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

Final Project Report

IT3280E - Assembly Language and Computer Architecture Lab

Lecturer: Le Ba Vui

Students:

Name	**Student ID**
Nguyen Trung Kien	20226052
Le Van Hau	20226038

Ha Noi, 2024

Part I: Check the syntax of an instruction (Assignment 5)

1. Problem definition

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 include of 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, t4 doesn't need to check anymore.

Tip: students should construct structures that can store the format of each instruction with the instruction name and type of operands

2. Source code

```
.data
# ------------------------------------------------------------------------------- #
# Opcode library
                        #
# Rule: each opcode has length of 8 byte, seperated by type and syntax        #
# ------------------------------------------------------------------------------- #

# Opcode Library:
opcodeLibrary:  .asciz "add,1  sub,1  addu,1 subu,1 mul,1  and,1  or,1   rem,1  xor,1
slt,1  xori,2 ori,2  srai,2 slli,2 slri,2 addi,2 addiu,2 andi,2 sll,2  srl,2  div,1  mv,3
lw,4   sw,4   lb,4   sb,4   lbu,4  lhu,4  ll,4   sh,4   lui,5  li,5   la,6   jr,7   beq,8  bne,8
blt,8  bge,8  j,9    jal,9  "

buffer:.space 100
opcode:     .space 10

# Message
input_message:      .asciz  "Enter string: "
correct_opcode_prompt:.asciz "\nCorrect opcode: "
end_prompt: .asciz "\nCorrect syntax."
not_valid_register_prompt: .asciz  "\nInvalid register syntax."
not_valid_number_prompt: .asciz "\nNot valid number."
not_valid_address_prompt: .asciz "\nNot valid address"
```

```
valid_syntax_prompt: .asciz "\nCorrect RISCV syntax."
continue_prompt: .asciz "\nContinue? (1. Yes 0. No): "
missing_prompt: .asciz "\nMissing operand"
newline: .asciz "\n"
# Syntax error mess:
missing_comma_prompt: .asciz "\nSyntax error: missing colon."
invalid_opcode_prompt: .asciz "\nOpcode is invalid or doesn't exist."
too_many_variable_prompt: .asciz "\nSyntax has too many variables."


# Registers library #
# each register has 8 bytes in registLibrary
registerLibrary: .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      "
# s0 is the address of input string
# t0 is used for traversing input string
# s1 is the address of opcode
# a1 is used for traversing opcode
# s2 is the address of opcodeLibrary
# a2 is used for traversing opcodeLibrary
# s3 is the address of registerLibrary
# a3 is used for traversing registerLibrary
.text
main:
   li a7, 4
   la a0, input_message              # Print input message
   ecall
read_data:
        li a7, 8
        la a0, buffer                                    # Store input data in buffer
        li a1, 100
        ecall
   mv s0, a0                       # Store adress of input string into s0

   jal clear_whitespace                             # Jump to clear white space

read_opcode:
        la a1, opcode                                    # a1 is used for incrementing
opcode character position
        la s1, opcode                                    # s1 = address of opcode
        mv t0, a0                                                # t0 = a0
loop_read_opcode:
```

```
        lb t1, 0(t0)                          # t1 = current character in opcode
        li s11, ' '                               # s11 = temp = ' '
        beq t1, s11, check_opcode         # if a whitespace is found then check the
opcode
        li s11, '\n'                          # s11 = temp = '\n'
        beq t1, s11, missing_                 # if a newline character is found
then the string is missing operands
        sb t1, 0(a1)                          # store current character into opcode
        addi t0, t0, 1                        # continue check the next char
        addi a1, a1, 1                        # increment current address of
opcode

        j loop_read_opcode

#Check opcode
check_opcode:
        mv a1, s1                                 # a1 = s1 = address of opcode
        mv s0, t0                                 # s0 points to the character
after opcode
        la s2, opcodeLibrary              # s2 = address of opcode library
        jal check
        j invalid_opcode                      # Jump to invalid opcode

check:
        mv a2, s2                                 # a2 points to the beginning
of library
loop_check:
        lb t2, 0(a2)                          # load each character from library
        li s11, ','                               # s11 = temp = ' '
        beq t2, s11, evaluation1          # if meet colon, evaluate whether it is
correct
        lb t1, 0(a1)                          # load each character in input
opcode
        beq t2, zero, jump_                   # if current character in opcode is \0
then we checked all possible opcodes in the library -> no valid input opcode
        bne t1, t2, next_opcode               # mismatch
        addi a1, a1, 1                        # next char
        addi a2, a2, 1
        j loop_check

evaluation1:
        lb t1, 0(a1)                          # load current character of opcode
        beq t1, zero, opcode_done         # if current character of opcode is null then
it has matched an opcode in opcode library
        j next_opcode                             # else continue checking
```

```
opcode in opcodeLibrary

next_opcode:
        addi s2, s2, 8                          # increment s2 by 8 because each
opcode has 8 bytes in opcode library
        mv a2, s2                                   # udpate a2
        mv a1, s1                                   # reset running for opcode
        j loop_check

opcode_done:
        jal correct_opcode                      # print correct opcode
        addi a2, a2, 1

        lb t2, 0(a2)                            # load synax type in t2
        jal clear_whitespace                # point to s0 to next vailid character after
opcode
        addi t2, t2, -48                        # minus value of t2 by 48 to get the
integer value
        li s11, 1                                   # s11 = temp = 1
        beq t2, s11, Type_1
        li s11, 2                                   # s11 = temp = 2
        beq t2, s11, Type_2
        li s11, 3                                   # s11 = temp = 3
        beq t2, s11, Type_3
        li s11, 4                                   # s11 = temp = 4
        beq t2, s11, Type_4
        li s11, 5                                   # s11 = temp = 5
        beq t2, s11, Type_5
        li s11, 6                                   # s11 = temp = 6
        beq t2, s11, Type_6
