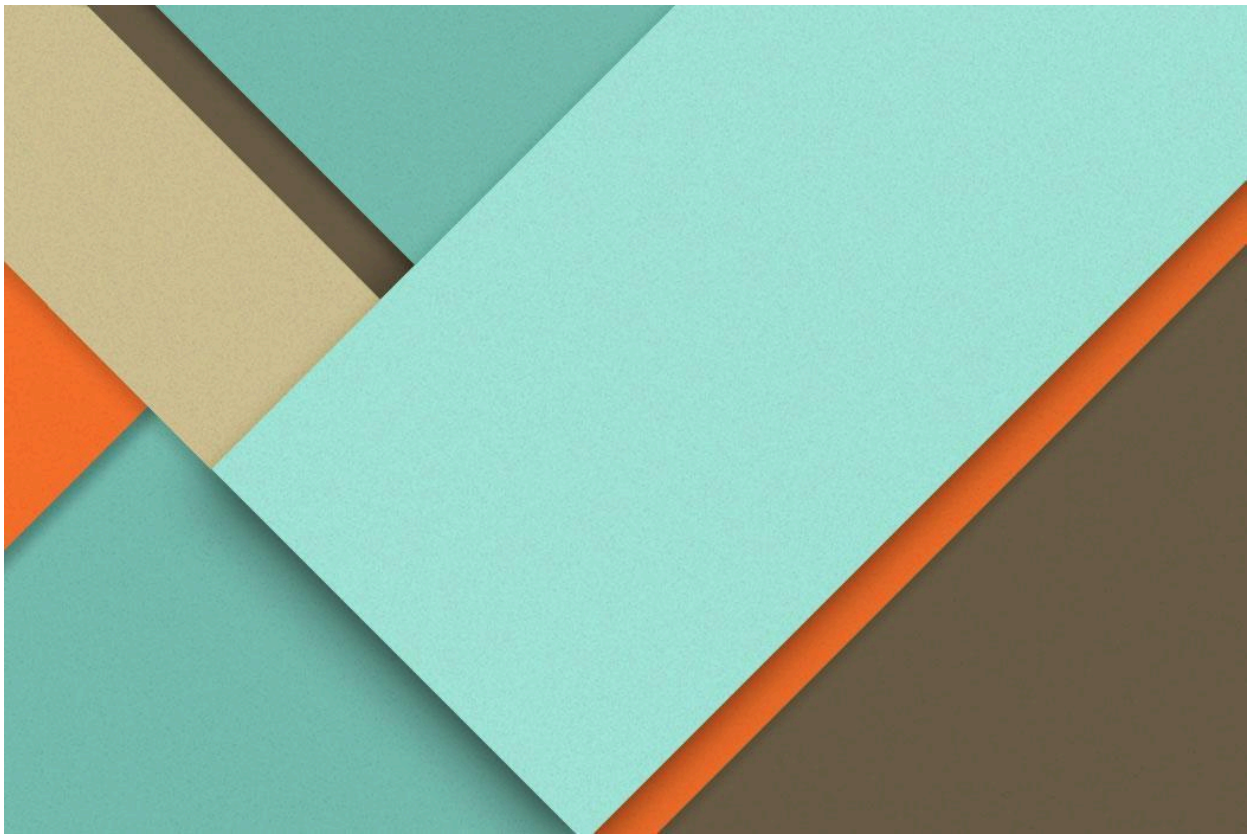


Traveller's Transpiler (A6)

Hall 5 - Team 1

Srujan Wadkar (221212)

Anaswar K B (220138)

Sanskaar Srivastava (220967)

Kartik Kulkarni (210493)

Overview

IITK Traveler is a graph-based esoteric language while Super Stack is a procedural one. There are 56 locations in IITK Traveler which have connected functions. Super Stack has 26 instructions. The corresponding mapping has been achieved using various functions whose implementation details are listed below.

According to the Problem Statement, we have not implemented random instruction. Also, the Logical Operations (and, or, not, xor, nand) have been converted to their Bitwise counterparts (like the C++) variants.

