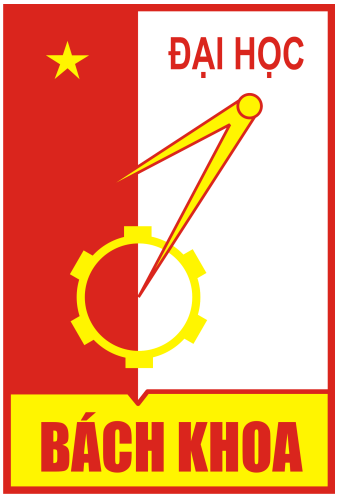
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY



Assembly Language and Computer Architecture Lab

Project

Group 16

Advisor: Dr. Le Ba Vui

HA NOI, JANUARY 2024

Member list

No.	Full name	Student ID
1	Trần Duy Khánh	20215213
2	Shwe Yee Win	2023T075

Exercise

Trần Duy Khánh does exercise 5 - Infix and postfix expressions.

Shwe Yee Win does exercise 2 - Moving a ball in the bitmap display

Contents

1 Exercise 5 - Infix and postfix expressions

1.1 Problem description

The infix notation is commonly used in texts. However, there is another type of notation that we often use to represent mathematical operations more efficiently than infix notation, and that is postfix notation. Some benefits of Postfix notation :

- Postfix notation eliminates the need for parentheses and operator precedence rules. The expression is evaluated strictly from left to right, making it easier to understand and evaluate.
- Postfix notation eliminates the ambiguity that can arise in infix notation due to multiple possible interpretations of parentheses and operator precedence
- Postfix notation lends itself well to stack-based evaluation algorithms. The evaluation process can be implemented using a stack. This approach allows for efficient and fast evaluation of postfix expressions.

Our goal in this exercise is to convert from infix notation to postfix notation and calculate the result of postfix notation.

1.2 Algorithm

1.2.1 Task 1 : convert infix to postfix expression

Step 1 : Scan the infix expression from left to right.

Step 2:

- If the scanned character is an operand, put it in the postfix expression.
- If the precedence and associativity of the scanned operator are greater than the precedence of the operator in the stack [or the stack is empty or the stack contains a '('], then push it in the stack.
- Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.

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

Step 3: Repeat steps 2 until all infix expressions is scanned. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.

Finally, print the postfix expression.

1.2.2 Task 2 : Evaluating Postfix expression

Step 1: Create a stack to store operands (or values).

Step 2: Scan the given expression from left to right and do the following for every scanned element.

- If the element is a number, push it into the stack.
- If the element is an operator, pop first two operands for the operator from the stack. Evaluate the operator and push the result back to the stack.

Step 3: When the expression is ended, the number in the stack is the final answer.

1.3 Code implementation with explaination

1. Input the infix expression and print it

```
start:
# input infix notation
        li $v0, 54
        la $a0, startMsg
        la $a1, infix
        la $a2, 256
        syscall
        beq $a1,-2,end                  # if cancel then end
        beq $a1,-3,start                # if enter then start
# print the infix notation
        li $v0, 4
        la $a0, prompt_infix
        syscall
        li $v0, 4
        la $a0, infix
        syscall
        li $v0, 11
        li $a0, '\n'
        syscall
# initialize
        li $s7,0                                        # check condition variable

                                                                # 1 - number from 0 - 99
                                                                # 2 -  "* / + - %"
                                                                # 3 - "("
                                                                # 4 - ")"
        li $t9,0                                        # count the number of operand
```