        li s11, 7                                   # s11 = temp = 7
        beq t2, s11, Type_7
        li s11, 8                                   # s11 = temp = 8
        beq t2, s11, Type_8
        li s11, 9                                   # s11 = temp = 9
        beq t2, s11, Type_9
end:
        j ending                                    # jump to ending
# clear whitespace until the first valid character
clear_whitespace:
        mv      t0, s0                              # load t0 as the address of
input string
        lb t1, 0(t0)                            # read the first char
        li s11, ' '                                 # s11 = temp = ' '
        beq t1, s11, loop_whitespace        # if the first char is a whitespace then
```

```
delete
        li s11, 9                                   # s11 = temp = tab character
        beq t1, s11, loop_whitespace            # if first char is a tab character then delete
        jr ra                                       # return when the first char is
neither a whitespace or a tab char
loop_whitespace:
        lb t1, 0(t0)                            # read current character
        li s11, ' '                                 # s11 = temp = ' '
        beq t1, s11, whitespace_found           # if the first char is a whitespace then
increment address
        li s11, 9                                   # s11 = temp = tab character
        beq t1, s11, whitespace_found           # if first char is a tab character then
increment address
        mv s0, t0                                   # there is no more invalid
char then update s0
        jr ra
whitespace_found:
        addi t0, t0, 1                          # increment address of input string
by 1 to delete invalid char
        j loop_whitespace                       # continue whitespace loop

# check if current character is a comma
check_comma:
        mv t0, s0                                   # update t0 = s0
        lb t1, 0(t0)                            # get the current char
        li s11, ','                                 # s11 = temp = ','
        bne t1, s11, missing_comma              # if current char != ',' then invalid
syntax
        jr ra

# check gap in instruction and check for comma
check_gap:
        addi sp, sp, -4
        sw ra, 0(sp)                            # store ra
        jal clear_whitespace
        jal check_comma
        addi t0, t0, 1                          # point to char/whitespace after
colon
        mv s0, t0                                   # update s0 point the the next
char
        jal clear_whitespace
        lw ra, 0(sp)
        addi sp, sp, 4
        jr ra
```

```
jump_:
        jr ra

# All types of instructions
# ---------------------------------------------------------------------------- #
OPCODE_TYPES:
Type_1:
# ---------------------------------------------------------------------------- #
#      Format: xyz x1, x2, x3                                          #
# ---------------------------------------------------------------------------- #
      jal reg_check

      jal check_gap

      jal reg_check

      jal check_gap

      jal reg_check

      jal check_end

Type_2:
# ---------------------------------------------------------------------------- #
#      Format: xyz x1, x2, 10000                                       #
# ---------------------------------------------------------------------------- #
      jal reg_check

      jal check_gap

      jal reg_check

      jal check_gap

      jal num_check

      jal check_end

Type_3:
# ---------------------------------------------------------------------------- #
#      Format: mult x2,x3                                              #
# ---------------------------------------------------------------------------- #
      jal reg_check

      jal check_gap
```

```
    jal reg_check

    jal check_end

Type_4:
# ------------------------------------------------------------------------------ #
#      Format: lw x1, 100(x2)                                          #
# ------------------------------------------------------------------------------ #
    jal reg_check

    jal check_gap

    jal address_check

    jal check_end

Type_5:
# ------------------------------------------------------------------------------ #
#      Format: li x1, 100                                          #
# ------------------------------------------------------------------------------ #
    jal reg_check

    jal check_gap

    jal num_check

    jal check_end

Type_6:
# ------------------------------------------------------------------------------ #
#      Format: la x1, label                                          #
# ------------------------------------------------------------------------------ #
    jal reg_check

    jal check_gap

    jal label_check
    li s11, 1                           # s11 = 1
    beq s7, s11, check_end              # case label is character and syntax is
correct

    jal num_check                       # case label is numerical value

    jal check_end
```

```
Type_7:
# ---------------------------------------------------------------------------- #
#       Format xyz x2                                              #
# ---------------------------------------------------------------------------- #
        jal reg_check

        jal check_end

Type_8:
# ---------------------------------------------------------------------------- #
#       Format: beq x1, x2, label                          #
# ---------------------------------------------------------------------------- #
    jal reg_check
    jal check_gap
    jal reg_check
    jal check_gap
    jal label_check
    li s11, 1                                           # s11 = 1
    beq s7, s11, check_end                        # case label is character and syntax is
correct
           jal num_check                                          # case label is numerical
value
           jal check_end
Type_9:
# ---------------------------------------------------------------------------- #
#       Format j 1000 ; j = label                                  #
# ---------------------------------------------------------------------------- #
           jal label_check
           li s11, 1                                              # s11 = 1
           beq s7, s11, check_end
           jal num_check
           jal check_end
# End of instruction types
# ---------------------------------------------------------------------------- #

# All syntax checking functions:
# ---------------------------------------------------------------------------- #
# check whether input string has ended or not
check_end:
