

Circuit Glitch Finder

Sean McNamee

Nick Passaretti

Dr. Nizich

12/17/19

**Problem Statement:**

Timing delays are very important logic systems and gate circuits. Knowing Propagation delay, the time it takes for a signal to process through a logic gate, Contamination delay, the minimum amount of time from when an input changes until any output starts to change its value, and glitches, when signal delays cause an unwanted output, we can design our gate arrays and circuits to be more efficient and consistent.

The inability to locate glitches in a system hinders logic engineers to figure out glitches in a system due to propagation delay.

These delays cause inefficiency in performance, speed, and reliability. This will help logic engineers to plan and design their systems with more efficiency and less time by using these time delays to test the different time outputs and analyzing it for glitches and locating them.

**Partner Contributions:**

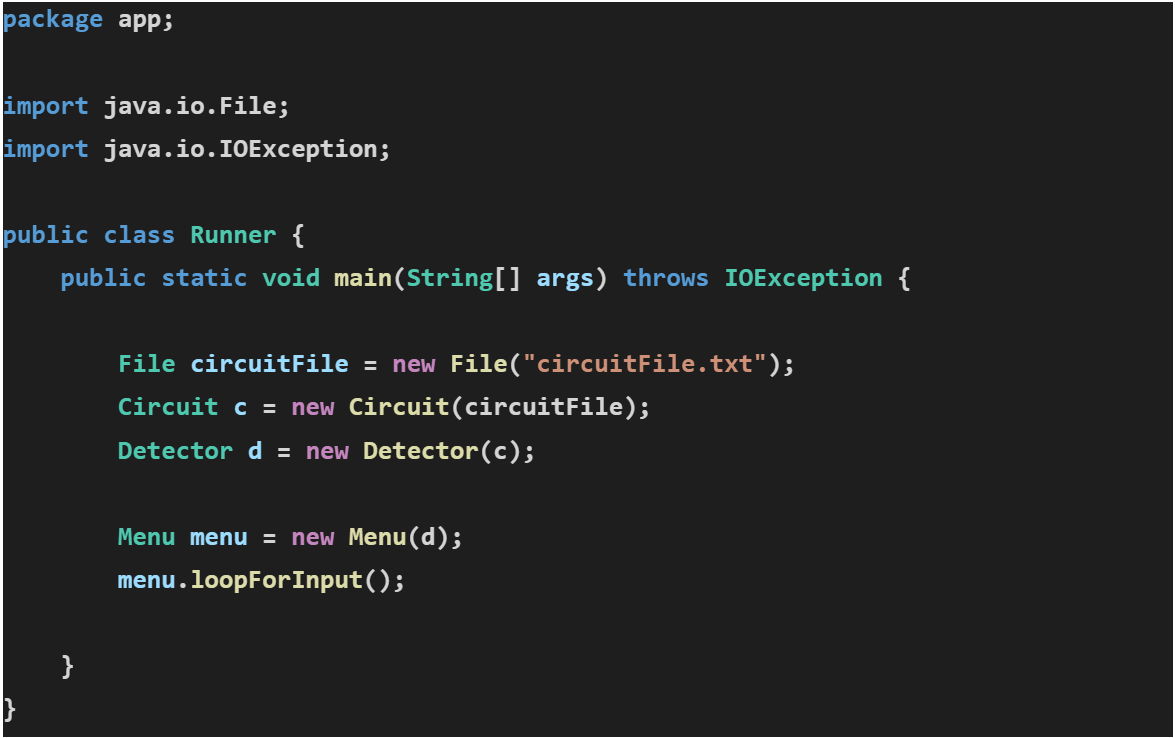
When we were deciding on what to do Sean proposed a rather complex challenge. We knew he had the knowledge to headline the code creation. Sean was the main coder, focusing on the main classes and functions, with Nick giving suggestions on methods, procedures and writing the report.

\*\*Nicks comments on Sean: Sean is a genius… he seems to know everything about Java and how to articulate his ideas. Sometimes he can come up with too many ways to do things and has to pick between 5 ideas.. It was almost clear cut which one to pick but he would always ask what I prefer and we would make sense of it.

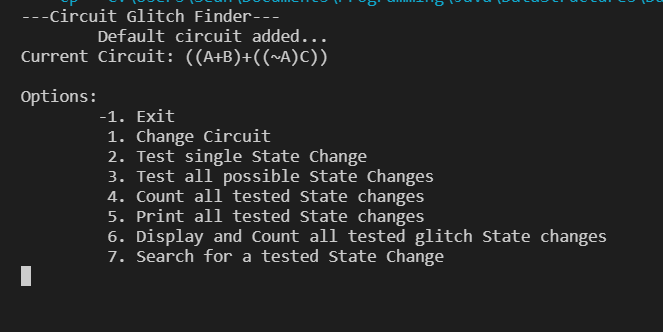
\*\*Seans comments on Nick: Although Nick wasn’t as proficient as me in programming, he could easily unravel whatever concept I proposed. In fact, he came up with a lot of cool tricks that we used to figure out where the glitches were. His insight as an electrical engineer was essential for our circuit-based project’s success.