2. Scan the infix expression , check wrong input

We only allow 2 digit number , so we use $ t9 register to check the condition of operands We also use $ s7 register to check the type of input character. Check the order of character that lead to wrong input .

```
        li $t5,-1                               # store postfix
        li $t6,-1                               # store operators
        la $t1, infix                       # load address
        la $t2, postfix
        la $t3, operator
        addi $t1,$t1,-1                     # infix init index = -1
# change to postfix
scanInfix:
# check input
        addi $t1, $t1, 1                        # infix index ++
        lb $t4, 0($t1)                          # value of infix notation
        beq $t4, ' ', scanInfix             # space then continue
        beq $t4, '\n', EOF                      # if enter , then pop all the remaining operator in stack
        beq $t9, 0, digit1                      # t9 =  0 => 0 number
        beq $t9, 1, digit2                      # t9 = 1 => 1 numbers
        beq $t9, 2, digit3                      # t9 = 2 => already have 2 numbers => can not input more number
        continueScan:
        beq $t4, '+', plusMinus
        beq $t4, '-', plusMinus
        beq $t4, '*', multiplyDivideModulo
        beq $t4, '/', multiplyDivideModulo
        beq $t4, '%', multiplyDivideModulo
        beq $t4, '(', openBracket
        beq $t4, ')', closeBracket
wrongInput:
        li $v0, 55
        la $a0, errorMsg
```

```
        j popAllOperatorInStack
digit1:
        beq $t4,'0',storeDigit1
        beq $t4,'1',storeDigit1
        beq $t4,'2',storeDigit1
        beq $t4,'3',storeDigit1
        beq $t4,'4',storeDigit1
        beq $t4,'5',storeDigit1
        beq $t4,'6',storeDigit1
        beq $t4,'7',storeDigit1
        beq $t4,'8',storeDigit1
        beq $t4,'9',storeDigit1
        j continueScan

digit2:
        beq $t4,'0',storeDigit2
        beq $t4,'1',storeDigit2
        beq $t4,'2',storeDigit2
        beq $t4,'3',storeDigit2
        beq $t4,'4',storeDigit2
        beq $t4,'5',storeDigit2
        beq $t4,'6',storeDigit2
        beq $t4,'7',storeDigit2
        beq $t4,'8',storeDigit2
        beq $t4,'9',storeDigit2
```

```
        j continueScan
digit3:
        # if scan a third number --> error
        beq $t4,'0',wrongInput
        beq $t4,'1',wrongInput
        beq $t4,'2',wrongInput
        beq $t4,'3',wrongInput
        beq $t4,'4',wrongInput
        beq $t4,'5',wrongInput
        beq $t4,'6',wrongInput
        beq $t4,'7',wrongInput
        beq $t4,'8',wrongInput
        beq $t4,'9',wrongInput

        jal numberToPostfix
        j continueScan

        storeDigit1:
        beq $s7,4,wrongInput                # if number is read after ")" then error
        addi $s4,$t4,-48                        # change char type to integer type
        add $t9,$zero,1                         # t9 changes to 1
        li $s7,1
        j scanInfix
storeDigit2:
        beq $s7,4,wrongInput                # if number is read after ")" then error
        addi $s5,$t4,-48                        # change char type to integer type
```

3. Convert infix expression to postfix expression using stack (using “operator” stack and “postfix” array)

We use loops to push the operators in the stack , and pop the operators when satisfy the condition.

```
continuePlusMinus:
        beq $t6,-1,inputOperatorToStack             #nothing in stack
        add $t8,$t6,$t3                                 # load address of operator
        lb $t7,($t8)
        beq $t7,'(',inputOperatorToStack
        beq $t7,'+',equalPrecedence
        beq $t7,'-',equalPrecedence
        beq $t7,'*',lowerPrecedence
        beq $t7,'/',lowerPrecedence
        beq $t7,'%',lowerPrecedence
multiplyDivideModulo:
        beq $s7,2,wrongInput
        beq $s7,3,wrongInput
        beq $s7,0,wrongInput
        li $s7,2
        beq $t6,-1,inputOperatorToStack
        add $t8,$t6,$t3                                 # load address of operator
        lb $t7,($t8)                                    # load value of operator
        beq $t7,'(',inputOperatorToStack
        beq $t7,'+',inputOperatorToStack
        beq $t7,'-',inputOperatorToStack
        beq $t7,'*',equalPrecedence
        beq $t7,'/',equalPrecedence
        beq $t7,'%',equalPrecedence
openBracket:
        beq $s7,1,wrongInput                        # "(" is placed after operand or ")"-> error
        beq $s7,4,wrongInput
```

```
openBracket:
        beq $s7,1,wrongInput                            # "(" is placed after operand or ")"-> error
        beq $s7,4,wrongInput
        li $s7,3
        j inputOperatorToStack
closeBracket:
        beq $s7,2,wrongInput                    # ")" is placed after operator or "(" -> error.
        beq $s7,3,wrongInput
        li $s7,4
        add $t8,$t6,$t3
        lb $t7,($t8)
        beq $t7,'(',wrongInput                  # it is () -> error.
continueCloseBracket:
        beq $t6,-1,wrongInput
        add $t8,$t6,$t3
        lb $t7,($t8)
        beq $t7,'(',matchBracket                # match the bracket
        jal PopOperatorToPostfix                # day toan tu o dinh vao postfix
        j continueCloseBracket                # tiep tuc vong lap cho den khi tim duoc ngoac phu hop

matchBracket:
        addi $t6,$t6,-1                       #remove brackets.
        j scanInfix
```