           jal clear_whitespace
           lb t5, 0(s0)
           li s11, '\n'                                # s11 = temp = '\n'
           beq t5, s11, valid_syntax
           li s11, '\0'                                # s11 = temp = '\0'
```

```
        beq t5, s11, valid_syntax
        li s11, '#'                                     # s11 = temp = '\#'
        beq t5, s11, valid_syntax
        j too_many_variable                         # not valid

# Check whether string is register or not
reg_check:
        la s3, registerLibrary
        mv a3, s3                               # a3 points to the beginning of register
library
        mv t0, s0                               # t0 points to the current char

loop_reg_check:
        lb t3, 0(a3)                                # load each character from lib
        lb t4, 0(t0)                                # load each character from input
string
        li s11, ' '                                     # s11 = temp = ' '
        beq t3, s11, evaluation2                # if reach space, evaluate whether it is
correct or not
        beq t3, zero, not_valid_register      # if reach '\0' then we have check every register
inside registerLib
        bne t4, t3, next_reg                    # character mismatch
        addi t0, t0, 1                              # next char
        addi a3, a3, 1
        j loop_reg_check
evaluation2:
        lb t4, 0(t0)
        li s11, ','                                     # s11 = ','
        beq t4, s11, found_reg
        li s11, ' '                                     # s11 = ' '
        beq t4, s11, found_reg
        beq t4, zero, found_reg
        li s11, '\n'                                # s11 = '\n'
        beq t4, s11, found_reg
        j next_reg                                      # jump to next register
next_reg:
        addi s3, s3, 8                              # move to next register
        mv a3, s3
        mv t0, s0
        j loop_reg_check
found_reg:
        mv s0, t0                                       # update pointer forward
        j jump_                                             # jump to jump_

# check whether current parameter is a valid number
```

```
num_check:
        mv t0, s0
num_check_loop:
        lb t4, 0(t0)
        li s11, '.'                             # s11 = ','
        beq t4, s11, is_num                     # end of parameter
        li s11, ' '                             # s11 = ' '
        beq t4, s11, is_num                     # end of parameter
        beq t4, zero, is_num                    # end of parameter
        li s11, '\n'                            # s11 = '\n'
        beq t4, s11, is_num                     # end of parameter
        li s11, '9'                             # s11 = '9'
        bgt t4, s11, not_num                    # if t4 > '9' then not num
        li s11, '0'                             # s11 = '0'
        blt t4, s11, not_num                    # if t4 < '0' then not num
        addi t0, t0, 1                          # continue checking
        j num_check_loop
is_num:
        mv s0, t0
        j jump_                                         # jump back
not_num:
        j not_num_error                         # jump to not_num_error

# check whether address syntax is correct
address_check:
adnum_check:
num_check_loop2:
        lb t4, 0(t0)                            # load char
        li s11, '('                             # s11 = temp = '('
        beq t4, s11, is_num2
        li s11, '9'                             # s11 = '9'
        bgt t4, s11, not_num2
        li s11, '0'                             # s11 = '0'
        blt t4, s11, not_num2
        addi t0, t0, 1                          # next char
        j num_check_loop2                       # continue checking next char

is_num2:
        mv s0, t0
        j adreg_check                           # continue check for second
register
not_num2:
        j not_valid_address

# check whether register in address is correct
```

```
adreg_check:
reg_check2:
        addi t0, t0, 1
        mv s0, t0
        la s3, registerLibrary
        mv a3, s3                                   # a3 points to the beginning
of register lib
        mv t0, s0
loop_reg_check2:
        lb t3, 0(a3)                          # load char from registerLb
        lb t4, 0(t0)                          # load char from input string
        li s11, ' '                                 # s11 = ' '
        beq t3, s11, evaluation3          # if reach space, evaluation whether it
correct
        beq t3, zero, not_valid_address2   # if reach \0 then we have checked all available
register
        bne    t4, t3, next_reg2              # if mismatch go  to the next reg
        addi t0, t0, 1                        # next char
        addi a3, a3, 1
        j loop_reg_check2
evaluation3:
        lb t4, 0(t0)
        li s11, ')'                                 # s11 = ')'
        beq t4, s11, found_reg2               # correct syntax
        j next_reg2                                 # else continue checking for
next register
next_reg2:
        addi s3, s3, 8                        # move to the next register in
registerLib
        mv a3, s3
        mv t0, s0
        j loop_reg_check2
not_valid_address2:
        j not_valid_address
found_reg2:
        addi t0, t0, 1
        mv s0, t0                                   # move pointers forward
        jr ra                                       # jump back

# check whether label syntax is correct (for characters)
# ------------------------------------------------------------------------------ #
# output: s7 = 1 if it is character and syntax is correct
#       s7 = 0 if it not character and to signal that input label could be in numerical values
# ------------------------------------------------------------------------------ #
label_check:
```

```
        mv t0, s0
first_char_check:                                   # Can't be number and can't
be underscore:
   lb t4, 0(t0)                                     # get current character of
input string
   li s11, 'a'                                      # s11 = 'a'
   blt t4, s11, not_lower                   # if less than 'a' then it is not lower case
character
   li s11, 'z'                                      # s11 = 'z'
   bgt t4, s11, not_lower                       # if greater than 'z' then it is not
lower case chracter
   j loop_label_check                               # it's lower so we jump to
2nd character
not_lower:
        li s11, 'A'                                     # s11 = 'A'