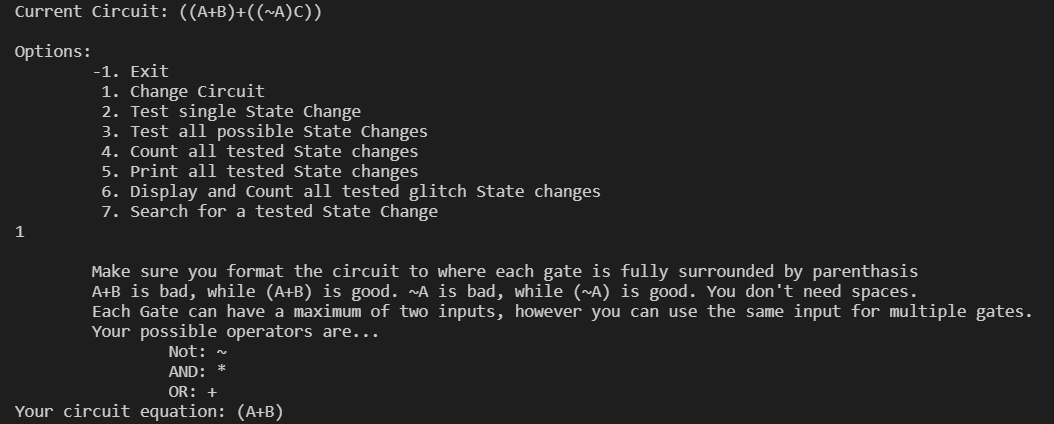
**Flow Chart:**

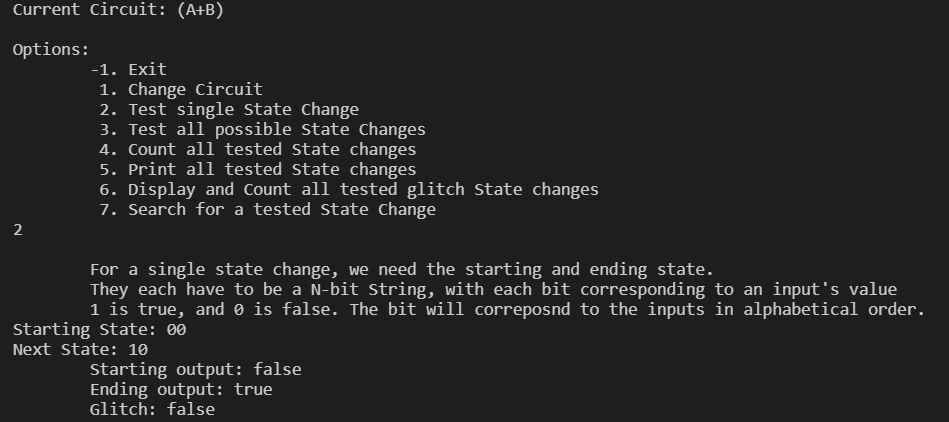
**Code:**

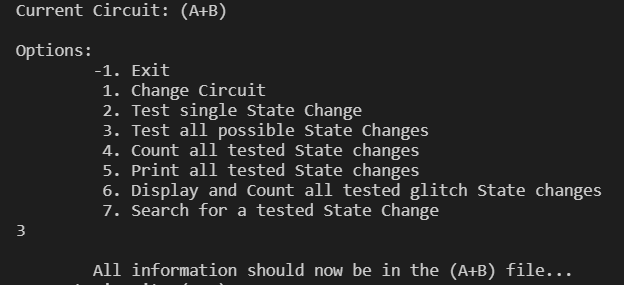
**Main Code (Runner Class):**

**Menu Pictures:**

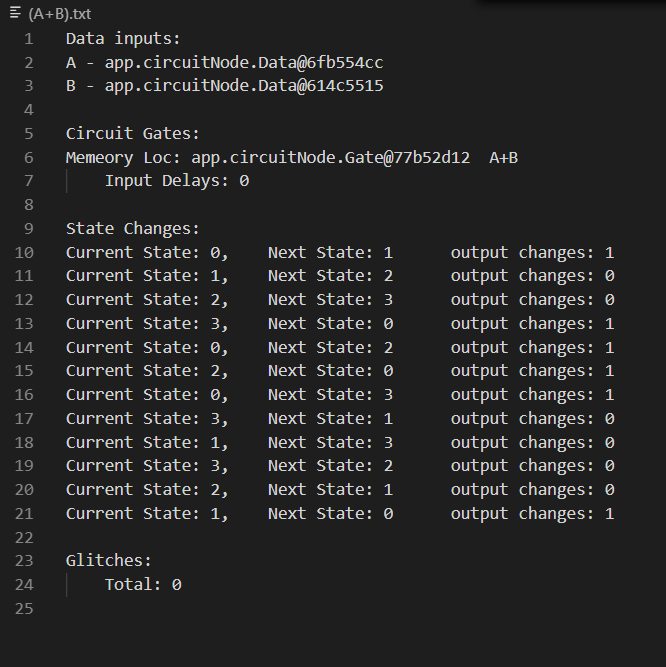
**Main menu:**

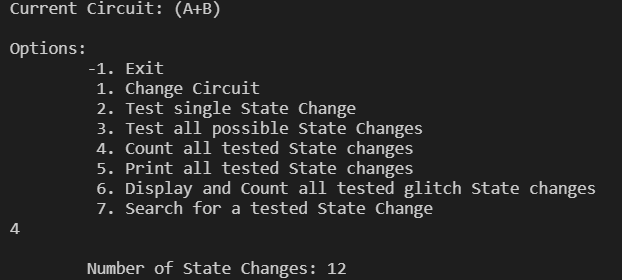
**Menu 1:**

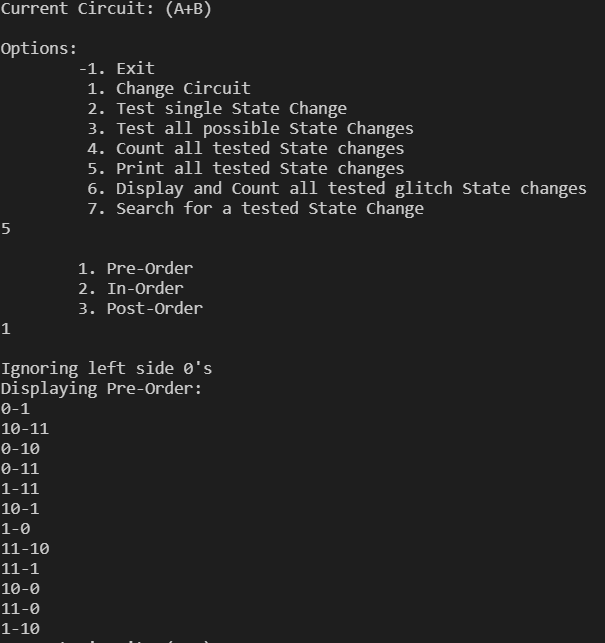
**Menu 2: **

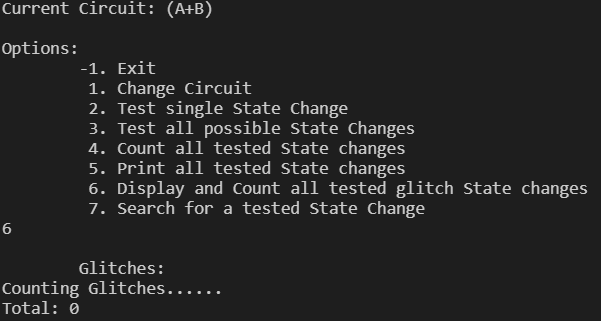
**Menu 3: **

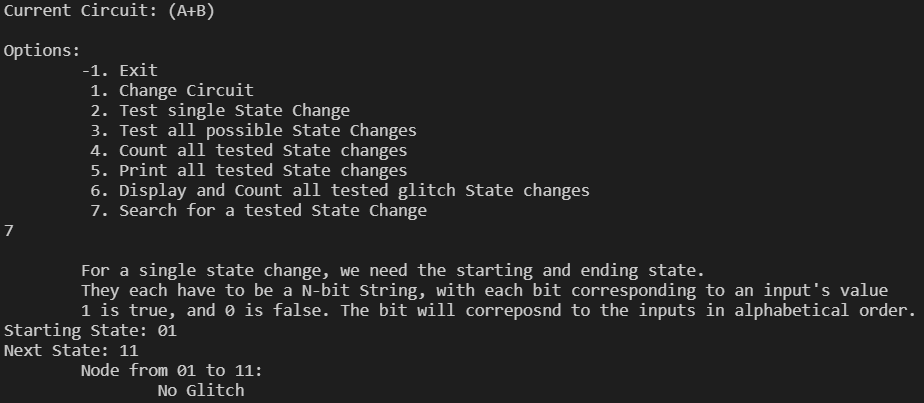
**Menu 3 display:**

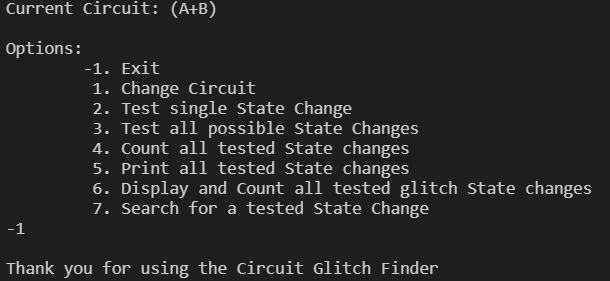


**Menu 4: **

**Menu 5: **

**Menu 6: **

**Menu 7: **

**Menu: -1**

**Class Pictures:**

1. **Runner Class-**

**The Runner class has an instance of the Detector and Menu classes**

**Pic**

1. **Menu Class-**

**The Menu class holds an instance of the Detector class. It can choose any action:**

1. Test single state change
   1. Take input for state change
   2. Use the detector’s testStateChange(start, start, null, null) to initialize the state
   3. Take the boolean of testStateChange(start, start, tree, null)
      1. Initializes the time, outputSwitch times and outputChangeCount (all to 0)
      2. Prepare gates for the loop
         1. Set the circuit’s inputs to next (circuit.setInputs(next))
         2. Initialize the queues of the gates
         3. Sort the gates by their queue times
      3. For each time,
         1. if a gate’s next delayInput time is =
            1. Update an internal Node, remove from delayInput Queue, and add to the delayOutput Queue
         2. If a gate’s next delayOutput time is =
            1. Update Gate’s value and remove from delayOutput Queue
         3. Check if the output changed. If it did
            1. Increment outputChangeCount and possibly the outputSwitch times
      4. Check if outputChangeCount >1 and the circuit output overall changed. If it did
         1. Make a glitch and mark down the glitch times
      5. Add the StateChange to the binary tree tree.insert(transition)
   4. If true, display that there was a glitch on the state change
2. **Test all state changes**
   1. Create many queues to represent all the state changes
      1. Each has numbers 1 to 2^N-1 (in that order)
   2. Open the file for the circuit
   3. For each state change, call testStateChange(current, next, tree, fileWriter)
      1. With the fileWriter, it will note transitions and if they’re glitches.
   4. Summarize glitches in the file tree.writeGlitchNodes(writer)
   5. Close the file
3. **Search for any single state change**
   1. Take input for state change
   2. Use the detector’s tree’s search(start, end) method (returns a node)
      1. The search is recursive on a binary search tree.
   3. If null, show not found
   4. Otherwise display the state change of the node
      1. If glitch is null, show no glitch
      2. Otherwise display the glitch’s start-end times.
4. **Count / show all glitches**
   1. Use the Detector’s BTree’s countGlitchNodes method (detector.getTree().countGlitchNodes())
      1. Calls the countGlitchNodes(BtNode node), sending in the tree’s root
         1. Recursively counts and displays glitch nodes
            1. Base case: node is null
            2. Recursive case: set count to 0

If a glitch, add 1 to count and display node info.

node.getStartValues()

node.getEndValues()

node.getGlitch.getStart()

node.getGlitch.getEnd()

Either way, return count+returns using the left and right nodes

1. **Count all state changes**
   1. Use Detector’s BTree’s countNodes method (detector.getTree().countNodes())
      1. Calls countNodes(BTNode node), sending in the tree’s root.
         1. Recursively counts all nodes
            1. Base case: node is null. Return 0
            2. Recursive case. Node not null. Returns 1+returns using the left and right nodes.
2. **Show all state changes**
   1. Give the selection between pre-order, in-order, and post-order
      1. Use Detector’s BTree’s selected traversal method (recursive)
         1. detector.getTree().preorder()
         2. detector.getTree().inorder()
         3. detector.getTree().postorder()
3. **Upload new circuits -** Stack, Array,
   1. Instantiate new Circuit Object (constructor)
      1. Runs equationToCircuit()
         1. Instantiate Circuit’s Gate Array and Hashtable<Character, Data>
         2. Creates temporary Stack for storing CircuitNodes
         3. Loop through each character of the equation to fill the Gate array and Hashtable
            1. For an operator, add an ‘inputless’ gate to the stack
            2. For a letter, Check if letter already in the hashtable

Is: Add that associated data to the stack

Not: Create new Data, add it to the stack and hashtable

* + - * 1. For a close parenthesis, take previous two off the stack.

