44446 fmul.s****44446 fmv.s.x***42000
fmv.x.s***24000 fnmadd.d**44446 fnmadd.s**44446 fnmsub.d**44446 fnmsub.s**44446 fsgnj.d***44400
fsgnj.s***44400 fsgnjn.d**44400 fsgnjn.s**44400 fsgnjx.d**44400 fsgnjx.s**44400 fsqrt.d***44600
fsqrt.s***44600 fsub.d****44446 fsub.s****44446 fld*******41700 flw*******41700 fsd*******41700
fsw*******41700 !"
```

```
fregister: .asciz "f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 f13 f14 f15 f16 f17 f18 f19 f20 f21 f22 f23 f24 f25 f26 f27 f28 f29 f30 f31 ft0 ft1 ft2 ft3 ft4 ft5 ft6 ft7 fs0 fs1 fa0 fa1 fa2 fa3 fa4 fa5 fa6 fa7 fs2 fs3 fs4 fs5 fs6 fs7 fs8 fs9 fs10 fs11 ft8 ft9 ft10 ft11 !"
rmode: .asciz "rne rtz rdn rup rmm dyn !"

x0_reg: .asciz "x0 zero !"

cformat: .asciz "csrrci*251 csrrsi*251 csrrwi*251 csrrc**252 csrrs**252 csrrw**252 !"
csregister: .asciz "ustatus fflags frm fcsr uie utvec uscratch uepc ucause utval uip cycle time instret cycleh timeh instreth !"

invalid_symbol: .asciz "!@#$%^&*()+={}[]|:;\"'<>,?/~ "

prompt1: .asciz "----------------------------------------------------------------------------------------------------"
prompt2: .asciz "Checking instruction: "
end_prompt: .asciz "Exited"

valid_message1: .asciz "Opcode "
valid_message2: .asciz " is valid."
valid_message3: .asciz " is invalid."
valid_message4: .asciz "Operand "
missing_message: .asciz "Too few arguments "
space: .asciz " "
newline: .asciz "\n"
asterisk: .asciz "*"
stop_mess: .asciz "exit "
.text
# Clear all the content of string to start a new loop
clear:
        la a1, opcode
        jal clear_string

        la a1, instruction
        jal clear_string

        la a1, base_address
        jal clear_string

        la a1, operand
        jal clear_string

        la a0, prompt1
        jal display_string
        jal display_newline
main:
        # Set up the necessary registers
        li  s0, 0xFFFF0004      # Keyboard data register address (KEY_CODE)
        li  s1, 0xFFFF0000      # Keyboard status register address (KEY_READY)
        la  a0, instruction       # Load the address of the buffer to s0 (where we will store the input)
        li t3, 8                   #ascii code for backspace
```

```
WaitForKey:
        # Poll the keyboard status register
        lw t1, 0(s1)            # Load the status of the keyboard
        beq t1, zero, WaitForKey  # If no new key, keep waiting

ReadKey:
        lb t0, 0(s0)            # Load the ASCII value of the pressed key
        jal backspace_handler
        # Skip storing if t0 is cleared (backspace handler sets it to zero)
        beq t0, zero, WaitForKey
        sb t0, 0(a0)            # Store the key in the buffer
        addi a0, a0, 1          # Increment the buffer pointer

        # Check if the Enter key (newline) was pressed to stop input
        li t2, 10               # ASCII for newline (Enter key)
        beq t0, t2, DoneReading   # If Enter is pressed, finish reading

        j WaitForKey            # Continue waiting for next key
# Handle the backspace press, skip \b char and delete the previous char
backspace_handler:
        bne t0, t3, done_backspace
        addi a0, a0, -1
        mv t0, zero
  done_backspace:
        jr ra
DoneReading:
        lb a4, newline
        lb t5, space
        lb t6, asterisk
        la a0, instruction
        la a1, opcode
        li s10, 0

store_opcode:
        add t1, a0, s10                         # shift 1 bit to the left of instruction
        add t2, a1, s10                         # shift 1 bit to the left of string opcode
        lb t3, 0(t1)                            # get the character from instruction
        sb t3, 0(t2)                            # store the character in opcode
        beq t3, a4, done_store_opcode
        beq t3, t5, done_store_opcode