   blt t4, s11, fail_case                   # if less than 'A' then not alphabet
        li s11, 'Z'                                     # s11 = 'Z'
        bgt t4, s11, fail_case                  # if greater than 'Z' then not alphabet

loop_label_check:                                   # Can be alphabet, number
and underscore
   addi t0, t0, 1                       # increment $a0 by 1 to get next character
   lb t4, (t0)                                  # load current character of input string
        li s11, ' '
   beq t4, s11, valid_label             # if we are here then all preceeding charactes are
valid
   li s11, '\n'                                 # s11 = '\n'
   beq t4, s11, valid_label             # if we are here then all preceeding charactes are
valid
        beq t4, zero, valid_label               # if we are here then all preceeding
charactes are valid
   li s11, 'a'                                      # s11 = 'a'
   blt t4, s11, not_lower2                  # if less than a then it is not lower case
character
   li s11, 'z'
   bgt t4, s11, not_lower2                      # if greater than z then it is not
lower case character
   j loop_label_check                               # else valid, continue to
check for next character

not_lower2:
        li s11, '_'                                     # s11 = '_'
        bne t4, s11, not_underscore             # if it is not underscore then continue
checking
   j loop_label_check                               # else valid, continue to
```

```
check for next character

not_underscore:
        li s11, 'A'                                         # s11 = 'A'
   blt t4, s11, not_upper2                   # If less than 'A' then it is not alphabet
        li s11, 'Z'                                         # s11 = 'Z'
        bgt t4, s11, not_upper2                       # If greater than 'Z' then it is
not alphabet
        j loop_label_check                             # else valid, continue to
check for next character

not_upper2:
        li s11, '0'                                         # s11 = '0'
   blt t4, s11, fail_case                    # if less than 0 then it is not number either
        li s11, '9'                                         # s11 = '9'
        bgt t4, zero, fail_case                       # if greater than 9 then it is
not number either, failcase
        j loop_label_check                             # else valid, continue to
check for next character

fail_case:
   mv t0, s0                                 # reset to before so we check other case
(not using label as address but numerical value instead)
        li s7, 0                                  # set $s7 = 0 to signal to check for
numerical value
   jr ra                                                 # jump back

valid_label:
        mv s0, t0                                 # Move pointer forward
        li s7, 1                                  # if label is all characters and correct then
set $s7 = 1
   jr ra

# End of syntax checking functions
# --------------------------------------------------------------------------- #

# print correct_opcode_prompt and input opcode
# --------------------------------------------------------------------------- #
correct_opcode:
        la a0, correct_opcode_prompt
   li a7, 4
   ecall
   la a0, opcode
   li a7, 4
   ecall
```

```
    li a7, 4
    la a0, newline
    ecall
    mv t0, s0                                  #  Return t0
    jr ra
# ------------------------------------------------------------------------- #


# All types of error messages when checking syntax:
# ------------------------------------------------------------------------- #
missing_comma:
        la a0, missing_comma_prompt
        li a7, 4
        ecall
        j ending

invalid_opcode:
        la a0, invalid_opcode_prompt
        li a7, 4
        ecall
        j ending

too_many_variable:
        la a0, too_many_variable_prompt
        li a7, 4
        ecall
        j ending

not_valid_register:
        la a0, not_valid_register_prompt
        li a7, 4
        ecall
        j ending

not_num_error:
        la a0, not_valid_number_prompt
        li a7, 4
        ecall
        j ending

not_valid_address:
        la a0, not_valid_address_prompt
        li a7, 4
        ecall
        j ending
```

```
missing_:
        la a0, missing_prompt
        li a7, 4
        ecall
        j ending

# End of error types
# --------------------------------------------------------------------------- #

# Print valid syntax
valid_syntax:
        la a0, valid_syntax_prompt
        li a7, 4
        ecall
        j ending

ending:
        la a0, continue_prompt
        li a7, 4
        ecall

        li a7, 5
        ecall
                li s11, 1                                   # s11 = 1
        beq a0, s11, resetAll_andContinue           # if user choose to continue
        # else end program
        li a7, 10
        ecall

# Reset function
resetAll_andContinue:
          li a7, 0
          jal clean_block                               # jump to clean_block
   jal clean_opcode                       # jump to clean_block
   li a0, 0
   li a1, 0
          li a2, 0
          li a3, 0
          li t0, 0
          li t1, 0
          li t2, 0
          li t3, 0
          li t4, 0
          li t5, 0
```

```
        li t6, 0
        li s0, 0
        li s1, 0
        li s2, 0
        li s3, 0
        li s4, 0
        li s5, 0
        li s6, 0
    li s7, 0
    li s11, 0
        j main


# reset all values stored in previous input string to 0
# ---------------------------------------------------------------------------- #
clean_block:
        li t0, 0
        li a1, 0
        la s0, buffer                   # point $s0 to the address of buffer
loop_block:
            li s11, 100                         # s11 = 100
        beq a1, s11, jump_
        sb t0, 0(s0)
        addi s0, s0, 1
        addi a1, a1, 1
        j loop_block
# ---------------------------------------------------------------------------- #




# reset all values stored in previous opcode to 0
# ---------------------------------------------------------------------------- #
clean_opcode:
        li t0, 0
        li a1, 0
        la s1, opcode                   # point $s1 to the address of opcode
loop_opcode:
            li s11, 10                          # s11 = 10
        beq a1, s11, jump_
        sb  t0, 0(s1)
        addi s1, s1, 1
        addi a1, a1, 1
        j loop_opcode
# ---------------------------------------------------------------------------- #
```

