Flip-Flop Evaluator

Technical Guide

Caquilala, Cedric

Famadico, Nichol John

Overview

Flip-flops are formed using logic gates, which are in turn made of transistors. Flip-flops are basic building blocks in the memory of electronic devices. Each flip-flop can store one bit of data. There are four basic types of flip flops: D, T, RS, and JK. Each of them works differently and have their own characteristic table.

Hence, this application simulates the characteristic tables of the four basic types of flipflop. It is written using Java; thus, a Java Virtual Machine is needed so that it can run on any device. It has three source files: *Main.java, BooleanPostfix.java,* and *InfixToPostFix.java.* The procedures are divided into 3 parts: input, process, and output. Each of these processes will be described below.

Part I – Input

The program has static global variables that will store the input of the user at the initial process of the application. Displayed in the table below are the variables that will be used for that matter.

|  |  |  |
| --- | --- | --- |
| **Type** | **Name** | **Description** |
| String | Type | Stores the input regarding about what type of flip-flop is involved |
| Integer | noFF | Stores the number of flip-flops involved |
| Integer | noIn | Stores the number of input variables involved |
| Integer | noOut | Stores the number output variables |
| ArrayList<String> | eqOut | Stores the Boolean function for output variable. |
| ArrayList<String> | eqNext | Stores the Boolean input functions for the flip-flops. |

After asking inputs from the user, the program then iterates and scans the input Boolean equations to collect the flip-flop variables or the uppercase variables and stores them to a container *ff* (ArrayList<String>). It also concurrently collects the input variables or the lowercase variables and stores them to a container *in* (ArrrayList<String>). Afterwards, the program calls *Collection.sort()* to sort the contents of the containers alphabetically.

Moving on, the program creates an array of *Row* objects with size 2 raise to the sum of *noFF* and *noIn. Row* objects are the minterms that will be evaluated on the next part of the process. But before proceeding to that part, here is the table giving details about the fields and methods static *Row* class.

Row Class

|  |  |  |
| --- | --- | --- |
| **Type/Return Type** | **Fields/Methods** | **Description** |
| String | Type | Stores what type of flip-flop is involved. |
| int[] | FFCurr | Stores the present state of the flip-flop |
| int[] | Input | Stores the possible values of the input variables |
| int[] | FFIn | Stores the evaluated values of the flip-flop input Boolean functions |
| String[] | FFNext | Stores the next state of the flip-flop |
| int[] | output | Stores the evaluated values of the output Boolean function. |
| Constructor | Row(String type, int noFF, int noIn, int noOut) | Parameters:  - String type  - int noFF  - int noIn  - int noOut  Creates an instance of the class and initializes field values. |
| void | setInputsFromDecimal(int x) | Parameters:  - int x  Initializes the present state and the possible values of the input variables in a form of minterm. |
| void | setFFNext(int index, int value) | Parameters:  - int index  - int value  Sets the value of field *FFNext* |
| void | print() | Prints the row |

Part II – Process

The process starts when an array of *Row* objects named *table* (Row[]) is initialized. The program iterates the array and instantiate individual *Row* objects using the global variables *type, noFF, noIn, noOut* as parameters for the class constructor. The object method *setInputsFromDecimal* is then called to initialize the minterm values of the flip-flop variables and the input variables. The former is treated as more significant variables than the latter. The instanstiated object will be referenced as one of the elements of the array *table*.

The process differs slightly depending on what type of flip-flop was inputted, but the initial parts of the process are mainly the same. First the program iterates to the container *eqNext* and process each Boolean function independently. The Boolean function is filtered with the “=” and anything on its left is removed. The variables are also collected and sorted. Afterwards, the program creates a HashMap *vars* (HashMap<String, Integer>), which maps the values of the Boolean literals to 1’s and 0’s depending on the minterm assigned to the row object.

The program then instantiates an *InfixToPostFix* object to convert the the Boolean function to postfix format so that it can be evaluated accordingly. It then instantiates a *BooleanPostfix* object to evaluate the Boolean function. The answer is then stored to the *FFIn* field of the *row* object.

Now that the equations are set up into ArrayLists<String>, these expressions are evaluated based on the values of the present states of the flip-flops and the inputs. These will be used to solve for the binary values of each part of the circuit for that specific state.

A. InfixToPostfix

Since the user’s input is an expression, the program must capable of understanding how to process Boolean algebra. However, this entails precedence for operators such as AND, OR, XOR, and parentheses. For example, in the expression A+Bx, Bx needs to be evaluated first before using the OR operator on A. To to do this, it is necessary to convert the user’s expressions into *postfix*.