        addi s10, s10, 1
        j store_opcode
done_store_opcode:
        jal compare_exit
prompt_display:
        la a0, prompt1
        jal display_string
        jal display_newline
        la a0, prompt2
        jal display_string
        la a0, instruction
        jal display_string
        jal display_newline

choose_opcode_list:
        la a1, opcode
        li a3, '!'
        lb t0, 0(a1)
        li t1, 'f'
        beq t0, t1, ck_opcode_f
        li t1, 'c'
        beq t0, t1, ck_opcode_c
        j ck_opcode_a


ck_opcode_f:
        li t0, 0
        la a2, fformat
        li s11, 16
        j ck_opcode
ck_opcode_c:
        li t0, 0
        la a2, cformat
        li s11, 11
        j ck_opcode
ck_opcode_a:
        li t0, 0
        la a2, aformat
        li s11, 11

#---------------------------------------------------------------------------------------------------
# Function to ck opcode based on the given opcode list
# Opcode list's address is stored in a2
#---------------------------------------------------------------------------------------------------
ck_opcode:
        la a1, opcode
        li t1, 0                           # i: index of 'format' list
        li t2, 0                           # j : index of opcode
        add t1, t1, t0                             # move to next opcode
        add t0, t0, s11                            # skip format value and asterisks

  # compare each character of opcode and a opcode from the list
  compare:
        add t3, a2, t1                             # t3 is the pointer to current opcode in list 'format'
        lb s0, 0(t3)

        beq s0, a3, invalid_opcode                 # s0 = '!' -> end of format list -> invalid opcode
        add t4, t2, a1                          # t4 is now pointer of opcode
        lb s1, 0(t4)

        beq s0, t6, ck                  # ck if is there any character in opcode if we move to '*' in the list


        bne s0, s1, ck_opcode           # if 2 character is not the same, move to next opcode

        addi t1, t1, 1                          # i++
        addi t2, t2, 1                          # j++
        j compare
 ck:
        bne s1, t5, ck_newline
        j done_opcode
 ck_newline:
        bne s1, a4, ck_opcode           # if the last character is not space nor newline -> wrong opcode
 done_opcode:
        j valid_opcode

invalid_opcode:
        la a0, valid_message1
        jal display_string

        la a0, opcode
        jal display_string

        la a0, valid_message3
        call display_string

        jal display_newline

        j clear
valid_opcode:
        la a0, valid_message1
        jal display_string

        la a0, opcode
        jal display_string

        la a0, valid_message2
        jal display_string

        jal display_newline


#---------------------------------------end of opcode cking-----------------------------------------------------
prep_operand:
        li t3, 0
        li s9, 0                                # flag for parenthesis (cking base address)
        li a5, 0                                # boolean value for cking out of operand  1-> out of
operand
        li a6, 0                                # boolean value for cking out of operand format: 1-> out of
operand format

        la a1, opcode
        lb s11, 0(a1)
        li s4, 'b'
        beq s11, s4, skip_ck_x0                 # skip cking x0 for first operand if the first char of opcode
is b -> branch
        li a7, 1                                # boolean ck for starting the first operand. 1 -> at first
operand. 0 -> other
        j ck_operand
skip_ck_x0:
        li a7, 0
# t1 now stores the index of the first * in format
# a2 now store the format list
# s10 now stores the index of character after the opcode in the instruction
ck_operand:
        la a0, instruction
        la a1, operand
        la a3, format_operand
clear_operand:
        jal clear_string
init_operand_ck:
        li t0, ','
        li s1, ' '
        li t2, '\t'
        li a4, '\n'
        li t6, '*'
        li s8, '('
        li t5, 0
skip_blank:
        addi s10, s10, 1
        add t3, a0, s10
        add t4, a1, t5
        lb s2, 0(t3)
        beq s2, t0, skip_blank
        beq s2, s1, skip_blank
        beq s2, t2, skip_blank
store_operand:
        add t3, a0, s10
        add t4, a1, t5
        lb s2, 0(t3)
        beq s2, a4, done_get_operand                    # stop if newline occurs
        li s7, 1
        beq s2, t0, done_get_operand                    # stop if ',' occurs
        beq s2, s1, done_get_operand                    # stop if space occurs
        beq s2, t2, done_get_operand                    # stop if tab occurs
        beq s9, s7, base_address_read                 # jump to base_address_read function if '('
occured