Development OS : Linux

Language Used for the Transpiler: C++

For running the code :

1 : Compile main.cpp- g++ main.cpp

2 : ./a.out <input file path>

Implementation Details

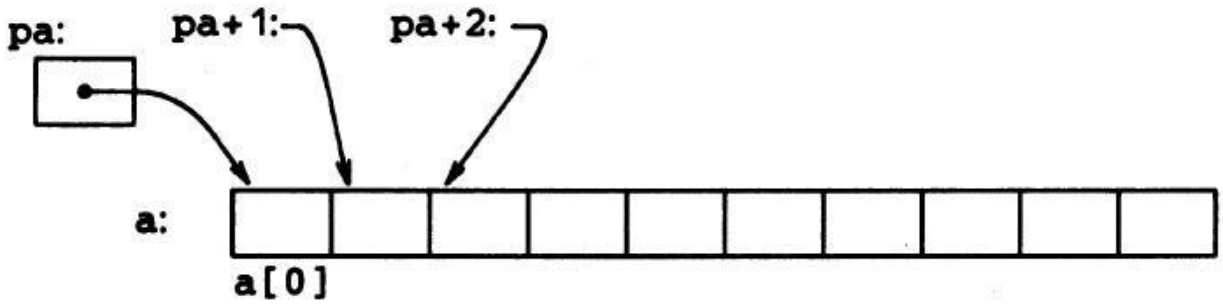
Basically IITK-Traveler consists of 3 pointers, mem_1, mem_2, mem_3 which use an infinite memory lane to store data. We have designed some algorithms in such a way that make use of this memory lane as a stack (just like in the super stack language) to store data and modify it using various operations.

So, to make use of this memory lane as a stack we arrange the 3 pointers in the following ways:-

1. **mem_1 :** *Points to the element just before the top of the stack.*
2. **mem_2:** *Points to the element on the top of the stack.*
3. **mem_3:** *Represents the end of the stack.*

Initially mem_2 has an EOS character and mem_3 has an EOS character as well, these EOS characters are very special and shall be used to implement the stack behavior in the language. We maintain the stack between these two EOS literals.

This is done using the ***initialize()*** function.



We have defined 3 global variables :

1. **cond:-** Keeps track of the conditional variable in the IITK-Traveler code. This variable has to be updated repetitively.
2. **current_position:-** We have initialized the current position as “start” (the initial point of the IITK-Traveler code). We keep updating the current_position in order to link the upcoming code with the previous one.
3. **loop_history:** Loop history keeps the record of the conditional variables used in the loops, we have initialized its datatype as a stack so that we can get the most recent occurrence of the loop initiated.

Q. Why do we go to oat_stage before every operation?

A- We increment the conditional variable in our code to ensure that for the same condition value we do not reach two different locations.

Basic Functions

123

An arbitrary number given in the super stack code is to be stored on the top of the stack (memory lane) in the IITK-traveler language.

To push the arbitrary number **N** in the stack, we use oat_stairs_2 **N** number of times.

add

We use hall_2 for this purpose. Input is taken with the help of iit_gate_in_1 and 2 and the result is stored on the top of stack by shifting the pointers and removing the numbers taken as input by using hall_13_3, rm_1, kd_1, etc. We have implemented a master function ***arithmetic(hall)*** which performs operations accordingly.

sub

Similar to the add function, except hall_5 is used here instead of hall_2.

mul

Similar to the add function, except hall_3 is used here instead of hall_2.

div

Similar to the add function, except hall_12 is used here instead of hall_2.

mod

We use Dividend = Divisor * Quotient + Remainder for this purpose where dividend is the first input, divisor is the second input and the quotient is found by using hall_12 on the given values. Again, rm and kd functions are used for shifting the pointers and mt functions are used to assign values to the locations, in the ***modulo()*** function.

pop

Popping is done by assigning EOS literal to the top of stack, changing the value at mem_3 to 0 and then shifting all pointers to the left by 1 so that mem_2 points to the new top of stack.

output

iit_gate_out_1 or 2 is used appropriately and then pop is implemented. This is achieved via the ***output(p)*** and ***pop()*** functions respectively.

input

Here we first shift the pointers to the right by 1 (using ***updatePositionsBeforeInsert()***) and then we take the input by using iit_gate_in_2 which puts it on top of stack (at mem_2).