2. Explanation

We create libraries to store possible opcodes and registers. For the opcode library, we need to store each opcode's syntax and its type, which depends on the number of parameters, whether it uses registers, or includes a label

- First, read the input string.

```
main:
   li a7, 4
   la a0, input_message              # Print input message
   ecall
read_data:
        li a7, 8
        la a0, buffer                                    # Store input data in buffer
        li a1, 100
        ecall
   mv s0, a0                         # Store adress of input string into s0

   jal clear_whitespace                          # Jump to clear white space
```

- Use the **clear_whitespace** function to remove any leading whitespace or tab characters.

```
# clear whitespace until the first valid character
clear_whitespace:
        mv      t0, s0                                        # load t0 as the
address of input string
        lb t1, 0(t0)                                  # read the first char
        li s11, ' '                                           # s11 = temp = ' '
        beq t1, s11, loop_whitespace              # if the first char is a whitespace
then delete
        li s11, 9                                             # s11 = temp = tab
character
        beq t1, s11, loop_whitespace              # if first char is a tab character then
delete
        jr ra                                                 # return when the first
char is neither a whitespace or a tab char
loop_whitespace:
        lb t1, 0(t0)                                  # read current character
        li s11, ' '                                           # s11 = temp = ' '
        beq t1, s11, whitespace_found             # if the first char is a whitespace
```

```
then increment address
        li s11, 9                                   # s11 = temp = tab
character
        beq t1, s11, whitespace_found         # if first char is a tab character then
increment address
        mv s0, t0                                   # there is no more
invalid char then update s0
        jr ra
whitespace_found:
        addi t0, t0, 1                          # increment address of input
string by 1 to delete invalid char
        j loop_whitespace                       # continue whitespace loop
```

- Traverse the input string to find the first newline or whitespace character, storing each character into **opcode** during traversal:
  - If a newline (**\n**) is encountered, it means the user did not provide operands. Handle this by jumping to the missing operand error.
  - If a whitespace character is found, check if the provided opcode exists in the **opcodeLibrary** using the **check_opcode** function.
- If the opcode is valid, proceed to **opcode_done**. After clearing the remaining whitespace in the input string, determine the instruction type, compare it with the 9 predefined types, and validate its syntax accordingly.

```
read_opcode:
        la a1, opcode                           # a1 is used for incrementing
opcode character position
        la s1, opcode                           # s1 = address of opcode
        mv t0, a0                                   # t0 = a0
loop_read_opcode:
        lb t1, 0(t0)                            # t1 = current character in
opcode
        li s11, ' '                                 # s11 = temp = ' '
        beq t1, s11, check_opcode             # if a whitespace is found then
check the opcode
        li s11, '\n'                            # s11 = temp = '\n'
        beq t1, s11, missing_                   # if a newline character is
found then the string is missing operands
        sb t1, 0(a1)                            # store current character into
opcode
        addi t0, t0, 1                          # continue check the next
char
        addi a1, a1, 1                          # increment current address
```

```
of opcode

        j loop_read_opcode

#Check opcode
check_opcode:
        mv a1, s1                                   # a1 = s1 = address of
opcode
        mv s0, t0                                   # s0 points to the
character after opcode
        la s2, opcodeLibrary                  # s2 = address of opcode library
        jal check
        j invalid_opcode                          # Jump to invalid opcode

check:
        mv a2, s2                                   # a2 points to the
beginning of library
loop_check:
        lb t2, 0(a2)                              # load each character from
library
        li s11, ','                                 # s11 = temp = ' '
        beq t2, s11, evaluation1             # if meet colon, evaluate whether it
is correct
        lb t1, 0(a1)                              # load each character in input
opcode
        beq t2, zero, jump_                   # if current character in
opcode is \0 then we checked all possible opcodes in the library -> no valid input
opcode
        bne t1, t2, next_opcode               # mismatch
        addi a1, a1, 1                            # next char
        addi a2, a2, 1
        j loop_check

evaluation1:
        lb t1, 0(a1)                              # load current character of
opcode
        beq t1, zero, opcode_done          # if current character of opcode is
null then it has matched an opcode in opcode library
        j next_opcode                               # else continue
checking opcode in opcodeLibrary

next_opcode:
        addi s2, s2, 8                            # increment s2 by 8 because
each opcode has 8 bytes in opcode library
        mv a2, s2                                   # udpate a2
```