        beq s2, s8, parenthesis_occur                 # stop if '(' occurs
        addi s10, s10, 1
        addi t5, t5, 1
        beq s2, zero, out_of_operand
        sb s2, 0(t4)
        j store_operand

  parenthesis_occur:
        li s9, 1
  done_get_operand:



store_format_operand:
        addi t1, t1, 1
        add t3, a2, t1
        lb s2, 0(t3)
        beq s2, s1, out_of_format_operand
        li s3, '0'
        beq s2, s3, out_of_format_operand
        bne s2, t6, done_store_format_operand
        j store_format_operand
done_store_format_operand:
        beqz a5, next_fo_ck
        li s3, '6'
        beq s2, s3, clear
        beqz a6, missing_operand
        j clear                                               # if both condition = 1 -> exit
next_fo_ck:

        bnez a6, invalid_operand
        # 1 corresponds to imm
        li s3, '1'
        beq s2, s3, ck_imm

        # 2 corresponds to register
        li s3, '2'
        beq s2, s3, r_ck

        # 3 corresponds to label
        li s3, '3'
        beq s2, s3, ck_label

        # 4 corresponds to label
        li s3, '4'
        beq s2, s3, f_ck
        # 5 corresponds to label
        li s3, '5'
        beq s2, s3, c_ck

        li s3, '6'
        beq s2, s3, ck_dyn

        li s3, '7'
        beq s2, s3,  ck_base_address
r_ck:
        la a4, register
        li s11, 1
        beq a7, s11, cking_x0_register                 # ck if the first char is zero reg or not
        j ck_register
c_ck:
        la a4, csregister
        j ck_register
f_ck:
        la a4, fregister
        j ck_register

ck_register:
        la a1, operand
        lb s5, 0(a1)
        li s2, 'a'
        blt s5, s2, invalid_operand                           #operand must start with an alphabet

        li s2, 0                                       # i = 0
        li s3, 0                                       # j = 0
        li s4, '!'
        li s5, ' '

 loop_ck_register:
        add s8, a1, s2                                        # operand
        add s11, a4, s3                                       # list of registers
        lb s6, 0(s8)
        lb s7, 0(s11)
        addi s2, s2, 1
        addi s3, s3, 1
        beq s7, s5, ck_valid_register                  # reach space -> end of a register in the líst
        beq s7, s4, invalid_operand                           # out of registers in list -> invalid operand
        bne s6, s7, reset_register_op
        j loop_ck_register
 ck_valid_register:
        beqz s6, valid_operand                                # reach null char of operand
 reset_register_op:
        li s2, 0
        j loop_ck_register
```

```
ck_imm:
        la a1, operand
        li s1, 0                                # i = 0
        li t2, '0'
        li t3, 'x'
        li t4, 'X'
        li t5, '9'
        li s11, '\n'
        li s8, '-'
 ck_head:
        add s2, s1, a1
        lb s3, 0(s2)
        beq s3, s8, ck_decimal_head
        bne s3, t2, ck_decimal
 ck_next: # continue cking the imm if it is Oct or Hexa
        addi s1, s1, 1
        add s2, s1, a1
        lb s3, 0(s2)
        beqz s3, valid_operand
        beq s3, t3, ck_hexa                             # ck the head 0x
        beq s3, t3, ck_hexa                             # ck the head 0X
 ck_oct:
        li t3, '7'
        blt s3, t2, invalid_operand
        bgt s3, t3, invalid_operand
        addi s1, s1, 1
        add s2, s2, s1
        lb s3, 0(s2)
        beq s3, zero, valid_operand
        j ck_oct
 ck_hexa:
        addi s1, s1, 1
        add s2, s2, s1
        lb s3, 0(s2)
        beq s3, zero, valid_operand
        li t3, 'A'
        li t4, 'F'
        blt s3, t2, invalid_operand
        bge t5, s3, ck_hexa                     # if '9' >= char >= '0'-> move to next
 ck_hexa_next: # ck if its char is in A between F
        blt s3, t3, invalid_operand
        bgt s3, t4, invalid_operand
        j ck_hexa
 ck_decimal_head:
        addi s1, s1, 1
        add s2, s2, s1
        lb s3, 0(s2)
 ck_decimal:
        blt s3, t2, invalid_operand
        bgt s3, t5, invalid_operand
```