A *postfix* expression is a mathematical expression where the operators are placed after its operands. Infix, on the other hand, is the pattern humans follow in writing math, where the operators are between the operands. For example, the expression A+B is in infix, while its postfix counterpart is A B +. This is very useful for computing since it is simpler for computers to read.

For the purposes of the program, an InfixToPostfix class was created. It accepts an infix String and returns its postfix counterpart using the convertToPostfix() method. Initially, it will add AND (\*) operators between connected operands such as AB [A\*B] and (A+B)C [(A+B)\*C]. To convert an infix expression into postfix, the stack data structure can be utilized. The following steps can be followed:

1. A stack for operators and a String field for the converted expression is initialized.
2. For each piece of the expression, determine what it is.
   1. If it is an operand, simply append it to the converted field.
   2. If it is an operator, check the top of the stack for precedence (for example, AND takes precedence over OR, meaning it should be done first).
      1. If the new operator is of greater precedence, simply push it into the stack.
      2. If the new operator is of lower precedence, then pop the stack into the converted field until it reaches a field with the same precedence. Then, push the new operator into the stack.
      3. If it is an open parenthesis, simply push it into the stack.
      4. If it is a closing parenthesis, pop the stack into the converted field until it reaches an opening parenthesis. Pop both parenthesis off the stack.
3. Once the end of the expression is reached, pop all remaining operators into the converted field.

InfixToPostfix Class

|  |  |  |
| --- | --- | --- |
| **Type/Return Type** | **Fields/Methods** | **Description** |
| String[] | operators | The list of operators. The following are used:  + - AND  \* - OR  @ - XOR  () - parentheses |
| String | infix | Represents the infix form or the user input. |
| void | convertToPostfix() | The major part of the conversion. It carries out the reading of each token in the infix expression as well as the conditional statements. |
| boolean | isOperator() | Parameters:  - String x  Return value:  Returns if the string x is an operator based on the operators field. |
| int | getPrecedence() | Parameters:  - char x  Return value:  Returns the precedence of char x. Higher precedence means should be done first. |
| boolean | isValidMultiply() | Parameters:  - String ch1  - String ch2  Return value:  Returns if strings ch1 and ch2 are supposed to be multiplied to each other. |

B. BooleanPostfix

After converting the user’s input to postfix notation, the program is now ready to solve the expression. For this purpose, a new BooleanPostfix class is created. Its purpose is to evaluate a postfix expression given a variable and value pair. The postfix expression is the String obtained from the converted infix from the previous step. The variable and value pair is obtained from the HashMap created from the row’s input and present state. For example, the expression A B + given the HashMap {A:0 ; B:1} will return 1.

When the evaluate() method is called, the calculation begins. This process also involves the stack data structure, but instead of having a stack for the operators, it uses a stack for the operands. It converts each operand with its respective value in the HashMap using the getVarValue() method and pushes it into the stack. It solves through the steps:

1. A stack for the operands and an initial value of 0 is initialized.
2. For each part of the postfix expression, determine what it is.
   1. If it is an operand, get its value and push it into the stack.
   2. If it is a complement sign [‘], pop the top value of the stack and get its complement (0 -> 1 or 1->0).
   3. If it is an operator, pop the top two values of the stack and perform the operation. Push the resulting value to the stack.
3. Once the end of the expression is reached, there should be only one operand remaining in the stack. Pop the top of the stack. This value is now the calculated value.

BooleanPostfix Class

|  |  |  |
| --- | --- | --- |
| **Type/Return Type** | **Fields/Methods** | **Description** |
| String | postfix | Represents the postfix form or the user input. |
| int | value | The calculated value of the expression. |
| HashMap<String,Integer> | vars | Contains the variable:value pairs. It serves as the dictionary for the class. |
| void | setValue() | Sets the calculated value. |
| int | getValue() | Return value:  Returns the calculated value. |
| int | evaluate() | Return value:  Solves the expression and returns the result of the evaluation. |
| int | getVarValue() | Parameters:  - String var  Return value:  Returns the respective value of var based on the HashMap vars. |
| boolean | isOperator() | Parameters:  - String x  Return value:  Determines if x is an operator (+ \* @) |
| int | negate() | Parameters:  - int x  Return value:  Returns the complement of x. |
| int | operate() | Parameters:  - int a  - int b  - String operator  Return value:  Solves a and b using the operator given and returns the resulting value. |

After the *FFIn* values has been set, the application will now evaluate the *FFNext* values using the characteristic table logically programmed into it. It will scan the respective *FFIn* values to know what the value of the FFNext should be. The logic is simply programmed using if-else conditionals. The Boolean function for the output variable is also evaluated using the same manner.

Part III – Output

When the iteration has been finished, the static method *printable(Row[] table)* is called to print the table describing the nature of the flip-flop.