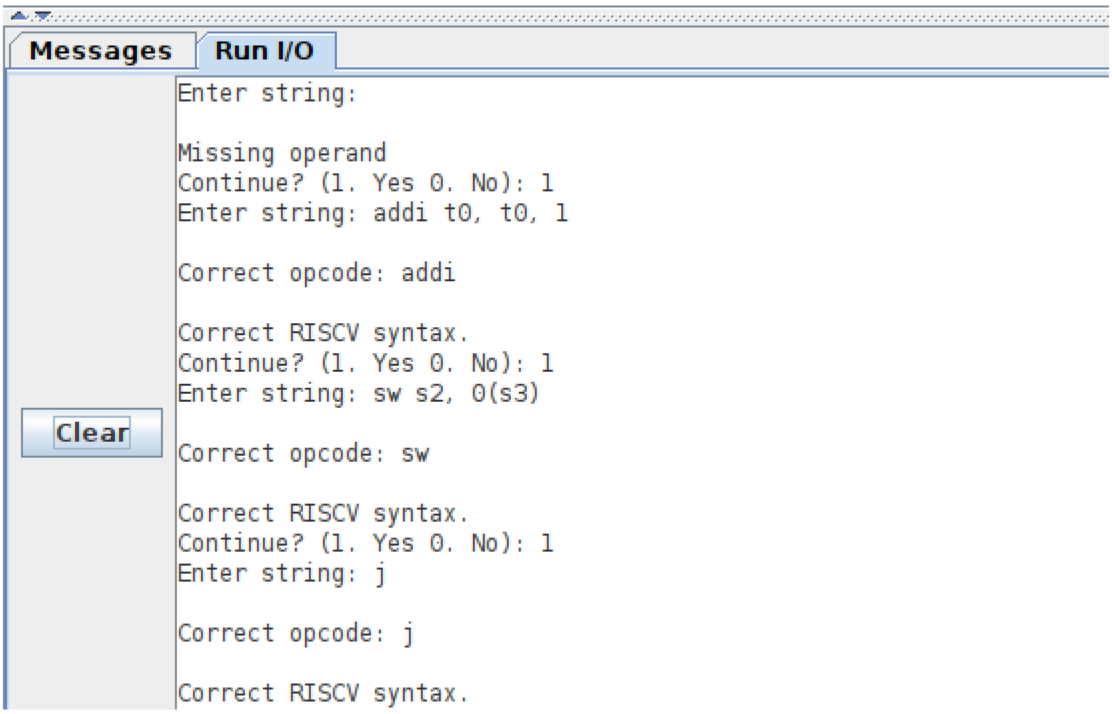
```
        mv a1, s1                                   # reset running for
opcode
        j loop_check

opcode_done:
        jal correct_opcode                          # print correct opcode
        addi a2, a2, 1

        lb t2, 0(a2)                                # load synax type in t2
        jal clear_whitespace                   # point to s0 to next vailid character
after opcode
        addi t2, t2, -48                            # minus value of t2 by 48 to
get the integer value
        li s11, 1                                   # s11 = temp = 1
        beq t2, s11, Type_1
        li s11, 2                                   # s11 = temp = 2
        beq t2, s11, Type_2
        li s11, 3                                   # s11 = temp = 3
        beq t2, s11, Type_3
        li s11, 4                                   # s11 = temp = 4
        beq t2, s11, Type_4
        li s11, 5                                   # s11 = temp = 5
        beq t2, s11, Type_5
        li s11, 6                                   # s11 = temp = 6
        beq t2, s11, Type_6
        li s11, 7                                   # s11 = temp = 7
        beq t2, s11, Type_7
        li s11, 8                                   # s11 = temp = 8
        beq t2, s11, Type_8
        li s11, 9                                   # s11 = temp = 9
        beq t2, s11, Type_9
end:
        j ending
```

- There are 9 types of instruction we defined in the program:
    - **Type_1**: Includes an opcode followed by three registers.
    - **Type_2**: Includes an opcode, two registers, and an immediate value (a number).
    - **Type_3**: Includes an opcode followed by two registers.
    - **Type_4**: Includes an opcode, one register, a shift amount, and an address specified in a second register.
    - **Type_5**: Includes an opcode, one register, and an immediate value (a number).
    - **Type_6**: Includes an opcode, one register, and a label.
    - **Type_7**: Includes an opcode followed by one register.
    - **Type_8**: Includes an opcode, two registers, and a label.

Type_9: Includes an opcode followed by a label.

- The subprograms we used in the program to check:
    - **check_opcode:** Checks if the input opcode exists in opcodeLibrary and jumps to appropriate labels based on the result.
    - **reg_check:** Validates if the next parameter is a valid register and updates s0 to point to the next parameter.
    - **check_gap:** Ensures proper syntax by checking for a comma and moves s0 to the next parameter after clearing whitespace.
    - **num_check:** Confirms if the input is a valid number and updates s0 to the character following the number.
    - **address_check:** Validates the shift amount and address syntax, including confirming the second parameter as a register.
    - **label_check:** Verifies if the label syntax is correct and sets s7 to indicate validity.
    - **check_end :** Removes trailing whitespaces and checks if the end of the input is valid.

3. Demonstration



Part II: Flip Card Game

1. Overview

The Flip Card Game is an interactive memory-based game implemented using a Bitmap Display and a Key Matrix. The game features a grid of face-down cards (4x4), where players flip cards two at a time to find matching pairs. The objective is to reveal all pairs correctly by memorizing card positions.

2. Methods and Algorithms

1. **Grid Setup**
    - The display consists of a 4x4 grid with 8 unique pairs of colors assigned randomly to positions.
    - Initially, all cards are shown face-down, with identical back-face images/colors.
2. **Randomization**
    - A random number generator system call is used to assign card colors to random grid positions.
    - Ensures that each color appears exactly twice.
3. **Card Flip Mechanism**
    - Players flip a card by pressing the corresponding Key Matrix button.
    - Button presses are mapped to grid coordinates (e.g., top-left is 1, bottom-right is 16).
4. **Match Checking**
    - When two cards are flipped:
        - If colors match → The cards remain open.
        - If colors don't match → The cards are flipped back face-down after a short delay.
5. **Game End Condition**
    - The game concludes when all card pairs are successfully revealed.

3. Implementation

- **Bitmap Display**
    - Used to visually render the 4x4 card grid.
    - Cards are drawn using distinct color codes for easy differentiation.
- **Key Matrix**

- Detects user input to identify the card to be flipped.
- Maps input keys to grid positions (e.g., Key 1 for grid[0][0]).
- We represent the matrix just by an array by using this formula: row * 4 + column = index

- **Game Logic**
  - Uses basic conditional checks:
    - Check if two flipped cards match.
    - Update game state if all cards are revealed.
- **System Calls**
  - Random Number Generation: Used for shuffling card positions.
  - Timer Delay: Introduced a delay before unmatched cards are flipped back.

4. Simulation Results

- **Game Initialization**:
  - The cards were randomized successfully on every new start.
- **Card Flipping**:
  - Pressing the correct key flipped cards to reveal colors.
- **Matching Logic**:
  - Correct pairs stayed open, and unmatched pairs flipped back as expected.
- **Game Completion**:
  - The game ended properly once all pairs were matched, displaying a "Game Over" message.