```
            addi s1, s1, 1
            add s2, s2, s1
            lb s3, 0(s2)
            beq s3, zero, valid_operand
            j ck_decimal
ck_label:
            # should not include number or '.' at first
            # cannot include special symbol:  `
            la a1, operand
            lb s3, 0(a1)
            li s2, '.'
            beq s3, s2, invalid_operand                       # if the first char of label is . -> invalid
            li s2, '9'
            bge s2, s3, invalid_operand                       # if the first char of label has ASCII code
<= 9 -> invalid
            li s1, 0                                  # i = 0
            la t2, invalid_symbol
            li s8, ' '
  loop_label:
            add s2, s1, a1
            lb s3, 0(s2)
            beq s3, zero, valid_operand
            addi s1, s1, 1
            li t3, 0                                  # j = 0
   loop_invalid_symbol:                               # loop the invalid_symbol list to ck char[i] in
operand
            add t4, t2, t3
            lb t5, 0(t4)
            beq s3, t4, invalid_operand
            beq t5, s8, loop_label
            addi t3, t3, 1
            j loop_invalid_symbol
ck_base_address:
            li t2, '('
            li t6, ')'
            li t3, '9'
            la a1, operand
            la s2, register                                   # list of register
            lb s3, 0(a1)
            bne s3, t2, invalid_operand                       # if the first char not '(' -> invalid
            li t4, 1                                  # i = 1
            add t5, t4, a1
            lb s3, 0(t5)
            bge t3, s3, invalid_operand                       # if next char is a number -> invalid
            li s4, 0                                  # j = 0
            li s6, '!'
            li s7, ' '
            li t2, ','
            li s11, '\t'
            li t4, 0
  base_address_loop:
            addi t4, t4, 1
            add t5, t4, a1
            lb s3, 0(t5)                                # current char of operand
            add s8, s4, s2
            lb t3, 0(s8)                                # current char of register list
            addi s4, s4, 1
            beq t3, s7, ck_valid_base           # end of a register in list -> ck if the two registers
match
            beq t3, s6, invalid_operand
            bne t3, s3, reset_base
            j base_address_loop
  ck_valid_base:
            beq s3, t6, valid_operand
  reset_base:
            li t4, 0
            j base_address_loop

ck_dyn:
            la a1, operand
            la s2, rmode
            li s1, 0                            # i =0
            li s3, 0                            # j =0
            li s11, '!'
            li t5, ' '
            li t4, 0
  dyn_loop:
            add s4, s1, a1
            add s5, s3, s2
            lb t2, 0(s4)
            lb t3, 0(s5)
            addi s3, s3, 1
            beq t3, t5, ck_valid_dyn
            beq t3, s11, invalid_operand
            bne t2, t3, reset_dyn
            addi s1, s1, 1
            j dyn_loop
  ck_valid_dyn:
            beqz t2, valid_operand
  reset_dyn:
            li s1, 0
            j dyn_loop
missing_operand:
            la a0, missing_message
            jal display_string

            la a0, opcode
            jal display_string

            jal display_newline
		j clear
valid_base_address:
		la a0, valid_message4
		jal display_string