If the most recent is a + or \*

Take the next off the stack as well

Use what you took off to create a new gate

///////////Create Gate instance

Uses MyLinkedList

Add the gate to the stack and Gate array

* + - 1. Set the last element of the Gate Array to be the circuit’s output Gate
    1. Sets the circuit name (for user printing/file printing
    2. Run writeCircuit()
       1. Creates a new file for this particular circuit
       2. Run writeData() (information about inputs (Data objects)
       3. Run writeGates() (information about gates (Gate objects)

**package app;**

**import java.io.IOException;**

**import java.util.Scanner;**

**import app.tree.BTNode;**

**/\*\***

**\***

**\*/**

**public class Menu {**

**private Detector detector;**

**private boolean continueLoop = true;**

**private final int EXIT = -1;**

**public Menu(Detector d) {**

**this.detector = d;**

**}**

**public void loopForInput() throws IOException {**

**Scanner scan = new Scanner(System.in);**

**System.out.println("---Circuit Glitch Finder---");**

**System.out.println("\tDefault circuit added...");**

**while (continueLoop) {**

**listOptions();**

**int selection = loopUntilValid(scan, 1, 7);**

**switch(selection) {**

**case 1://Change Circuit**

**explainCircuit();**

**System.out.print("Your circuit equation: ");**

**String equation = scan.nextLine();**

**detector.setCircuit(new Circuit(equation));**

**break;**

**case 2://Test single State Change**

**case 7://Search for single State Change**

**//input**

**explainStateChange();**

**System.out.print("Starting State: ");**

**String strStartState = scan.nextLine();**

**System.out.print("Next State: ");**

**String strNextState = scan.nextLine();**

**int startState = Integer.parseInt(strStartState);**

**int nextState = Integer.parseInt(strNextState);**

**if (selection == 2) {**

**//output/calculation**

**detector.testStateChange(startState, startState, null, null); //Load initial state**

**System.out.println("\tStarting output: " + detector.getCircuit().getCircuitOutput());**

**boolean isGlitch = detector.testStateChange(startState, nextState, detector.getTree(), null);**

**System.out.println("\tEnding output: " + detector.getCircuit().getCircuitOutput());**

**System.out.println("\tGlitch: " + isGlitch);**

**} else { //7**

**BTNode node = detector.getTree().search(startState, nextState);**

**if (node != null) {**

**System.out.println("\tNode from " + strStartState + " to " + strNextState + ":");**

**if (node.getGlitch() != null) {**

**System.out.println("\t\tGlitch from t=" + node.getGlitch().getStart() + " to " + node.getGlitch().getEnd());**

**} else {**

**System.out.println("\t\tNo Glitch");**

**}**

**} else {**

**System.out.println("\tState change not found");**

**}**

**}**

**break;**

**case 3://Test all possible State Changes**

**System.out.println("\tAll information should now be in the " + detector.getCircuit().getCircuitName() + " file...");**

**detector.testAllStateChanges();**

**break;**

**case 4://Count all tested State changes**

**System.out.println("\tNumber of State Changes: " + detector.getTree().countNodes());**

**break;**

**case 5://Print all tested State changes**

**listPrintOrders();**

**int printType = loopUntilValid(scan, 1, 7);**

**System.out.println("Ignoring left side 0's");**

**System.out.print("Displaying ");**

**switch(printType) {**

**case 1://Pre-Order**

**System.out.println("Pre-Order:");**

**detector.getTree().preorder();**

**break;**

**case 2://In-Order**

**System.out.println("In-Order:");**

**detector.getTree().inorder();**

**break;**

**case 3://Post-Order**

**System.out.println("Post-Order:");**

**detector.getTree().postorder();**

**break;**

**default:**

**System.out.println("\tNone selected...");**

**break;**

**}**

**break;**

**case 6://Disply and Count all tested glitch State changes**

**System.out.println("\tGlitches:");**

**int count = detector.getTree().countGlitchNodes();**

**System.out.println("Total: " + count);**

**break;**

**default://Exit**

**this.continueLoop = false;**

**break;**

**}**

**}**

**System.out.println("Thank you for using the Circuit Glitch Finder");**

**}**

**//Error trap for a valid number input**

**private int loopUntilValid(Scanner scan, int lowValid, int highValid) {**

**int ans = 0;**

**do {**

**String userAns = scan.nextLine();**

**ans = Integer.parseInt(userAns);**

**} while ( !(ans == EXIT || (ans >= lowValid && ans <= highValid)) );**

**System.out.println();**

**return ans;**

**}**

**private void listOptions() {**

**System.out.println("Current Circuit: " + detector.getCircuit().getCircuitName());**

**System.out.println("\nOptions:");**

**System.out.println("\t-1. Exit");**

**System.out.println("\t 1. Change Circuit");**

**System.out.println("\t 2. Test single State Change");**

**System.out.println("\t 3. Test all possible State Changes");**

**System.out.println("\t 4. Count all tested State changes");**

**System.out.println("\t 5. Print all tested State changes");**

**System.out.println("\t 6. Display and Count all tested glitch State changes");**

**System.out.println("\t 7. Search for a tested State Change");**

**}**

**private void explainCircuit() {**

**System.out.println("\tMake sure you format the circuit to where each gate is fully surrounded by parenthasis");**

**System.out.println("\tA+B is bad, while (A+B) is good. ~A is bad, while (~A) is good. You don't need spaces.");**

**System.out.println("\tEach Gate can have a maximum of two inputs, however you can use the same input for multiple gates.");**