5. Conclusion

The Flip Card Game provides a functional, interactive memory game using **RISC-V assembly language**. Key features include card randomization, user input handling via a key matrix, and dynamic visual updates on the Bitmap Display.

6. Results:

- **Choose row and column to flip**

```
Run I/O
Enter row (1->4): 1
Enter coloumn (1->4): 1
Enter row (1->4):
```

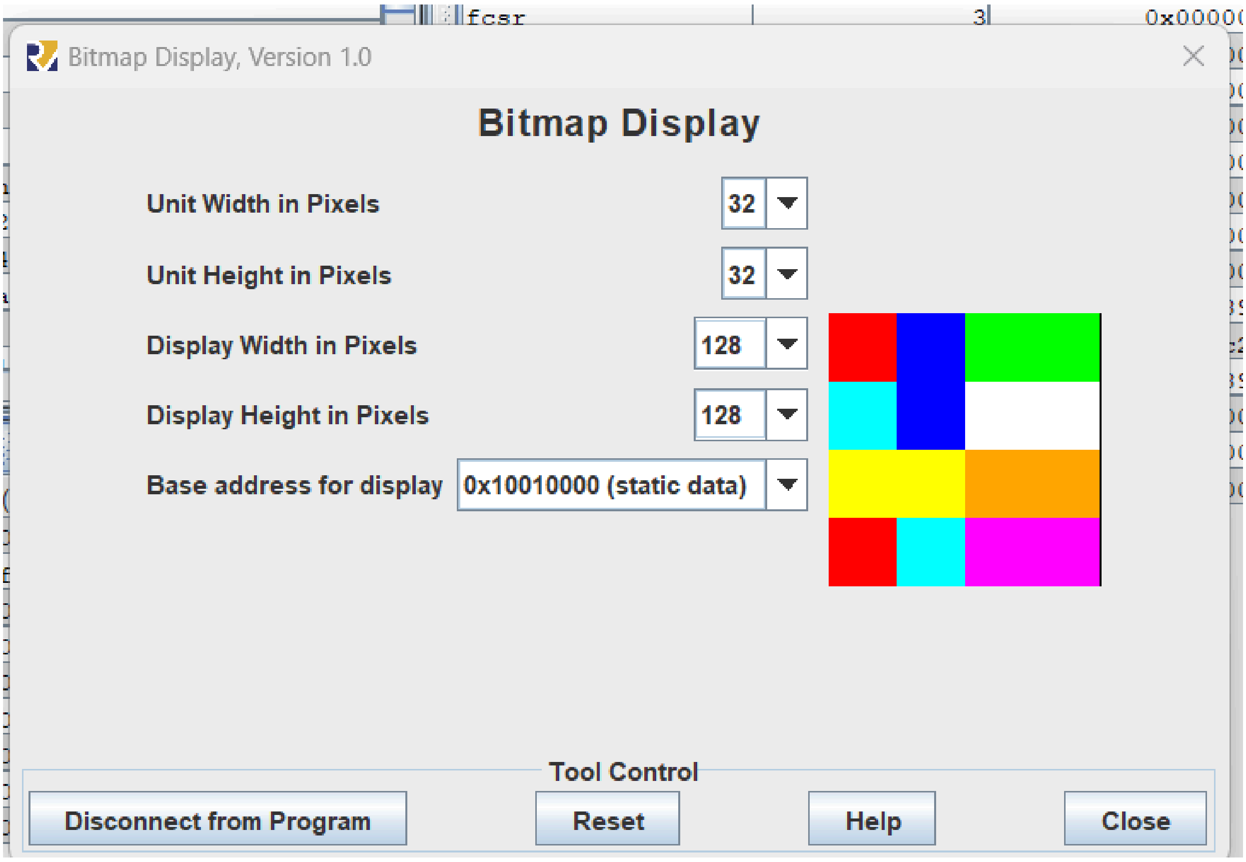
-
- **Wrong input**

```
Enter row (1->4): 1
Enter coloumn (1->4): 1
Enter row (1->4): 1
Enter coloumn (1->4): 5
Wrong input. Please, try again
Enter row (1->4):
```

-
- **Opened card:**

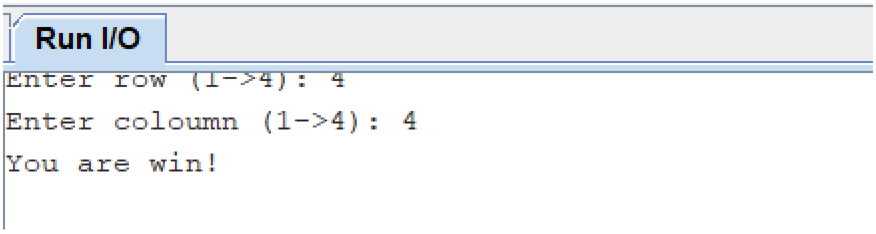
```
Enter row (1->4): 1
Enter coloumn (1->4): 5
Wrong input. Please, try again
Enter row (1->4): 1
Enter coloumn (1->4): 1
Opened card, try another
Enter row (1->4):
```