		la a0, base_address
		jal display_string

		la a0, valid_message2
		jal display_string

		jal display_newline

		j ck_operand
valid_operand:
		la a0, valid_message4
		jal display_string

		la a0, operand
		jal display_string

		la a0, valid_message2
		jal display_string

		jal display_newline

		j ck_operand
invalid_operand:
		la a0, valid_message4
		jal display_string

		la a0, operand
		jal display_string

		la a0, valid_message3
		jal display_string

		jal display_newline

		j clear
#---------------------
# s2 store the char
# t3 store the address
# s10 store the index
# t5 is the index of operand. start from 0
# t4 store current pointer of operand
#---------------------
base_address_read:
		li s9, 0
		la a1, operand
```

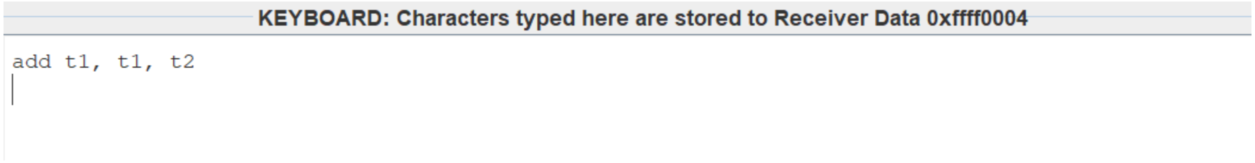
```
            la a0, instruction

            li s11, '('
            li t5, 0
            add t4, t5, a1
            sb s11, 0(t4)

            addi t5, t5, 1
            li s3, ' '
            li s4, ','
            li s5, '\t'
            li s6, ')'
 loop_ba_read:
            add t3, a0, s10
            addi s10, s10, 1
            lb s2, 0(t3)
            beqz s2, invalid_operand                # if ')' didn't occur -> invalid operand
            beq s2, s3, loop_ba_read                # skip the space
            beq s2, s4, loop_ba_read                # skip ','
            beq s2, s5, loop_ba_read                # skip tab


            add t4, t5, a1
            sb s2, 0(t4)
            beq s2, s6, done_get_operand                        # if ')' occured stop reading
            addi t5, t5, 1
            j loop_ba_read
#-----------------------------------------------------------------------
# Function print out a null - terminated string
# param[in] = a0 (address of string)
#-----------------------------------------------------------------------
display_string:
            li s4, '\n'
            li s5, ' '
            li s0, DISPLAY_CODE
            li s1, DISPLAY_READY
            lb s8, 0(a0)
            beqz s8, done_display_string
            beq s8, s4, done_display_string
  WaitForDis:
            lw s6, 0(s1)
            beq s6, zero, WaitForDis
  Showkey:
            sb s8, 0(s0)
            addi a0, a0, 1
            j display_string
  done_display_string:
            jr ra

#-----------------------------------------------------------------------
```

```
# Function to print out a newline to the Simulator
#----------------------------------------------------------------------
display_newline:
            li s0, DISPLAY_CODE
            li s1, DISPLAY_READY
            lb s8, newline
  WaitForDis1:
            lw s6, 0(s1)
            beq s6, zero, WaitForDis
  Showkey1:
            sb s8, 0(s0)
  done_display_newline:
            jr ra

# setting boolean to ck redundant/missing operand
out_of_format_operand:
            li a6, 1
            j done_store_format_operand
out_of_operand:
            li a5, 1                                # boolean value for cking out of operand
            j done_get_operand

# cking the first operand is zero or not
cking_x0_register:
            li a7, 0
            la a0, operand
            la a1, x0_reg
            li, s2, 0                               # i = 0
            li s3, 0                                # j = 0
            li t4, ' '
            li t5, '!'
  cking_x0_loop:
            add s4, a0, s2
            add s5, a1, s3
            lb t2, 0(s4)
            lb t3, 0(s5)
            addi s3, s3, 1
            beq t3, t4, ck_valid_x0                 # ck if the operand and current register are completely
matched
            beq t3, t5, ck_register                 # if t3 = !, which means end of the list, operand is neither
x0 nor zero -> continue
            bne t2, t3, reset_x0_reg                # reset the index of the operand once the characters are
not matched
            addi s2, s2, 1
            j cking_x0_loop
  ck_valid_x0:
            beqz t2, invalid_operand                # if t2 = 0, which means operand is either x0 or zero ->
invalid
  reset_x0_reg:
            li s2, 0
```