**System.out.println("\tYour possible operators are...");**

**System.out.println("\t\tNot: ~");**

**System.out.println("\t\tAND: \*");**

**System.out.println("\t\tOR: +");**

**}**

**private void explainStateChange() {**

**System.out.println("\tFor a single state change, we need the starting and ending state.");**

**System.out.println("\tThey each have to be a N-bit String, with each bit corresponding to an input's value");**

**System.out.println("\t1 is true, and 0 is false. The bit will correposnd to the inputs in alphabetical order.");**

**}**

**private void listPrintOrders() {**

**System.out.println("\t1. Pre-Order");**

**System.out.println("\t2. In-Order");**

**System.out.println("\t3. Post-Order");**

**}**

**}**

1. **Detector Class-**

**The Detector class has an instance of the Circuit class and Binary Tree (BTree) class. It can:**

1. **Test single state changes**
2. **Test all possible state changes**

**package app;**

**import java.io.BufferedWriter;**

**import java.io.File;**

**import java.io.FileWriter;**

**import java.io.IOException;**

**import java.util.LinkedList;**

**import java.util.Queue;**

**import app.circuitNode.CircuitNode;**

**import app.circuitNode.Gate;**

**import app.tree.BTNode;**

**import app.tree.BTree;**

**/\*\***

**\* Detector**

**\*/**

**public class Detector {**

**private Circuit circuit;**

**private BTree tree;**

**public Detector(Circuit c) {**

**circuit = c;**

**tree = new BTree();**

**}**

**public Detector(File f) {**

**circuit = new Circuit(f);**

**tree = new BTree();**

**}**

**public Detector(String s) {**

**circuit = new Circuit(s);**

**tree = new BTree();**

**}**

**public void setCircuit(Circuit c) {**

**this.circuit = c;**

**}**

**public Circuit getCircuit() {**

**return this.circuit;**

**}**

**/\*\***

**\* After a lot of trying, we decided queues would be the best way to figure out**

**\* every possible state change in a way where we never had to preload a state.**

**\***

**\* @throws IOException**

**\*/**

**public void testAllStateChanges() throws IOException {**

**// initialize array of queues**

**// A 2 input circuit has 4 possibilities. A 3 input circuit has 8 (2^n)**

**BufferedWriter writer = null;**

**writer = new BufferedWriter(new FileWriter(circuit.getOutputFile(), true));**

**int inputCount = circuit.getInputs().size();**

**int largest = (int) Math.pow(2, inputCount);**

**Queue<Integer>[] transitions = new LinkedList[largest];**

**// each queue will have numbers from 1-Math.pow(2, n)-1**

**for (int i = 0; i < transitions.length; i++) {**

**transitions[i] = new LinkedList<Integer>();**

**for (int t = 1; t < largest; t++) {**

**transitions[i].add(t);**

**}**

**}**

**// initialize full circuit for all false**

**testStateChange(0, 0, null, null);**

**// Loop through 2^n\*((2^n)-1) = 2^2n - 2^n times**

**writer.append("State Changes:\n");**

**int max = (int) (Math.pow(2, 2 \* inputCount) - Math.pow(2, inputCount));**

**int count = 0;**

**int currentState = 0;**

**int nextState = 0;**

**while (count < max) {**

**nextState = (currentState + transitions[currentState].remove()) % largest;**

**writer.append("Current State: " + currentState + ",\t Next State: " + nextState);**

**testStateChange(toBase2(currentState), toBase2(nextState), tree, writer);**

**currentState = nextState;**

**count++;**

**}**

**tree.writeGlitchNodes(writer);**

**writer.close();**

**}**

**private int toBase2(int n) {**

**return Integer.parseInt(Integer.toBinaryString(n));**

**}**

**public BTree getTree() {**

**return tree;**

**}**

**/\*\***

**\* Will calculate the circuit values at every given time Assumes that the**

**\* circuit is initialized with values**

**\***

**\* @param startState base 2 number, where each 1 is true and each 0 is false**

**\* for a specific output (ABCD corresponds to 0000)**

**\* @param endState same layout as startState, but the change to this is what**

**\* is observed**

**\* @param glitchStates a tree representing all the transitions and if they have**

**\* glitches**

**\* @return if the current state change resulted in a glitch**

**\* @throws IOException**

**\*/**

**public boolean testStateChange(int startState, int nextState, BTree glitchStates, BufferedWriter writer)**

**throws IOException {**

**int time = 0; // The first possible time to take inputs would be 0**

**boolean previousValue, outputValue;**

**int earliestSwitch, latestSwitch;**

**int outputChangeCount = 0;**

**previousValue = outputValue = circuit.getCircuitOutput();**

**earliestSwitch = latestSwitch = 0;**

**// Set initial outputValue (from nextState)**

**circuit.setInputs(nextState);**

**circuit.initializeGatesForCheck();**

**circuit.sortByQueues();**

**int circuitEndTime = circuit.circuitEndTime();**

**// While there's still more in the queue**

**while (circuitEndTime >= time) {**

**// Update all gates for this time**

**for (CircuitNode c : circuit.getNodes()) {**

**Gate g = (Gate) c;**

**if (g.getNextOutputTime() == time) {**

**g.updateInternalAndRemove();**

**// System.out.println(" at time " + time);**

**}**

**if (g.getNextInputTime() == time) {**

**g.takeInputsAndTransfer();**

**}**

**}**

**time += 5;**

**// Check for output change**

**if (circuit.getCircuitOutput() != outputValue) {**

**outputChangeCount++;**

**outputValue = !outputValue;**

**// Figure out earliest and latest output switch times**

**if (earliestSwitch == 0) {**

**earliestSwitch = time;**

**}**

**latestSwitch = time;**

**}**

**}**

**if (writer != null) {**

**writer.append("\t\toutput changes: " + outputChangeCount + "\n");**

**}**

**BTNode transition;**

**// Must be both because if an output oscillates but ends up changing, its okay**

**if (previousValue == circuit.getCircuitOutput() && outputChangeCount >= 2) {**

**if (writer != null) {**

**writer.append("\tGlitch from " + startState + " to " + nextState + "\n");**

**}**

**transition = new BTNode(startState, nextState, earliestSwitch, latestSwitch);**

**} else {**

**// System.out.println("ALL CLEAR, No Glitch!");**

**transition = new BTNode(startState, nextState, null);**

**}**

**if (glitchStates != null) {**

**glitchStates.insert(transition);**

**}**

**// Lets the calling function know if this test case was a glitch**

**return (transition.getGlitch() != null);**

**}**

**}**

1. **Binary Tree (BTree) Class-**

**The Binary Tree class can:**

1. **Insert circuits**
2. **Count glitches**
3. **Search**
4. **Transverse**

**package app.tree;**

**import java.io.BufferedWriter;**

**import java.io.IOException;**

**/\* Class BT**

**This ends up being a Binary Search Tree because of the orderedInsert Method\*/**

**public class BTree {**

**private BTNode root;**

**/\* Constructor \*/**

**public BTree() {**

**root = null;**

**}**

**/\* Function to check if tree is empty \*/**

**public boolean isEmpty() {**

**return root == null;**

**}**

**/\* Functions to insert data \*/**

**public void insert(BTNode nodeToInsert) {**

**root = orderedInsert(root, nodeToInsert);**

**}**

**//Returns the Node after inserting the Node in order**

**//Makes this a binary search tree**

**private BTNode orderedInsert(BTNode node, BTNode toInsert) {**

**if (node==null) {**

**node = toInsert;**

**} else if(larger(toInsert, node)) { //Must go to the right**

**node.right = orderedInsert(node.right, toInsert);**

**} else {**

**node.left = orderedInsert(node.left, toInsert);**

**}**

**return node;**

**}**

**private boolean larger(BTNode node1, BTNode node2) {**

**return node1.getStartValues()>node2.getStartValues() || (node1.getStartValues()==node2.getStartValues()&&node1.getEndValues()>node2.getEndValues());**

**}**

**/\* Function to count number of nodes \*/**

**public int countNodes() {**

**return countNodes(root);**

**}**

**/\* Function to count number of nodes recursively \*/**

**private int countNodes(BTNode r) {**

**if (r == null)**

**return 0;**

**else {**

**int count = 1;**

**count += countNodes(r.getLeft());**

**count += countNodes(r.getRight());**

**return count;**

**}**

**}**

**public void writeGlitchNodes(BufferedWriter writer) throws IOException {**

**writer.append("\nGlitches:\n");**

**int count = countGlitchNodes(root, writer);**

**writer.append("\tTotal: " + count+"\n");**

**}**

**public int countGlitchNodes() throws IOException {**

**System.out.println("Counting Glitches......");**

**return countGlitchNodes(root, null);**

**}**

**/\*\***

**\* Count in a preorder fashion**

**\* @param r**

**\* @param writer**

**\* @return**

**\* @throws IOException**

**\*/**

**private int countGlitchNodes(BTNode r, BufferedWriter writer) throws IOException {**

**if (r == null) {**

**return 0;**

**} else {**

**int count = 0;**

**//System.out.println("\t\t\t\tNode from " + r.getStartValues() + " to " + r.getEndValues());**

**if (r.getGlitch() != null) {**

**String output = "\t\tGlitch in " + r.getStartValues() + " to " + r.getEndValues() + " from " + r.getGlitch().getStart() + " to " + r.getGlitch().getEnd();**

