CS 351 Project 4 Assembly Silos (Due 4-13-23)

Project Overview

In this project you are tasked with creating so-called *Assembly Silos*. Each silo acts as an independent **virtual machine** running a simple assembly language. Each silo must run on its own thread and must communicate with other silos through a transfer region. Each silo must remain in sync with every other silo, that is each silo runs exactly 1 instruction then waits for every other silo to finish their current instruction before moving onto the next instruction. Each silo has a single accumulator **register** and a backup register.

Architecture Overview

Each silo contains a program which is written in the language described below. When the last instruction of a program is executed execution should automatically continue from the first instruction of the program. Effectively the entire program is always wrapped in a while true loop.

When a silo writes to a port it must wait for the silo on the other side to read this value before it can carry on. The same goes for attempting to read a value. For synchronization purposes a silo which is waiting to read/write from a port should be considered idle and other silos do not need to wait on it until it.

Your silos should be artificially limited to 1 instruction per second in order to facilitate grading.

There will be any number of input/output streams of data. Input streams are to be read from by a silo they are adjacent to and output streams are to be written to by a silo they are adjacent to.

Assembly Language Overview

NOTE: Everything must be case-sensitive in your implementation

[SRC] and [DST] are parameters which are either a port or register. Ports are how silos communicate between each other. Each silo has exactly 4 ports:

- UP - communicate with the silo above
- RIGHT - communicate with the silo to the right
- DOWN - communicate with the silo below
- LEFT - communicate with the silo to the left

Registers are how a silo can store a piece of data. Each silo has the following registers:

- ACC - accumulator register which may be written to directly, holds an int
- BAK - temporary storage for value from ACC, cannot be written to directly. instead the *SAVE* command must be invoked which copies the value from ACC into BAK.

- NIL - a *virtual* register which when written has no effect and when reading from produces 0.

In addition to the above [SRC] may also be an integer literal.

[LABEL] means text is given which must be a label defined elsewhere in the program.

Syntax

- Labels
  - Syntax = *:[LABEL]:*
  - Labels are used to mark a line of code to jump to. When jumped to the instruction following the label is executed.
- NOOP
  - Syntax = *NOOP*
  - NOOP is simply an instruction which does nothing
- MOVE
  - Syntax = *MOVE [SRC] [DST]*
  - Read [SRC] and write the result to [DST]
  - Examples:
    - *MOVE 10 ACC*
    - *MOVE LEFT RIGHT*
    - *MOVE DOWN NIL*
- SWAP
  - Syntax = *SWAP*
  - Switch the value of the *ACC* register with the value of the *BAK* register
- SAVE
  - Syntax = *SAVE*
  - Write the value from the *ACC* register onto the *BAK* register
- ADD
  - Syntax = *ADD [SRC]*
  - The value of [SRC] is added to the value in the *ACC* register
  - Examples:
    - *ADD 10*
    - *ADD UP*
- SUB
  - Syntax = *SUB [SRC]*
  - The value of [SRC] is subtracted for the value in the *ACC* register
  - Examples:
    - *SUB 11*
    - *SUB DOWN*
- NEGATE
  - Syntax = *NEGATE*
  - The value of the register *ACC* is negated, zero remains zero
- JUMP

  - Syntax = *JUMP [LABEL]*
  - Jumps control of the program to the instruction following the given [LABEL]
- JEZ
  - Syntax = *JEZ [LABEL]*
  - Jumps control of the program to the instruction following the given [LABEL] if the value in the register *ACC* is equal to zero
- JNZ
  - Syntax = *JNZ [LABEL]*
  - Jumps control of the program to the instruction following the given [LABEL] if the value in the register *ACC* is not equal to zero
- JGZ
  - Syntax = *JGZ [LABEL]*
  - Jumps control of the program to the instruction following the given [LABEL] if the value in the register *ACC* is greater than zero
- JLZ
  - Syntax = *JLZ [LABEL]*
  - Jumps control of the program to the instruction following the given [LABEL] if the value in the register *ACC* is less than zero
- JRO
  - Syntax = *JRO [SRC]*
  - Jumps control of the program to the instruction specified by the offset which is the value contained within [SRC].
  - If the offset is greater than the size of the program then you should wrap around to the beginning of the program until you get to an offset which is contained within the program. The same goes for negative offsets.
  - Examples:
    * *JRO 0* - executes the current instruction next, infinite loop
    * *JRO -1* - executes the previous instruction next
    * *JRO 2* - executes the instruction after the next instruction
    * *JRO ACC* - the value of the register *ACC* determines the next instruction

Project Requirements

- Your GUI must be written in JavaFX.
- You must have a text area where the user can input a program
- You must display the current value of the *ACC* and *BAK* registers
- You must display values which are currently being transferred between silos
- You must have a start button which starts execution of each silo's code
- You must have a pause/step button. This button should display as pause while the code is running and step while the code is halted. When clicked in pause mode the execution of the code is halted. When clicked in step mode then 1 instruction from each silo must be executed then execution is

halted again.
- You must have a stop button which halts execution and resets all the silos.
- You must have an area which shows the current value of each input stream and all previous values written to an output stream
- Each *interpreter* must run on its own thread
- Each *transfer region* must run on its own thread
- Initial state of the game must be constructible using the file format given below

Program Inputs

Your program must ask user input through the command line in order to construct the initial state of the program. The input will have the following format:

```
[numRows] [numCols]
[instruction]
[instruction]
.
.
.
END
[instruction]
[instruction]
.
.
.
END
.
.
.
END
INPUT
[rowNum] [rowCol]
[inputNum]
[inputNum]
.
.
.
END
OUTPUT
[rowNum] [rowCol]
END
```

The first line tells you how many rows and columns the grid is going to have. The subsequent lines after that are the instructions which must be loaded into the first silo. Note the silos will be given row by row. Then when you see *END* that means the instructions for that silo are done. You then move onto the next

silo and so on and so forth. Next you must read in the input/output streams. An input stream will follow the keyword *INPUT* and will then be followed the coordinates of the stream. Note these coordinates will have 1 out of bound coordinate. This is to signify that is not by of the normal silo grid but rather adjacent to it. In the example above the input stream would go above the silo grid and over the second column. The values of the stream are given after these coordinates. An output stream will follow the keyword *OUTPUT* and only coordinates will be given. Both stream will end with *END*. You can assume that this input will be well-formed.

Suggestions

- Start by writing a *parser*
  - A *parser* takes in the raw text which represents the silo's program and transforms it into a structured form in your code. This structure can come in the form of *interfaces*, *classes*, etc. This structure which can be referred to as the *AST* or abstract syntax tree is then used by your *interpreter* to actually execute the code.
- Next write an *interpreter*
  - An *interpreter* handles the execution of your internal *AST*.
  - The interpreter in essence represents a silo logically.
  - It must have an *ACC* and a *BAK* field
  - It must have access to a way to transfer to the 4 ports
  - It must have a field to keep track of the current instruction to be executed
  - You can then test this independently to make sure the logic of your language implementation is sound before moving onto the concurrency and GUI part of the project.
- Then move onto how you are going to represent each silo visually in JavaFX.
- Finally, put it all together and tackle how you are going to handle threading.