```
        j cking_x0_loop

#----------------------------------------------------------------------
# Function to clear the content of a string
# Param[in]: a1 - address of the string
#----------------------------------------------------------------------
clear_string:
        li s11, 0                                   # i = 0
  loop_clear_string:
        add s4, s11, a1
        lb s5, 0(s4)
        beqz s5, done_clear_string
        sb zero, 0(s4)
        addi s11, s11, 1
        j loop_clear_string
done_clear_string:
        jr ra

#----------------------------------------------------------------------
# Function that compares the opcode with "exit" to terminate the program
#----------------------------------------------------------------------
compare_exit:
        la s1, opcode
        la s2, stop_mess
        li s11, ' '
        li s5, '\n'
  compare_exit_loop:
        lb s3, 0(s1)
        lb s4, 0(s2)
        beq s4, s11, ck_valid_exit
        bne s3, s4, done_compare_exit
        addi s1, s1, 1
        addi s2, s2, 1
        j compare_exit_loop
  ck_valid_exit:
        beq s3, s11, exit
        beq s3, s5, exit
done_compare_exit:
        jr ra
exit:
        la a0, end_prompt
        jal display_string
        jal display_newline

        la a0, prompt1
        jal display_string
```

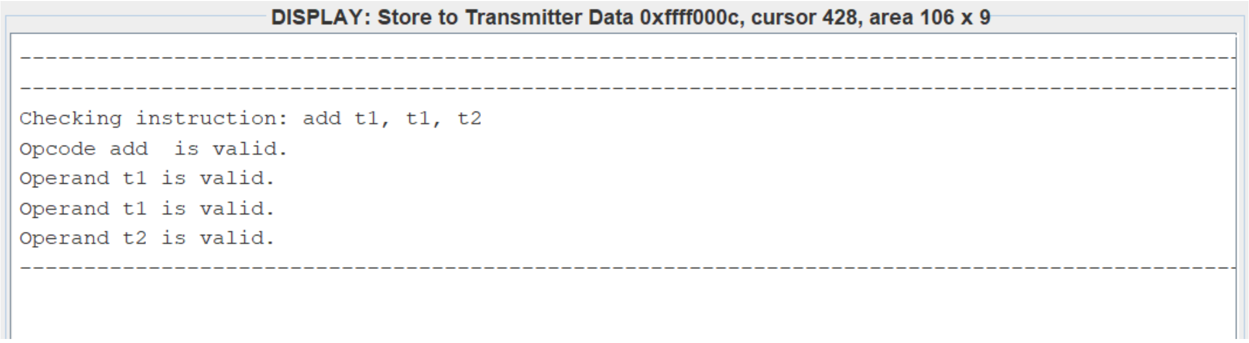
## 5. Result

## 1. List of correct instructions



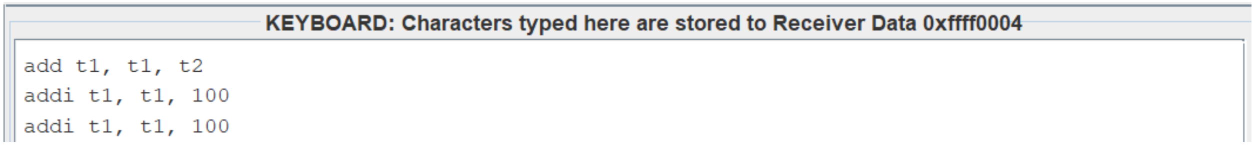
KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004