**if (writer!= null) {**

**writer.write(output+"\n");**

**} else {**

**System.out.println(output);**

**}**

**count = 1;**

**}**

**count += countGlitchNodes(r.getLeft(), writer);**

**count += countGlitchNodes(r.getRight(), writer);**

**return count;**

**}**

**}**

**/\* Function to search for an element \*/**

**public BTNode search(int valStart, int valEnd) {**

**return search(root, valStart, valEnd);**

**}**

**/\*\* Function to search for an element recursively**

**/\* Assumes a Binary Search Tree**

**\*/**

**private BTNode search(BTNode r, int start, int end) {**

**BTNode node;**

**if (r==null) {**

**return null;**

**}**

**//else if(larger(toInsert, node)) { //Must go to the right**

**if (r.getStartValues() == start && r.getEndValues() == end) {**

**return r;**

**}**

**if (larger(new BTNode(start, end, null), r)) {**

**return search(r.getRight(), start, end);**

**} else {**

**return search(r.getLeft(), start, end);**

**}**

**}**

**/\* Function for inorder traversal \*/**

**public void inorder() {**

**inorder(root);**

**}**

**private void inorder(BTNode r) {**

**if (r != null) {**

**inorder(r.getLeft());**

**System.out.println(r.getStartValues() + "-" + r.getEndValues());**

**inorder(r.getRight());**

**}**

**}**

**/\* Function for preorder traversal \*/**

**public void preorder() {**

**preorder(root);**

**}**

**private void preorder(BTNode r) {**

**if (r != null) {**

**System.out.println(r.getStartValues() + "-" + r.getEndValues());**

**preorder(r.getLeft());**

**preorder(r.getRight());**

**}**

**}**

**/\* Function for postorder traversal \*/**

**public void postorder() {**

**postorder(root);**

**}**

**private void postorder(BTNode r) {**

**if (r != null) {**

**postorder(r.getLeft());**

**postorder(r.getRight());**

**System.out.println(r.getStartValues() + "-" + r.getEndValues());**

**}**

**}**

**}**

1. **Binary Tree Node (BTNode) Class-**

**The Binary Tree Node is a stateStart and stateEnd function and is also a glitch instance**

**package app.tree;**

**//import java.util.Scanner;**

**/\* Class BTNode \*/**

**public class BTNode {**

**public BTNode left, right;**

**private int startValues, endValues;**

**private Glitch glitch;**

**/\* Constructor \*/**

**public BTNode() {**

**left = null;**

**right = null;**

**startValues = 0;**

**endValues = 0;**

**glitch = null;**

**}**

**/\* Constructor \*/**

**public BTNode(int startV, int endV, Glitch g) {**

**left = null;**

**right = null;**

**startValues = startV;**

**endValues = endV;**

**glitch = g;**

**}**

**public BTNode(int startV, int endV, int startTime, int endTime) {**

**left = null;**

**right = null;**

**startValues = startV;**

**endValues = endV;**

**glitch = new Glitch(startTime, endTime);**

**}**

**/\* Function to set left node \*/**

**public void setLeft(BTNode n) {**

**left = n;**

**}**

**/\* Function to set right node \*/**

**public void setRight(BTNode n) {**

**right = n;**

**}**

**/\* Function to get left node \*/**

**public BTNode getLeft() {**

**return left;**

**}**

**/\* Function to get right node \*/**

**public BTNode getRight() {**

**return right;**

**}**

**/\* Function to set data to node \*/**

**public void setData(int startV, int endV, Glitch g) {**

**startValues = startV;**

**endValues = endV;**

**glitch = g;**

**}**

**/\* Function to get data from node \*/**

**public int getStartValues() {**

**return startValues;**

**}**

**public int getEndValues() {**

**return endValues;**

**}**

**public Glitch getGlitch() {**

**return glitch;**

**}**

**}**

1. **Glitch Class**

**The Glitch class contains the glitch startTime and glitch endTime functions**

**package app.tree;**

**/\*\***

**\* Glitch**

**\*/**

**public class Glitch {**

**private int startTime, endTime;**

**public Glitch(int t0, int t1) {**

**startTime = t0;**

**endTime = t1;**

**}**

**public int getStart() {**

**return startTime;**

**}**

**public int getEnd() {**

**return endTime;**

**}**

**}**

1. **Circuit Class**

**The Circuit class is an array of circuit nodes and is a variable of of <char , data>. The Circuit class is able to:**

1. **Convert from string to circuit**
2. **Change data values**
3. **Get circuit output**
4. **Get circuit end time**

**package app;**

**import java.io.BufferedWriter;**

**import java.io.File;**

**import java.io.FileNotFoundException;**

**import java.io.FileWriter;**

**import java.io.IOException;**

**import java.util.Hashtable;**

**import java.util.Scanner;**

**import java.util.Stack;**

**import app.circuitNode.Data;**

**import app.circuitNode.Gate;**

**import app.circuitNode.CircuitNode;**

**/\*\***

**\***

**\* Includes a Hashtable to easily manipulate Data inputs**

**\***

**\* //////Project Requirments//////// Represents a Circuit in terms of a Node**

**\* (Gate and Data) Array. Includes a BubbleSort to sort the nodes Includes a**

**\* Stack used when translating from a String equation to its representation**

**\*/**

**public class Circuit {**

**public final char START = '(', END = ')', ADD = '+', MULT = '\*', NOT = '~', MULTREPLACEMENT = (char)128;**

**private Gate[] nodes; // All the Important points inside the circuit**

**private Hashtable<Character, Data> inputs = new Hashtable<Character, Data>(); // Maps Data to user's characters**

**private int firstInput = Integer.MAX\_VALUE; // Remembers ASCII of a-Most character**

**private Gate outputGate;**

**private String circuitName;**

**// Take a file to produce circuit**

**public Circuit(File path) {**

**String equation = "";**

**Scanner input;**

**try {**

**input = new Scanner(path);**

**equation = input.nextLine();**

**equationToCircuit(equation);**

**this.circuitName = equation.replace(MULT, MULTREPLACEMENT);**

**writeCircuit();**

**} catch (FileNotFoundException e) {**

**e.printStackTrace();**

**System.out.println("BAD");**

**}**

**}**

**public Circuit(String equation) {**

**equationToCircuit(equation);**

**this.circuitName = equation.replace(MULT, MULTREPLACEMENT);**

**writeCircuit();**

**}**

**//For the file that the user can read...**

**private void writeCircuit() {**

**BufferedWriter writer = null;**

**try {**

**writer = new BufferedWriter(new FileWriter(getOutputFile()));**

**writeData(writer);**

**writeGates(writer);**

**writer.close();**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

**}**

**//Explains Data input info**

**private void writeData(BufferedWriter writer) throws IOException {**

**writer.write("Data inputs:\n");**

**for (Character key : inputs.keySet()) {**

**writer.append(key + " - " + inputs.get(key)+"\n");**

**}**

**}**

**//Explains all the gates of the circuit**

**private void writeGates(BufferedWriter writer) throws IOException {**

**writer.append("\nCircuit Gates:\n");**

**for (Gate s : nodes) {**

**writer.append("Memeory Loc: " + s);**

**if (s.getClass() == Gate.class) {**

**writer.append("\t"+s.getGateString(this)+"\n");**

**(s).getDelays().printList(writer);**

**}**

**}**

**writer.append("\n");**

**}**

**/\*\***

**\* Translates an integer into a boolean array, then calls setInputs(boolean[] b)**

**\* @param inputValues A base 2 integer, where 1 represents true and 0 represents false**

**\*/**

**public void setInputs(int inputValues) {**

**String strEnteredInputs = inputValues+"";**

**int enteredInputs = strEnteredInputs.length();**

**int numOfInputs = inputs.size();**

**int amountToAdd = numOfInputs-enteredInputs;**

**if(amountToAdd < 0) amountToAdd = 0;**

**boolean[] boolInputs = new boolean[inputs.size()];**

**//System.out.println("Entered: " + strEnteredInputs + ", length: " + enteredInputs + ", numOfInputs: " + numOfInputs + ", amountToAdd: " + amountToAdd);**