4. Calculate postfix expression and print value (using “stack2” stack and “postfix” array)
We use loops to push the operands in the stack , and pop first two operands when scan the operator

```
CalculatorPost:
        addi $t6,$t6,1                  # index postfix ++
        add $t8,$t2,$t6                 # load address of postfix
        lbu $t7,($t8)
        bgt $t6,$t5,printResult
        bgt $t7,99,calculate            # value of postfix > 99 --> operator --> pop 2 operands
        # if not then it is operand
        addi $t9,$t9,4                  # index stack + 4
        add $t4,$t3,$t9
        sw $t7, ($t4)                    #put operand into stack
        j CalculatorPost                        # Loop
        calculate:
                # Pop first operand
                add $t4,$t3,$t9
                lw $t0,($t4)
                # pop second operand
                addi $t9,$t9,-4
                add $t4,$t3,$t9
                lw $t1,($t4)
                # Decode operator
                beq $t7,143,plus
                beq $t7,145,minus
                beq $t7,142,multiply
                beq $t7,147,divide
                beq $t7, 137, modulo
                plus:
                        add $t0,$t0,$t1
```

```
                sw $t0,($t4)
#               li $t0, 0                       # Reset t0, t1
#               li $t1, 0
                j CalculatorPost
        multiply:
                mul $t0, $t1,$t0
                sw $t0,($t4)
#               li $t0, 0                       # Reset t0, t1
#               li $t1, 0
                j CalculatorPost
        divide:
                div $t1, $t0
                mflo $t0
                sw $t0,($t4)
#               li $t0, 0                       # Reset t0, t1
#               li $t1, 0
                j CalculatorPost
        modulo:
                div $t1, $t0
                mfhi $t0
                sw $t0,($t4)
#               li $t0, 0                       # Reset t0, t1
#               li $t1, 0
                j CalculatorPost
printResult:
        li $v0, 4
        la $a0, prompt_result
```

1.4 Result

When Input infix expression correctly, we got the postfix expression and result like this

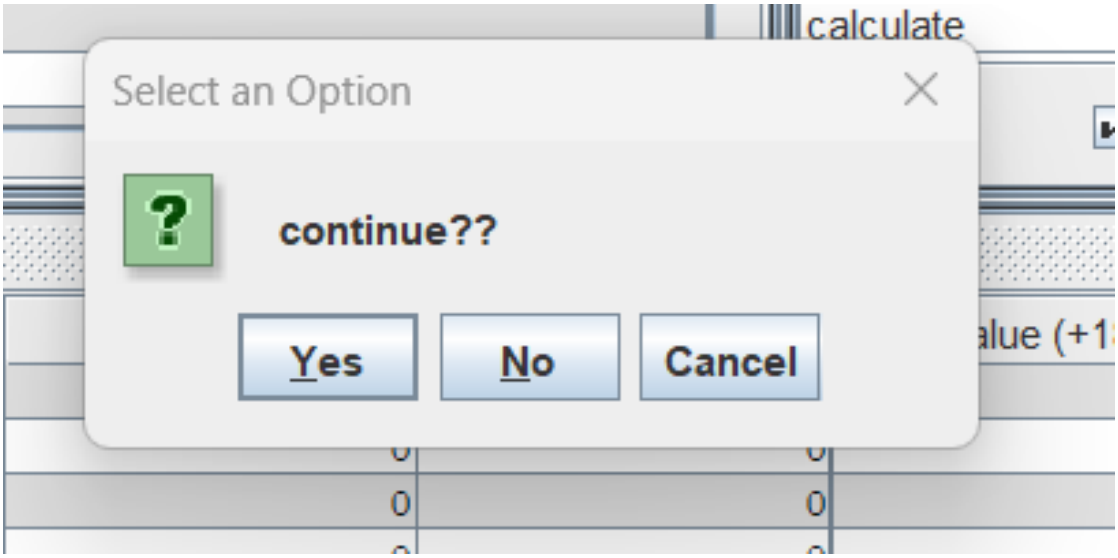
```
infix expression:   1+2*3+4

postfix expression: 1 2 3 * + 4 +
result: 11
```

