Traveler's Transpiler(A6)

Hall 5 - Team 2

Talin Gupta (211095)

Havi Bohra (210429)

Nevish Pathe (220757)

Ashish Ahuja (220235)

Introduction

In IITK Traveler, there are 56 places with linked features. Super Stack has 26 instructions. The following set of functions, together with information on how they were implemented, were used to create the associated mapping.

The Bitwise forms of the Logical Operations (and, or, not, xor, and nand) have been transformed.

Python language has been used for the transpiler.

Development OS : Linux

To run the code, your stack code should be in a file ‘path.txt’

Implementation Details

IITK-Traveler is basically made up of 3 pointers called mem_1, mem_2, and mem _ 3 that store data in an infinite memory lane. We have created some algorithms that use this memory lane as a stack to store data and perform various operations on it, just like in the super stack language.

So, in order to use this memory lane as a stack, the three pointers are arranged as follows:

1. **mem_1**: Indicates the element that comes before the stack's top.
2. **mem_2**: Indicates the top-most element in the stack.
3. **mem_3**: The stack's final element.

In the beginning, mem_2 and mem_3 each have an EOS character. These special characters will be used to implement the stack behaviour in the language. Between these two EOS literals, we keep the stack alive.

The **initialise**() function is used for this.

We've created three global variables:

1. **current position**:- The original value for the present position is "start" (the initial point of the IITK-Traveler code). To connect the incoming code with the outgoing one, we keep updating the current position.
2. **cond** :The IITK-Traveler algorithm keeps track of this conditional variable. This variable must be updated regularly.
3. **loop history**: We have configured the datatype of the loop history to be a stack so that we may get the most recent instance of the loop launched. Loop history maintains a record of the conditional variables used in the loops.

In our code, we increase the conditional variable to make sure that we don't end up in two distinct locations for the same condition value**.**

Basic Feature Implementation

123

In the IITK-traveler language, a random integer specified in the super stack code is to be kept on top of the stack (memory lane).

We use oat stairs 2 N times to put the random integer N onto the stack.

add

We do this by using hall_2. IIT gate in 1 and 2 are used to accept the input, and the output is placed on top of the stack by shifting the pointers and erasing the numbers with the aid of hall 13 3, rm 1, kd 1, etc. A master function called arithmetic(hall) has been built and executes operations as necessary.

sub

Equivalent to the add function, although here hall 5 is used rather than hall 2.

mul

Equivalent to the add function, although here hall 3 is used rather than hall 2.

div

Equivalent to the add function, although here hall 12 is used rather than hall 2.

mod

Dividend is the first input, divisor is the second, and the quotient is determined by applying hall 12 to the supplied numbers in the formula **Dividend** = **Divisor** * **Quotient** + **Remainder.** The modulo() function once again uses the rm and kd functions to shift the pointers and the mt functions to assign values to the places.

pop

Changing the value at mem 3 to 0 and moving all pointers to the left by 1 are the steps involved in popping, which causes mem 2 to refer to the new top of the stack.

output

Pop is implemented after the proper usage of iit gate out 1 or 2. The methods output(p) and pop() are used to do this.

input

Here, we use **updatePositionsBeforeInsert**() to first move the pointers to the right by 1 before taking the input via iit gate in 2, which places it at the top of the stack (at mem 2).

By using the **get_Input**(**p**) function, this is accomplished.

`outputascii`

In the IITK-Traveler code, the nankari gate out 2 function was invoked before popping the ascii value off the stack.

`swap`

To save the data and later replace them in the opposite order, we utilised the pointers mem 2 and mem 3 in the swap method (essentially treating mem 3 as a temporary variable).

`cycle`

In order to implement this function, items are sequentially swapped such that the one at the top of the stack is delivered at the bottom without affecting the order of the other elements.

`rcycle`

The cycle is the same. The only difference is that swapping is carried out in the reverse order, sending the element at the bottom of the stack to the top.

`dup`

Duplication is accomplished by producing a second duplicate of the number and stacking it on top. Pointers are suitably relocated by 1 to the right.

`rev`

The execution of this function emphasises the need of preserving the stack between two EOS. Before assigning the values from this reversed stack to the original stack, we first build a duplicate of the supplied stack immediately after the EOS. The stack is thus turned around.

`inputascii`

Similar to how reverse works, this function is also implemented.

`if`

An if statement is used to start a loop. When an if statement is encountered, the value of cond is incremented and the current value is placed in a stack called loop history. The value put in loop history is kept as a reserve for looping back.

`fi`

At the conclusion of a loop, fi is used. The loop is finished when the value of cond is updated to the value at the top of the stack loop history. To prevent an intersection in the graph, the value of cond should be increased after the loop is finished to a value larger than its starting value (the value shortly before running into fi).

`quit`

To put this into practice, we choose the program's final destination. For the same, **end_graph**() is used.

`debug`

Again, this emphasises how crucial it is to keep the stack between two EOS literals. From the EOS at the bottom of the stack to the EOS at the top of the stack, iit gate out 2 is used.

`push`

In the push(n) method, we first move the pointers to the right by 1, initialise mem 2 to 0, and then we sequentially increase its value at mem 2 for n times (using prep_insert()).

Advanced Featured 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

Making use of a temporary variable and lecture_hall_eq to compare the bits.

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

Applying ***and*** followed by ***not***.

Nor

Applying ***or*** followed by ***not***.