**for (int i = 0; i < numOfInputs; i++) {**

**if (i < amountToAdd) {**

**boolInputs[i] = false;**

**} else {**

**boolInputs[i] = (strEnteredInputs.charAt(i-amountToAdd)=='1');**

**}**

**}**

**setInputs(boolInputs);**

**}**

**//Assumes correct size of array, with inputs[0] as the a-Most character**

**public void setInputs(boolean[] inputValues) {**

**int currentInputAscii = firstInput;**

**for (boolean b : inputValues) {**

**while (this.inputs.get((char)currentInputAscii) == null) {**

**currentInputAscii++;**

**}**

**//System.out.println("Bool: " + b + " for " + (char)currentInputAscii);**

**this.inputs.get((char)currentInputAscii).setValue(b);**

**currentInputAscii++;**

**}**

**}**

**/\*\***

**\* Uses a stack to create the Node array and the Mapping of Characters to Data inputs**

**\* @param equation**

**\*/**

**public void equationToCircuit(String equation) {**

**int equationNum = 0;**

**Stack<CircuitNode> terms = new Stack<CircuitNode>(); //Remember the order of how things are seen in the equation**

**this.nodes = new Gate[countOperators(equation)]; //Each of the terms (will be sorted with timings)**

**int equationLength = equation.length();**

**//Look at each part of the equation**

**for (int i = 0; i < equationLength; i++) {**

**//System.out.println("Equation: " + equation);**

**//System.out.println("\tLocation: " + i + " < " + equationLength);**

**char index = equation.charAt(i);**

**switch(index) {**

**//A closing parenthasis, create a new Element (using operator and data point(s))**

**case START:**

**break;**

**case END: //Create a new (Node) Element, and update the position in the array**

**//System.out.println("\tStacking Node");**

**//Create new term**

**addElement(terms, equationNum);**

**equationNum++;**

**break;**

**case ADD: //Operator (AND/MULT/NOT), Add a full gate to the stack, but only the operator field**

**//System.out.println("\tStacking Operator +");**

**terms.add(new Gate(null, Gate.TYPE.OR, null));**

**break;**

**case MULT:**

**//System.out.println("\tStacking Operator \*");**

**terms.add(new Gate(null, Gate.TYPE.AND, null));**

**break;**

**case NOT:**

**//System.out.println("\tStacking Operator ~");**

**terms.add(new Gate(null, Gate.TYPE.NOT, null));**

**break;**

**default: //Regular input, add a Data to the stack.**

**//System.out.println("\tStacking Data " + index);**

**//Remembers the smallest input, and pairs each Data Input with the user's character key**

**if (inputs.containsKey(index)) {**

**//System.out.println("\t\tIt already exists");**

**terms.add(inputs.get(index));**

**} else { //Only add a mapping for new characters**

**//System.out.println("\t\tNew Data point");**

**if ((int)index < firstInput) {**

**firstInput = (int)index;**

**}**

**CircuitNode temp = new Data(false);**

**inputs.put(index, (Data)temp);**

**terms.add(temp);**

**}**

**}**

**}**

**outputGate = this.nodes[this.nodes.length-1];**

**}**

**/\*\***

**\* Used to figure out how big the array of Nodes has to be.**

**\* @param equation The user inputted String with**

**\* @return The number of \*, +, and ~ in the equation.**

**\*/**

**private int countOperators(String equation) {**

**int count = 0;**

**for (int i = 0; i < equation.length(); i++) {**

**char index = equation.charAt(i);**

**if (index == ADD || index == MULT || index == NOT) {**

**count++;**

**}**

**}**

**return count;**

**}**

**/\*\***

**\* Uses the top Circuit Nodes from the stack to create a new Gate.**

**\* This new Gate is added to the stack and the this.nodes array**

**\* @param terms The stack which is used for ordering terms (Nodes)**

**\* @param equationNum the location in the array to place the new complete Gate**

**\*/**

**private void addElement(Stack<CircuitNode> terms, int equationNum) {**

**CircuitNode term;**

**Gate fullElement;**

**term = terms.pop(); //Top element on the stack can be a Data or Gate.**

**Gate operator = (Gate)terms.pop(); //Second element on the stack MUST be an Operator (partial Gate)**

**//A NOT only requires one Data point. AND/OR require two**

**if (operator.getOperator() == Gate.TYPE.NOT) {**

**fullElement = new Gate(null, operator.getOperator(), term);**

**} else {**

**fullElement = new Gate(terms.pop(), operator.getOperator(), term);**

**}**

**//The fullElement might become an input to future Gates. Either way, store it into the array.**

**terms.add(fullElement);**

**nodes[equationNum] = fullElement;**

**}**

**/\*\***

**\* @return The Character that the user inputted to represent this Node**

**\* @param value The internally created/stored Circuit Node**

**\*/**

**public Character findKey(CircuitNode value) {**

**for (Character key : inputs.keySet()) {**

**if (inputs.get(key) == value) {**

**return key;**

**}**

**}**

**return '\_';**

**}**

**public Hashtable<Character, Data> getInputs() {**

**return inputs;**

**}**

**public CircuitNode[] getNodes() {**

**return nodes;**

**}**

**public int getNumOfInputs() {**

**return inputs.size();**

**}**

**public String getCircuitName() {**

**return circuitName;**

**}**

**public String getOutputFile() {**

**return circuitName+".txt";**

**}**

**public Gate getOutputGate() {**

**return outputGate;**

**}**

**public boolean getCircuitOutput() {**

**return outputGate.getValue();**

**}**

**public int circuitEndTime() {**

**return outputGate.getDelays().getLastNode().getData()+outputGate.getGateDelay();**

**}**

**/\*\***

**\* Must be done before EACH state change check.**

**\* This is because the queues get used up throughout a check.**

**\*/**

**public void initializeGatesForCheck() {**

**for (Gate g : nodes) {**

**g.initializeInputQueue();**

**}**

**}**

**/\*\***

**\* Bubble sort based on the largest of the InputQueue and OutputQueue's size.**

**\*/**

**public void sortByQueues() {**

**int numOfInputs = nodes.length;**

**//Loop through n-1 times**

**for (short num = 0; num < numOfInputs-1; num++) {**

**//Loop until the rest of the array is sorted**

**for (short i = 0; i< numOfInputs-num-1; i++) {**

**Gate currentNode = nodes[i];**

**Gate nextNode = nodes[i+1];**

**if (currentNode.largestQueueSize() > nextNode.largestQueueSize()) {**

**nodes[i] = nodes[i+1];**

**nodes[i+1] = currentNode;**

**}**

**}**

**}**

**}**

**}**

1. **Circuit Node Class**

**The Circuit node class is an abstract class that uses a boolean value, the integer gateDelay and MyLinkedList inputDelays. This class is also implemented by the Data and Gate classes.**

**package app.circuitNode;**

**import app.linkedList.MyLinkedList;**

**/\*\***

**\* Node**

**\*/**

**public interface CircuitNode {**

**public MyLinkedList getDelays();**

**public boolean getValue();**

**public int getGateDelay();**

**}**

1. **Gate**

**The Gate class defines a gate using a left and right input (circuitNode, which can be data or a gate) and an operator. Using that and its connection to other gates, It calculates the delay time for each gate, which can be used to find output values for the gates, and therefore possible glitches.**

**package app.circuitNode;**

**import java.util.LinkedList;**

**import java.util.Queue;**

**import app.Circuit;**

**import app.linkedList.LinkedListNode;**

**import app.linkedList.MyLinkedList;**

**/\*\***

**\* Gate: represents some type of operation and at least one input**

**\***

**\* //////////Project Requirements/////////**

**\* Uses a Queue for setting the internal nodes in the proper order**

**\* Uses a LinkedList for organizing all the delay times for this Gate**

**\*/**

**public class Gate implements CircuitNode {**

**// Gates have a left and right input (NOT only uses the rightInput)**

**private final CircuitNode leftInput, rightInput;**

**// The type of gate**

**private final TYPE operator;**

**// All the times this gate TAKES PREVIOUS inputs.**

**private MyLinkedList setOfDelays;**

**//Queues for when to update internalValue**

**private Queue<DelayUpdate> inputDelays = new LinkedList<DelayUpdate>();**

**private Queue<DelayUpdate> outputDelays = new LinkedList<DelayUpdate>();**

**// Internal node for gate**

**private boolean internalValue;**

**public Gate(final CircuitNode leftInput, final TYPE operator, final CircuitNode rightInput) {**

**this.leftInput = leftInput;**

**this.rightInput = rightInput;**

**this.operator = operator;**

**calcGateDelays();**

**if (this.rightInput != null) {**

**internalValue = calcValue();**

**}**

**}**

**// Gate types and their delays**

**public enum TYPE {**

**AND, OR, NOT**

**};**

**public int getGateDelay() {**

**if (operator == TYPE.AND)**

**return 25;**

**if (operator == TYPE.OR)**

**return 20;**

**if (operator == TYPE.NOT)**

**return 10;**

**// else**

**return 0;**

**}**

**/\*\***

**\* Calculate and return internal node value**

**\* Used when the internal node has to be updated**

**\*/**

**public boolean calcValue() {**

**boolean newInternal;**

**if (this.operator == TYPE.AND) {**

**newInternal = leftInput.getValue() && rightInput.getValue();**

**} else if (this.operator == TYPE.OR) {**

**newInternal = leftInput.getValue() || rightInput.getValue();**

**} else if (this.operator == TYPE.NOT) {**

**newInternal = !rightInput.getValue();**

**} else {**

**System.out.println("Error in Gate.java with operator: " + this.operator);**

**System.exit(-1);**

**newInternal = internalValue;**

**}**

**return newInternal;**

**}**

**/\*\***

**\* Form list of gate delay times**

**\* happens when creating the gate**

**\*/**

**private void calcGateDelays() {**

**// Possible operator only gates**

**if (this.leftInput != null || this.rightInput != null) {**

**if (this.leftInput == null) {**

**this.setOfDelays = this.rightInput.getDelays();**

**this.setOfDelays.addToEach(this.rightInput.getGateDelay());**

**} else {**

**this.setOfDelays = MyLinkedList.sortWithOtherAdded(this.leftInput.getDelays(), this.leftInput.getGateDelay(), this.rightInput.getDelays(), this.rightInput.getGateDelay());**