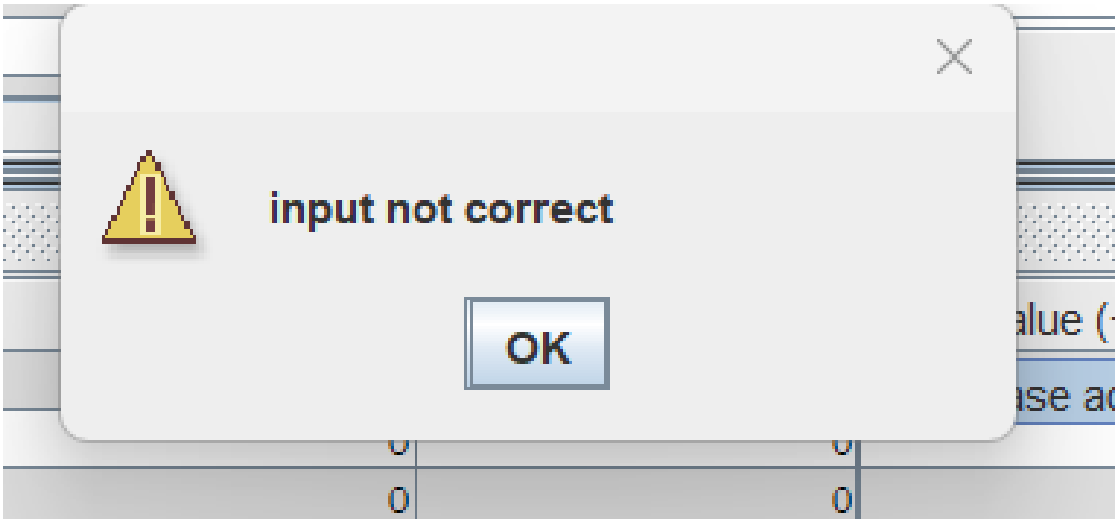
```
infix expression:   (12+20)*2+1-5

postfix expression: 12 20 + 2 * 1 + 5 -
result: 60
```

After we input , we will have a dialog to ask if you want to continue or not.



When input incorrectly, we got the alert.



2 Exercise 2 - Moving a ball in the bitmap display

2.1 Code Explaination

1) Creating a program that displays a movable round ball on the bitmap screen.

- define KEY data for input from keyboard and COLOR with hexacode for monitor(black) and circle(yellow)

```
.eqv KEY_CODE 0xFFFF0004        #ASCII code from keyboard, 1 byte
.eqv KEY_READY 0xFFFF0000       # =1 if has a new keycode ?

.eqv MONITOR_SCREEN 0x10010000
.eqv YELLOW 0x00FFFF00

.data
circle_end:     .word   1               # The end of the "circle" array
circle:         .word                   # The pointer to the "circle" 2-dimentional array

.text
setup:
        addi    $s0, $0, 255            # x = 255
        addi    $s1, $0, 255            # y = 255
        addi    $s2, $0, 1              # dx = 1
        add     $s3, $0, $0             # dy = 0
        addi    $s4, $0, 20             # r = 20 (radius)
        addi    $a0, $0, 50             # t = 50ms/frame
        jal     circle_data

input:
        li      $k0, KEY_READY  # Check whether there is input data
        lw      $t0, 0($k0)
        bne     $t0, 1, edge_check
        jal     direction_change

# Check whether the circle has touched the edge
```

2) Displaying width and height to 512 pixels, unit width and height to 1 pixel. The default position is the center of the screen. (x & y at 255)