```
add t1, t1, t2
```



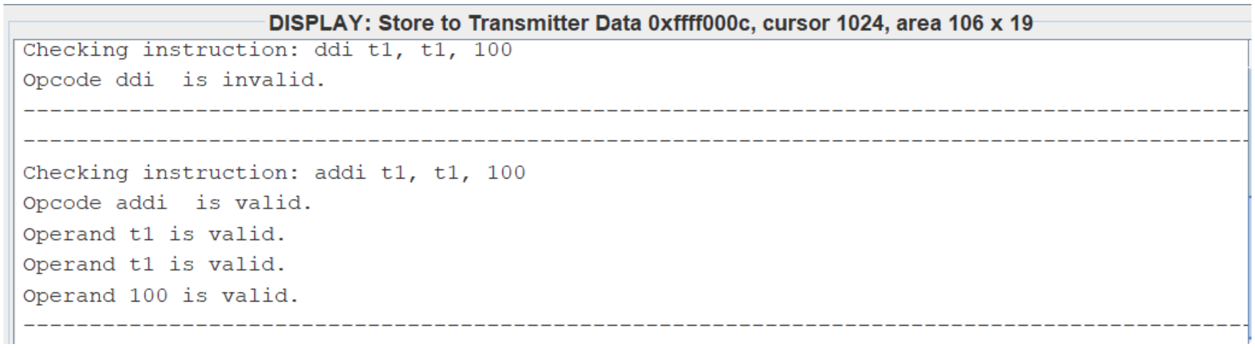
DISPLAY: Store to Transmitter Data 0xffff000c, cursor 428, area 106 x 9

```
----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------
Checking instruction: add t1, t1, t2
Opcode add  is valid.
Operand t1 is valid.
Operand t1 is valid.
Operand t2 is valid.
----------------------------------------------------------------------------------------------
```



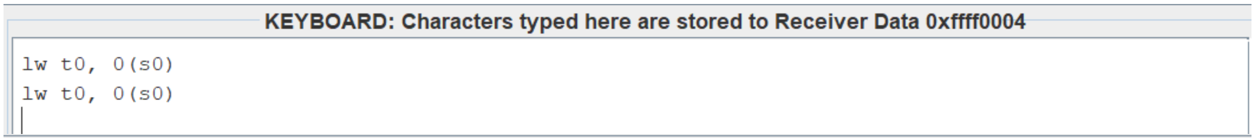
KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004

```
add t1, t1, t2
addi t1, t1, 100
addi t1, t1, 100
```



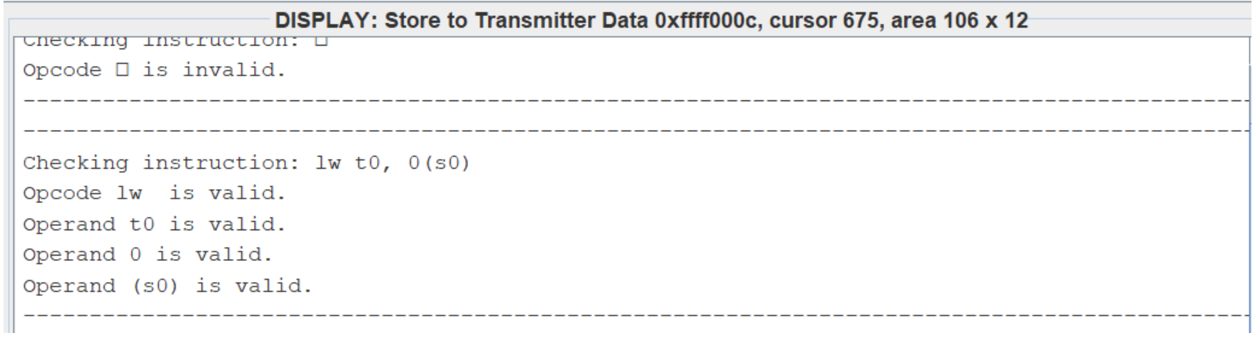
DISPLAY: Store to Transmitter Data 0xffff000c, cursor 1024, area 106 x 19