**}**

**this.setOfDelays.RemoveDuplicates();**

**//System.out.println("Delays:");**

**//this.setOfDelays.printList();**

**}**

**}**

**////////////InternalValue updating with the Queues/////////////////**

**//Important for waiting before updating the internalNode of the gate.**

**private class DelayUpdate {**

**int inputTime, outputTime;**

**boolean internalNode = false;**

**public DelayUpdate(int inputTime, int gateDelay) {**

**this.inputTime = inputTime;**

**this.outputTime = this.inputTime+gateDelay;**

**internalNode = false;**

**}**

**//Getters and Setters**

**public void setInternalNode(boolean inputValue) { internalNode = inputValue; }**

**public boolean getInternalNode() { return internalNode; }**

**public int getInputTime() { return inputTime; }**

**public int getOutputTime() { return outputTime; }**

**}**

**public void initializeInputQueue() {**

**LinkedListNode trav = this.setOfDelays.getRoot();**

**while(trav != null) {**

**inputDelays.add(new DelayUpdate(trav.getData(), getGateDelay()));**

**trav = trav.getNext();**

**}**

**}**

**public void takeInputsAndTransfer() {**

**inputDelays.peek().setInternalNode(calcValue());**

**outputDelays.add(inputDelays.remove());**

**}**

**public void updateInternalAndRemove() {**

**//System.out.print("Node: " + this);**

**this.internalValue = outputDelays.remove().getInternalNode();**

**//System.out.print("Updates to " + this.internalValue);**

**}**

**public int largestQueueSize() {**

**return Math.max(inputDelays.size(), outputDelays.size());**

**}**

**public int getNextInputTime() {**

**if (inputDelays.size() > 0) {**

**return inputDelays.peek().getInputTime();**

**} else {**

**return -1;**

**}**

**}**

**public int getNextOutputTime() {**

**if (outputDelays.size() > 0) {**

**return outputDelays.peek().getOutputTime();**

**} else {**

**return -1;**

**}**

**}**

**public String queueLengths() {**

**return "input Length: " + inputDelays.size() + ", Output Length: " + outputDelays.size();**

**}**

**/////////////////Getters for variables of the Gate///////////////////////**

**public Gate.TYPE getOperator() {**

**return operator;**

**}**

**public CircuitNode getLeftInput() {**

**return this.leftInput;**

**}**

**public CircuitNode getRightInput() {**

**return this.rightInput;**

**}**

**@Override**

**public boolean getValue() {**

**return internalValue;**

**}**

**@Override**

**public MyLinkedList getDelays() {**

**return setOfDelays;**

**}**

**///////////////////// ONLY USED FOR OUTPUT FOR USER////////////////////////////**

**/\*\***

**\* Recursive, Follows the the circuit down to its smallest parts**

**\*/**

**public String getGateString(final Circuit c) {**

**return getLeftString(c) + getStringOperator(c) + getRightString(c);**

**}**

**public String getLeftString(final Circuit c) {**

**if (this.leftInput == null) {**

**return "";**

**} else if (this.leftInput.getClass() == Gate.class) {**

**return "("+((Gate) this.leftInput).getGateString(c)+")";**

**} else {**

**return ((Data) this.leftInput).getString(c) + "";**

**}**

**}**

**public String getRightString(final Circuit c) {**

**if (this.rightInput == null) {**

**return "";**

**} else if (this.rightInput.getClass() == Gate.class) {**

**return "("+((Gate) this.rightInput).getGateString(c)+")";**

**} else {**

**return ((Data) this.rightInput).getString(c) + "";**

**}**

**}**

**public Character getStringOperator(final Circuit c) {**

**switch (this.operator) {**

**case AND:**

**return c.MULT;**

**case OR:**

**return c.ADD;**

**case NOT:**

**return c.NOT;**

**default:**

**System.out.println("Error! Wrong Operator in this GATE!");**

**return '#';**

**}**

**}**

**}**

1. **Delay (DelayUpdate) Class**

**The Delay Update class consists of the inputTime (int), outputTime (int) and internalValue (bool). This class is used inside the gate class.**

1. **Data Class**

**package app.circuitNode;**

**import app.Circuit;**

**import app.linkedList.MyLinkedList;**

**import app.linkedList.LinkedListNode;**

**/\*\***

**\* Data**

**\*/**

**public class Data implements CircuitNode {**

**private boolean value;**

**public Data(boolean value) {**

**this.value = value;**

**}**

**@Override**

**public boolean getValue() {**

**return this.value;**

**}**

**@Override**

**/\*\***

**\* The delay directly from an input is 0**

**\*/**

**public MyLinkedList getDelays() {**

**return new MyLinkedList(new LinkedListNode(0));**

**}**

**@Override**

**//Data inputs change instantaneously**

**public int getGateDelay() {**

**return 0;**

**}**

**public void setValue(boolean newBool) {**

**this.value = newBool;**

**}**

**public Character getString(Circuit c) {**

**return c.findKey(this);**

**}**

**}**

1. **My Linked List Class**

**The LinkedList class contains the following:**

1. **getRoot function**
2. **getLast function**
3. **sortWithOther function**
4. **removeDupicates function**
5. **addToEach function**
6. **addToBack function**

**package app.linkedList;**

**import java.io.BufferedWriter;**

**import java.io.IOException;**

**/\*\***

**\* LinkedList**

**\*/**

**public class MyLinkedList {**

**private LinkedListNode root;**

**public MyLinkedList() {**

**this(null);**

**}**

**public MyLinkedList(LinkedListNode root) {**

**this.root = root;**

**}**

**public LinkedListNode getRoot() {**

**return root;**

**}**

**public LinkedListNode getLastNode() {**

**LinkedListNode trav = root;**

**if (trav == null) {**

**return null;**

**}**

**while (trav.getNext() != null) {**

**trav = trav.getNext();**

**}**

**return trav;**

**}**

**/\*\***

**\* Used to have a gate's delay add on to each of it's input's delays.**

**\*/**

**public void addToEach(int value) {**

**LinkedListNode trav = root;**

**if (trav != null) {**

**while (trav != null) {**

**trav.setData(trav.getData() + value);**

**trav = trav.getNext();**

**}**

**}**

**}**

**/\*\***

**\* Used in accociation with the calculation of gate delays**

**\*/**

**public void RemoveDuplicates() {**

**LinkedListNode trav = root;**

**if (trav != null) {**

**while (trav.getNext() != null) {**

**if (trav.getData() == trav.getNext().getData()) {**

**trav.setNext(trav.getNext().getNext());**

**continue;**

**}**

**trav = trav.getNext();**

**}**

**}**

**}**

**public void printList(BufferedWriter writer) throws IOException {**

**LinkedListNode trav = root;**

**if(trav != null) {**

**writer.append("\tInput Delays: ");**

**while (trav!= null) {**

**writer.append(trav.getData() + " ");**

**trav = trav.getNext();**

**}**

**} else {**

**writer.append("\tEmpty List...");**

**}**

**writer.append("\n");**

**}**

**/\*\***

**\* Uses a version of merge sort to completely sort two linked lists together**

**\* @param first doesn't matter which linked List**

**\* @param other The other linekd list**

**\* @return A fully sorted array.**

**\*/**

**public static MyLinkedList sortWithOtherAdded(MyLinkedList first, int firstDelay, MyLinkedList other, int otherDelay) {**

**MyLinkedList newList = new MyLinkedList();**

**LinkedListNode t1 = first.root;**

**LinkedListNode t2 = other.root;**

**//Pick the smaller of each list until one is empty**

**while (t1 != null && t2 != null) {**

**if (t1.getData()+firstDelay > t2.getData()+otherDelay) {**

**newList.addToBack(new LinkedListNode(t2.getData()+otherDelay));**

**t2 = t2.getNext();**

**} else {**

**newList.addToBack(new LinkedListNode(t1.getData()+firstDelay));**

**t1 = t1.getNext();**

**}**

**}**

**//Then add all the elements on (can't just tack on one because changes to one linkedlist will affect another)**

**if (t2 != null) {**

**t1 = t2;**

**firstDelay = otherDelay;**

**}**

**while (t1 != null) {**

**newList.addToBack(new LinkedListNode(t1.getData()+firstDelay));**

**t1 = t1.getNext();**

**}**

**return newList;**

**}**

**private void addToBack(LinkedListNode newNode) {**

**if (this.root == null) {**

**this.root = newNode;**

**} else {**

**getLastNode().setNext(newNode);**

**}**

**}**

**}**

1. **Linked List Node Class**

**The LinkedListNode class is an class that contains the interger data and the next node location**

**package app.linkedList;**

**/\*\***

**\* LinkedListNode**

**\*/**

**public class LinkedListNode {**

**private int data;**

**private LinkedListNode next;**

**//Main constructor**

**public LinkedListNode(int data, LinkedListNode next) {**

**this.data = data;**

**this.next = next;**

**}**

**//Side constructors**

**public LinkedListNode() {**

**this(-1, null);**

**}**

**public LinkedListNode(int data) {**

**this(data, null);**

**}**

**//Getters/setters**

**public void setNext(LinkedListNode next) {**

**this.next = next;**

**}**

**public void setData(int data) {**

**this.data = data;**

**}**

**public int getData() {**

**return this.data;**

**}**

**public LinkedListNode getNext() {**

**return this.next;**

**}**

**}**

****

**Project Summary:**

**Our code solved the issue of calculating and locating system glitches. We were able to do this by calculating delays within each gate output, using those to realistically simulate a state change, noting any changes in output.**

**Requirements Document:**

**Our program had to do the following:**

* **Represent a Circuit**
* **Be able to edit the input values of the circuit**
* **Figure out delay times for each circuit gate**
* **Use the delay times to calculate gate values over time**
* **Test all possible state changes for any circuit**
* **Store all state changes in an accessible way**
* **Be able to quickly find and display state changes, especially particular kinds**

**1: This code shall contain the following:**

1. **An Array**
2. **A Linked List**
3. **A Stack**
4. **A Queue**
5. **A Binary Tree**
6. **A Bubble Sort**