- setup circle with radius, moving time, xy location in bitmap and its direction.
- it has a lot of procedures to draw a circle (circle_data to pixel_save)

```
        jal     draw_circle             # Erase the old circle

        add     $s0, $s0, $s2           # Set x and y to the coordinates
# of the center of the new circle
        add     $s1, $s1, $s3
        li      $s5, YELLOW             # Set color to yellow
        jal     draw_circle             # Draw the new circle

loop:
        li $v0, 32                      # Syscall value for sleep
        syscall
        j       input                   # Renew the cycle

# Procedure below

circle_data:
        addi    $sp, $sp, -4            # Save $ra
        sw      $ra, 0($sp)
        la      $s5, circle             # $s5 becomes the pointer of the "circle" array
        mul     $a3, $s4, $s4           # $a3 = r^2
        add     $s7, $0, $0             # pixel x (px) = 0
#-----
pixel_data_loop:
        bgt     $s7, $s4, data_end
        mul     $t0, $s7, $s7           # $t0 = px^2
        sub     $a2, $a3, $t0           # $a2 = r^2 - px^2 = py^2
        jal     root                    # $a2 = py
```

3) If the ball touches the edge of the screen, it will move in the opposite direction.

- check whether there is input data
- If not, then check whether the circle has touched the edge
- (check_right/check_left/check_up/check_down),
- if the edge touches reverse_direction then move_circle

```
# Check whether the circle has touched the edge
edge_check:


right:
        bne     $s2, 1, left
        j       check_right

left:
        bne     $s2, -1, down
        j       check_left

down:
        bne     $s3, 1, up
        j       check_down

up:
        bne     $s3, -1, move_circle
        j       check_up

move_circle:
        add     $s5, $0, $0             # Set color to black
        jal     draw_circle             # Erase the old circle

        add     $s0, $s0, $s2           # Set x and y to the coordinates
```

```
        j       move_circle                     # Return if not

check_down:
        add     $t0, $s1, $s4                   # Set $t0 to the down side of the circle
        beq     $t0, 511, reverse_direction     # Reverse direction if the side has touched
#the edge
        j       move_circle                     # Return if not

check_up:
        sub     $t0, $s1, $s4                   # Set $t0 to the up side of the circle
        beq     $t0, 0, reverse_direction       # Reverse direction if the side has touched
#the edge
        j       move_circle                     # Return if not

reverse_direction:
        sub     $s2, $0, $s2                    # dx = -dx
        sub     $s3, $0, $s3                    # dy = -dy
        j       move_circle

draw_circle:
        addi    $sp, $sp, -4                    # Save $ra
        sw      $ra, 0($sp)
        la      $s6, circle_end
        lw      $s7, 0($s6)     # $s7 becomes the end address of the "circle" array
        la      $s6, circle     # $s6 becomes the pointer to the "circle" array
```

4) To move an object, we delete it from its old position and draw it in a new position. To delete an object, draw it with the background color.

- delete the old circle by changing with its background color and
- draw a new circle by setting x and y to the coordinates to move opposite direction of edge

```
        add     $a1, $0, $s7            # $a1 = px
        add     $s6, $0, $0     # After saving (px, py), (-px, py), (-px, -py), (px, py),
#we swap px and py, then save (-py, px), (py, px), (py, -px), (-py, -px)

symmetric:
        beq     $s6, 2, finish
        jal     pixel_save              # px, py >= 0
        sub     $a1, $0, $a1
        jal     pixel_save              # px <= 0, py >= 0
        sub     $a2, $0, $a2
        jal     pixel_save              # px, py <= 0
        sub     $a1, $0, $a1
        jal     pixel_save              # px >= 0, py <= 0

        add     $t0, $0, $a1            # Swap px and -py
        add     $a1, $0, $a2
        add     $a2, $0, $t0
        addi    $s6, $s6, 1
        j       symmetric

finish:
        addi    $s7, $s7, 1
        j       pixel_data_loop

data_end:
        la      $t0, circle_end
```

```
data_end:
        la      $t0, circle_end
        sw      $s5, 0($t0)             # Save the end address of the "circle" array
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        jr      $ra

root:                                   # Find the square root of $a2
        add     $t0, $0, $0             # Set $t0 = 0
        add     $t1, $0, $0             # $t1 = $t0^2

root_loop:
        beq     $t0, $s4, root_end      # If $t0 exceeds 20, 20 will be the square root
        addi    $t2, $t0, 1             # $t2 = $t0 + 1
        mul     $t2, $t2, $t2           # $t2 = ($t0 + 1)^2
        sub     $t3, $a2, $t1           # $t3 = $a2 - $t0^2
        bgez    $t3, continue           # If $t3 < 0, $t3 = -$t3
        sub     $t3, $0, $t3

continue:
        sub     $t4, $a2, $t2           # $t4 = $a2 - ($t0 + 1)^2
        bgez    $t4, compare            # If $t4 < 0, $t4 = -$t4
        sub     $t4, $0, $t4

compare:
        blt     $t4, $t3, root_continue # If $t3 >= $t4, $t0 is not nearer to square root of $a2 than $t0 + 1
```

5) The direction of movement depends on the key pressed from the keyboard. (W moves up, S moves down, A moves left, D moves right, Z speeds up, X slows down).