-
- **Visualization by bitmap tool:**



-

- **Win**



-

7. Source code:

```
.eqv MONITOR_SCREEN 0x10010000  # Start address of the bitmap display
.eqv RED           0x00FF0000   # Common color values
.eqv GREEN         0x0000FF00
.eqv BLUE          0x000000FF
.eqv WHITE         0x00FFFFFF
.eqv YELLOW        0x00FFFF00
.eqv BLACK         0xFF000000
.eqv CYAN          0xFF00FFFF
.eqv MAGENTA       0xFFFF00FF
.eqv ORANGE 0xFFFFA500
```

```
.data
        space_distance: .space 8000
        colors: .word RED, RED, GREEN, GREEN, BLUE, BLUE, WHITE, WHITE,
YELLOW, YELLOW, ORANGE, ORANGE, CYAN, CYAN, MAGENTA, MAGENTA

        enter_row: .asciz "Enter row (1->4): "
        enter_column: .asciz "Enter coloumn (1->4): "
        win: .asciz "You are win!\n"
        wrong_inp: .asciz "Wrong input. Please, try again\n"
        opened: .asciz "Opened card, try another\n"
.text

setup_monitor:
        li s11, MONITOR_SCREEN
        li a1, 16
        li a2, 0
        li a3, BLACK

        loop1:
                beq a2, a1, end_loop1

                add a4, a2, a2
                add a4, a4, a4
                add a4, s11, a4

                sw a3, 0(a4)

                addi a2, a2, 1
                j loop1

        end_loop1:

shuffle:
   la t0, colors          # Load base address of the grid_colors array
   li t1, 16           # Load size of the array (16)

   li t2, 0                  # Initialize loop counter (i = 0)

shuffle_loop:
   bge t2, t1, shuffle_end      # If i >= array_size, exit loop

   # Generate random value 0 or 1
   li a7, 42                  # System call for random number
   ecall                     # Random number generated in a0
   li t3, 2                  # Set divisor (2 for 0 or 1)
```

```
   rem a0, a0, t3              # a0 = a0 % 2 (0 or 1)

   beq a0, zero, no_swap       # If random value is 0, skip the swap

   # Generate random index j
   li a7, 42                   # System call to get the random index for swapping
   ecall
   rem a1, a0, t1              # a1 = a0 % array_size (random index j)

   # Perform the swap: colors[i] <-> colors[j]
   slli t4, t2, 2              # t4 = i * 4 (byte offset for grid_colors[i])
   add t5, t0, t4              # t5 = &grid_colors[i]
   lw t6, 0(t5)                # t6 = grid_colors[i]

   slli s7, a1, 2              # t7 = j * 4 (byte offset for grid_colors[j])
   add s8, t0, s7              # t8 = &grid_colors[j]
   lw s9, 0(s8)                # t9 = grid_colors[j]

   sw s9, 0(t5)                # grid_colors[i] = grid_colors[j]
   sw t6, 0(s8)                # grid_colors[j] = grid_colors[i]

        no_swap:
           addi t2, t2, 1             # i++
           j shuffle_loop

shuffle_end:
   # Go to main

main:
        li s11, MONITOR_SCREEN # Address of monitor
        la s1, colors  # Address of store_colors[0]
        li a2, 16 # Size
        li a3, 0 # Point value

        li s10, BLACK

        li s9, 0 # idx1
        li a4, 0 # cnt value

        # We will win went score upto 16
        loop:
                beq a3, a2, end_loop

                la a0, enter_row
                li a7, 4
```

```
ecall

li a7, 5
ecall

li a7, 4
bgt a0, a7, wrong_input
li a7, 1
blt a0, a7, wrong_input # Check the row input

addi a5, a0, -1

la a0, enter_column
li a7, 4
ecall

li a7, 5
ecall

li a7, 4
bgt a0, a7, wrong_input
li a7, 1
blt a0, a7, wrong_input # Check the row input

addi a6, a0, -1

# Now we have row and coloumn => find idx
add a5, a5, a5
add a5, a5, a5
add a5, a5, a6 # a5 is idx

add a7, a5, a5
add a7, a7, a7
add t1, s11, a7 # Find address
lw t0, 0(t1) # Load current color

bne t0, s10, print_not_black  # Not is black enter again

addi a4, a4, 1
    print:
        add t1, s1, a7 # Address of colors[i]
        lw t0, 0(t1)

        addi t1, a5, 0
        jal print_color_to_monitor
```

```
            delay:
               li a0, 250000      # Load a large loop count (adjust for your clock
speed)

            delay_loop:
               addi a0, a0, -1      # Decrement the counter
               bnez a0, delay_loop  # If counter != 0, keep looping


        # If else cnt == 1, cnt == 2
            li t2, 1
            bne a4, t2, else
            if:
                # case cnt == 1 => store idx to s9
                addi s9, a5, 0
                j end_if_else
            else:
                add a7, s9, s9
                add a7, a7, a7 # find address
                add t1, s11, a7
                lw t2, 0(t1) # Load pre-color

                bne t2, t0, else_2
                if_check_same_color:
                    addi a3, a3, 2 # Increase points
                    j end_if2
                else_2:
                    sw s10, 0(t1) # Print black again
                    add a7, a5, a5
                    add a7, a7, a7
                    add t1, s11, a7
                    sw s10, 0(t1)

                end_if2:
                    li a4, 0

            end_if_else:

        continue_loop:
            j loop

        print_not_black:
            li a7, 4
            la a0, opened
```

```
                    ecall
                    j continue_loop

        end_loop:
                j end_main

        print_color_to_monitor: # Get two values t0_color and t1 idx
                add t2, t1, t1
                add t2, t2, t2 # Multiply 4 * t1
                add t2, s11, t2
                sw t0, 0(t2) # Print color
                jr ra

        wrong_input:
                li a7, 4
                la a0, wrong_inp
                ecall # Print wrong_inp message
                j loop

end_main:
        la a0, win
        li a7, 4
        ecall

        li a7, 10
        ecall # Exit()
```