This is achieved via the ***takeInput(p)*** function

outputascii

We have used the nankari_gate_out_2 function in the IITK-Traveler code followed by popping the ascii value from the stack.

swap

In the swap function we have used the mem_2 and mem_3 pointers to store the values and then replace them in the reverse order (basically treating mem_3 as a temporary variable).

cycle

This function is implemented by going on swapping the elements consecutively such that the element at top of stack is sent at the bottom of the stack without changing the order of other elements.

rcycle

Same as the cycle. Only difference is that swapping is done in the opposite direction such that the element at the bottom of the stack is sent to the top of the stack.

dup

Duplication is done by making another copy of the number and placing it at the top of the stack. Pointers are shifted to the right by 1 appropriately.

rev

The importance of maintaining the stack between two EOS is highlighted in the implementation of this function. We first make a copy of the given stack right after the EOS in reverse order and then we assign the values of this reversed stack to the original stack. Thus the stack is reversed.

inputascii

This function is implemented in a similar fashion as that of reverse.

if

If is used at the beginning of a loop. On encountering ***if***, the current value of cond is stored in a stack loop_history and the value of cond is increased, keeping the value pushed in loop_history reserved for looping back.

fi

fi is used at the end of a loop. The value of cond is changed to the value on top of the stack loop_history and the loop is completed. On exiting the loop, the value of cond should be changed to something greater than the initial value of cond (the value just before encountering fi) to avoid intersection in the graph.

quit

We set the last location of the program to finish to implement this. The function ***addFinishLocation()*** is called for the same.

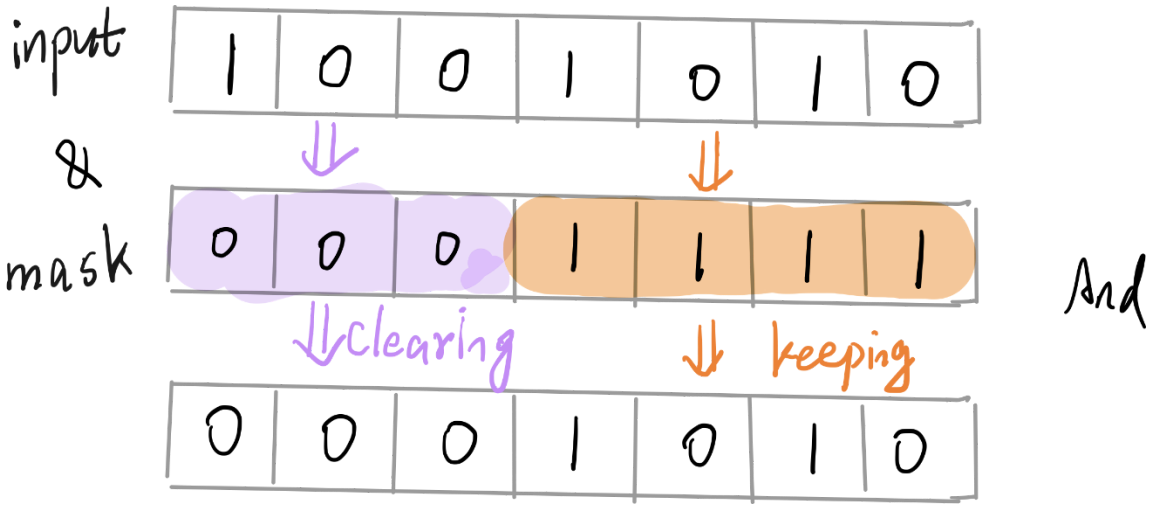
debug

Here, again the importance of maintaining the stack between two EOS literals is highlighted. iit_gate_out_2 is used from the EOS at the bottom of the stack to the EOS at the top of the stack.