- for direction_change, check input data * and current position
- If input is W, dy must be 1 and for S, dy is -1
- If input is A, dx must be -1 and for D, dx is 1
- If input is Z, increase moving time and for X, decrease default time

```
compare:
        blt     $t4, $t3, root_continue# If $t3 >= $t4, $t0 is not nearer to square root of $a2 than $t0 + 1
        add     $a2, $0, $t0                    # Else $t0 is the nearest number to square root of $a2
        jr      $ra

root_continue:
        addi    $t0, $t0, 1
        add     $t1, $0, $t2
        j       root_loop

root_end:
        add     $a2, $0, $t0
        jr      $ra

pixel_save:
        sw      $a1, 0($s5)                     # Store px in the "circle" array
        sw      $a2, 4($s5)                     # Store py in the "circle" array
        addi    $s5, $s5, 8                     # Move the pointer to a null block
        jr      $ra
#---------
direction_change:
        li      $k0, KEY_CODE
        lw      $t0, 0($k0)
```

```
        lw      $s7, 0($s6)     # $s7 becomes the end address of the "circle" array
        la      $s6, circle     # $s6 becomes the pointer to the "circle" array

draw_loop:
        beq     $s6, $s7, draw_end              # Stop when $s6 = $s7
        lw      $a1, 0($s6)                     # Get px
        lw      $a2, 4($s6)                     # Get py
        jal     pixel_draw
        addi    $s6, $s6, 8                     # Get to the next pixel
        j       draw_loop

draw_end:
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        jr      $ra

pixel_draw:
        li      $t0, MONITOR_SCREEN
        add     $t1, $s0, $a1                   # final x (fx) = x + px
        add     $t2, $s1, $a2                   # fy = y + py
        sll     $t2, $t2, 9                     # $t2 = fy * 512
        add     $t2, $t2, $t1                   # $t2 += fx
        sll     $t2, $t2, 2                     # $t2 *= 4
        add     $t0, $t0, $t2
        sw      $s5, 0($t0)
        jr      $ra
```

```
case_d:
        bne     $t0, 'd', case_a
        addi    $s2, $0, 1                      # dx = 1
        add     $s3, $0, $0                     # dy = 0
        jr      $ra

case_a:
        bne     $t0, 'a', case_s
        addi    $s2, $0, -1                     # dx = -1
        add     $s3, $0, $0                     # dy = 0
        jr      $ra

case_s:
        bne     $t0, 's', case_w
        add     $s2, $0, $0                     # dx = 0
        addi    $s3, $0, 1                      # dy = 1
        jr      $ra

case_w:
        bne     $t0, 'w', case_x
        add     $s2, $0, $0                     # dx = 0
        addi    $s3, $0, -1                     # dy = -1
        jr      $ra

case_x:
        bne     $t0, 'x', case_z
```

```
case_x:
        bne     $t0, 'x', case_z
        addi    $a0, $a0, 30                    # t(50) += 30
        jr      $ra

case_z:
        bne     $t0, 'z', default
        beq     $a0, 0, default                 # Only reduce t when t >= 0
        addi    $a0, $a0, -30                   # t(50+30) -= 30

default:
        jr      $ra

check_right:
        add     $t0, $s0, $s4                   # Set $t0 to the right side of the circle
        beq     $t0, 511, reverse_direction     # Reverse direction if the side has touched
#the edge
        j       move_circle                     # Return if not

check_left:
        sub     $t0, $s0, $s4                   # Set $t0 to the left side of the circle
        beq     $t0, 0, reverse_direction       # Reverse direction if the side has touched
#the edge
        j       move_circle                     # Return if not

check_down:
```

2.2 Result

The result of the program .