```
Checking instruction: ddi t1, t1, 100
Opcode ddi  is invalid.
----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------
Checking instruction: addi t1, t1, 100
Opcode addi  is valid.
Operand t1 is valid.
Operand t1 is valid.
Operand 100 is valid.
----------------------------------------------------------------------------------------------
```



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

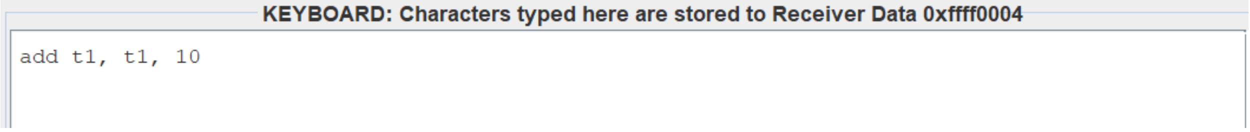
```
lw t0, 0(s0)
lw t0, 0(s0)
```



DISPLAY: Store to Transmitter Data 0xffff000c, cursor 675, area 106 x 12

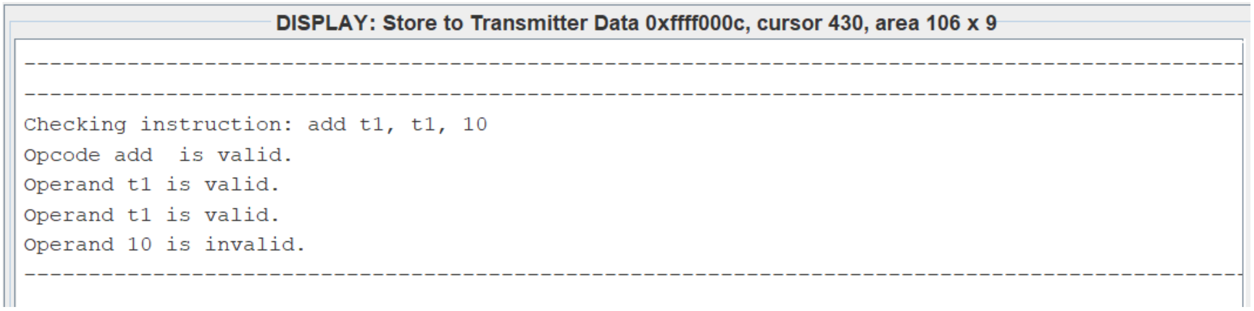
```
Checking instruction: □
Opcode □ is invalid.
----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------
Checking instruction: lw t0, 0(s0)
Opcode lw  is valid.
Operand t0 is valid.
Operand 0 is valid.
Operand (s0) is valid.
----------------------------------------------------------------------------------------------
```

## 2. List of incorrect instructions



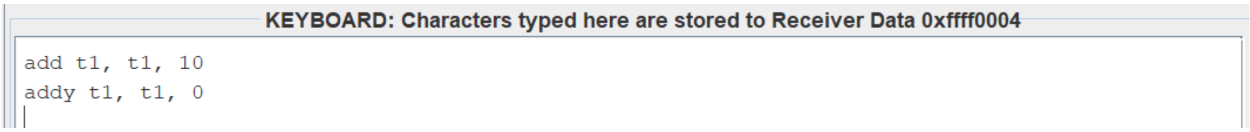
KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004

```
add t1, t1, 10
```



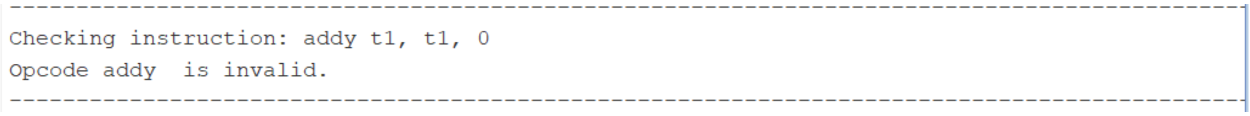
DISPLAY: Store to Transmitter Data 0xffff000c, cursor 430, area 106 x 9

```
----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------
Checking instruction: add t1, t1, 10
Opcode add  is valid.
Operand t1 is valid.
Operand t1 is valid.
Operand 10 is invalid.
----------------------------------------------------------------------------------------------
```



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

```
add t1, t1, 10
addy t1, t1, 0
```



```
----------------------------------------------------------------------------------------------
Checking instruction: addy t1, t1, 0
Opcode addy  is invalid.
----------------------------------------------------------------------------------------------
```