push

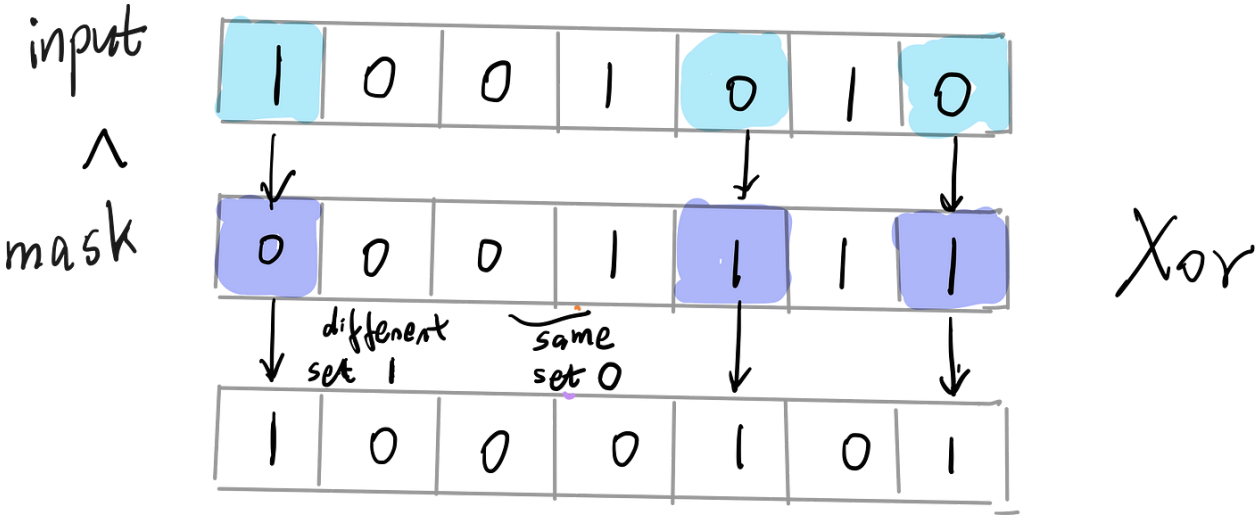
In the ***push(n)*** function, we first shift the pointers to the right by 1, initialize mem_2 to 0 (using ***updatePositionsBeforeInsert()***) and then we increment the value at mem_2 successively for **n** times

Advanced Functions



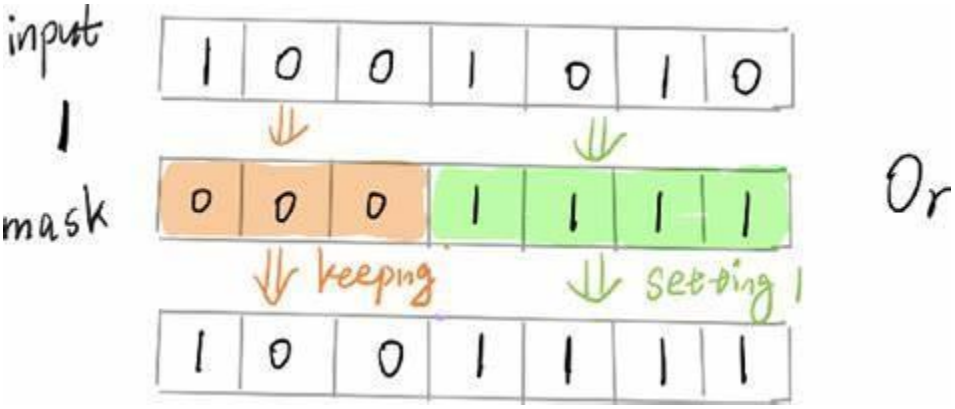
And

First we convert the two given numbers into binary and store them into the memory lane. Then, we multiply each bit of the first number with the corresponding bit of the second number and update it at the corresponding positions of the first number. And then we convert the resultant binary number into decimal in a similar way as discussed in the optimization of push function.



Xor

This function is also implemented in a similar way as that of ***and*** function but instead of multiplying individual bits of the function as done in ***and*** we check whether the two bits are equal or not using the lecture hall location and if they are equal we assign 0 else 1.



Or

Again this function is similar to the ***and*** one. But here we use the formula **x+y-x*y** to calculate each bit of the resultant value.

Not

For this function we use the ***signed not* implementation of the C++ language.
Here we first generate a -1 in the stack by using southern_labs_2 and then we multiply the input with -1.Finally we subtract the number with -1 to get the required ***not*** value.

Nand

Implemented simply by applying ***and*** then ***not***.

Nor

Implemented simply by applying ***or*** then ***not***.

Optimisations

I. Pushing numbers into stack

Pushing numbers into the stack can be a tedious process since implementing a loop in O(n) line complexity will lead to a large number of lines just for achieving a value.

Instead breaking that particular number into its binary format and multiplying the corresponding indexes with the formula (2^index)*(value of bit) hence the line complexity will lead to a shortened line complexity of O(logn). Which is incredibly useful for a set of larger numbers.

For example:-

Push Optimization

Number	Push	Push Optimised
33	85	91
30	79	85
50	119	93
100	219	103
10000	20019	177