**2: This code shall be able to do the following tasks:**

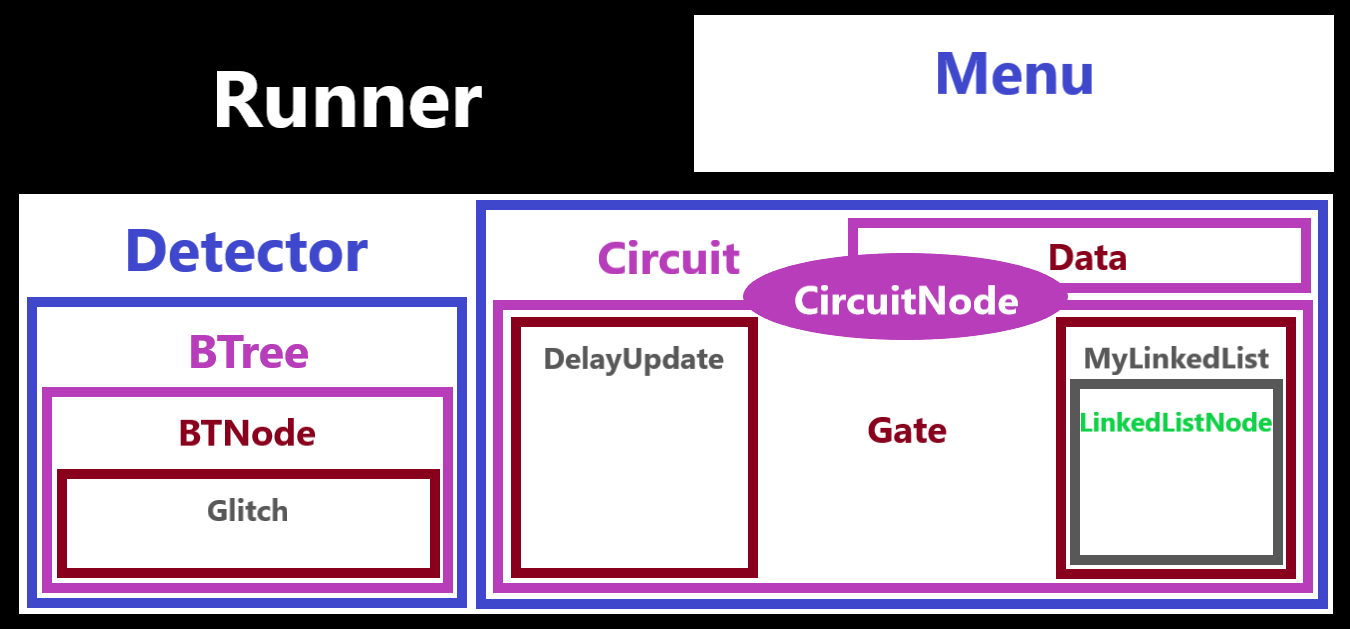
1. **Input user circuit**
2. **Calculate and display glitches in circuit**
3. **Display all time values and glitches**
4. **Show the locations of all glitches present**
5. **Display propagation delay times**

**3: This code shall be presented in a clear, user-friendly manner.**

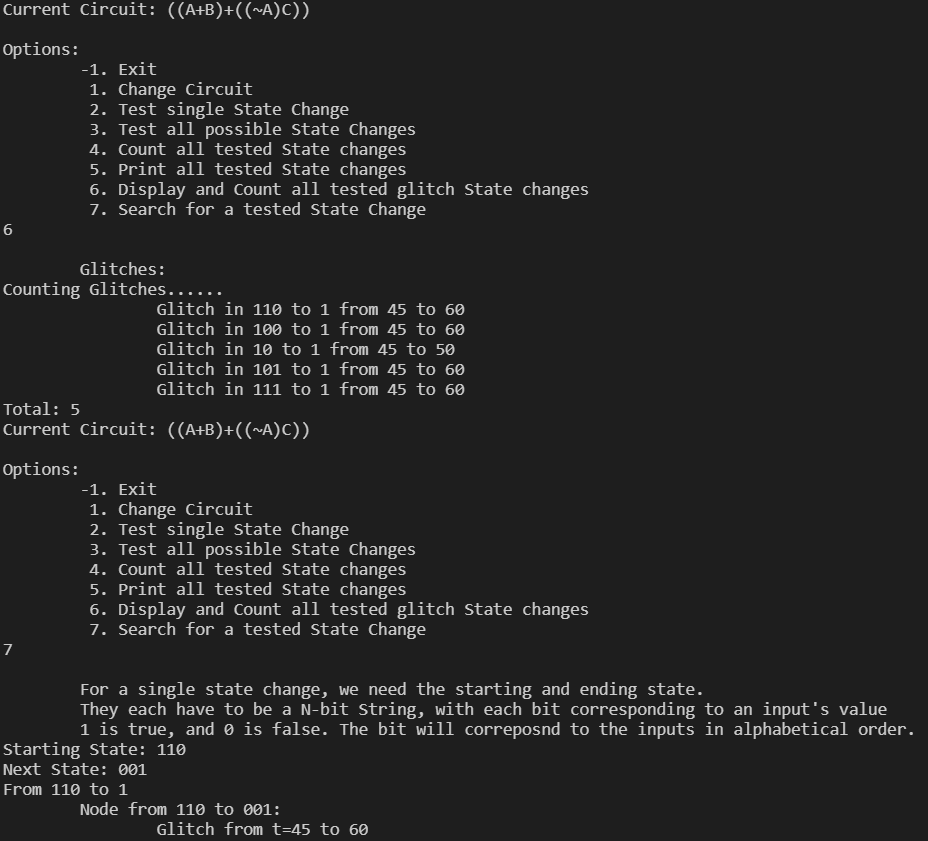
**4: This code shall solve an organizational problem.**

**How we went about structuring the program:**

1. The start of our class would come from the **Runner** class, which would mostly hold the main method. Throughout development. it was extensively used for testing specific classes.
2. To actually solve the problem, we would need a way to represent the circuit and a way to detect changes to a change in the circuit’s output. This is how we came about creating the **Circuit** and **Detector** class. The **Menu** class wasn’t added until the end of development, as the user interface was considered a low priority to a working program
3. A large portion of the design was how we would make the Circuit class work. We knew the circuit had to achieve the following
   1. Represent itself in terms of gates and inputs
   2. Create itself from a String input
   3. Be able to change the inputs
   4. Be able to take the circuit’s total output
   5. Be able to sort the gates so that detection would be easier
4. Therefore, we came to the conclusion that the **Circuit** itself would consist of **CircuitNodes**, which could either be a **Gate** or a **Data**(circuit input). The Each of the Gates would have to be easily accessible and sortable, so they would be put in an **array**. Using the user’s circuit equation it would be simple to find the array size, so there was no need for an arraylist or linkedlist. Each of the Datas would have to be represented by a letter, so they would be put in a hashtable, with the key being the Character that the user entered for the circuit. Converting from a string to the node array required the use of a **stack**. Here are some example circuits:
   1. (A+B)
      1. A and B would be represented by separate Data objects, while the entire A+B would be represented by one Gate.
      2. A put on stack, + put on Stack, B put on stack, A+B created
   2. (A\*B)+((~A)+C)
      1. A, B, and C would represent separate Data objects, while A\*B, ~A, ~A+C, and A\*B+(~A+B) would be 4 different gates.
      2. Stack on/off order… Stack contents
         1. A on, \* on, B on. {A, \*, B}
         2. Pull all three off and put on (A\*B) {**A\*B**}
         3. + on, ~ on, A on {A\*B, +, ~, A}
         4. Pull of two and put on ~A {A\*B, +, **~A**}
         5. + on, C on {A\*B, +, ~A, +, B}
         6. Pull off three and put on ((~A)+C) {A\*B, +, **~A+B**}
         7. Pull off three and put on (A\*B)+((~A)+C) {**(A\*B)+((~A)+C)**}
      3. Notice that a gate’s sides can be represented by other gates. For example we can say that
         1. Let p = (A\*B)
         2. Let m = (~A)
         3. Let r = m+C = ((~A)+C)
         4. Let o = p+r = (A\*B)+(m+C) = (A\*B)+((~A)+C)
      4. We can then say that p, m, r, and o are our four gates that we need our circuit to remember. This is important because each smaller gate being updated will result in the final gate being updated.
5. These Gates and Datas were linked, as a Gate’s inputs could either be another gate (as shown with the final gate of (A\*B)+((~A)+C), or an input, like any other gate in the same example. Therefore, both a Gate and Data would have to implement the abstract **CircuitNode** class. They would have to do the following
   1. Give the current output value of the gate
   2. Give the delay times for taking their input
      1. A gate has to recalculate their output each time their input’s change.
         1. A Gate with two Data inputs would do this once
         2. A Gate with Gate inputs could do this many more times
   3. Give the delay time for processing an input change (gateDelay)
      1. For Data, it would be instantaneous (0)
      2. For a Gate it would depend on the operation
         1. AND: 25
         2. OR: 20
         3. NOT: 10
      3. The differing delays are what would mess up a circuit and create a glitch. Therefore, we had to make each CircuitNode this intricate.
6. The **Data** class was easy to create. It had the ability to have its input changed and returned. It would always give a gate delay of 0, and it’s list of delays would only have one element, 0. Since any Data would be accessible in the hashtable of the circuit class, changing inputs is very easy.
7. The **Gate** class is much more complex. It has to do the following.
   1. Hold the gate’s inputs
   2. Know what operation is being performed (and its delay time)
   3. Hold the boolean output value of the gate
   4. Have a reusable list of delays to take the value of its input
8. To hold inputs, the **Gate** class held two CircuitNodes, which could therefore be a Gate or Data. Using an enum, an operator was held within the class to be either AND, OR, or NOT. The boolean output value of the gate would be based off of the inputs and operator. Since a delay was required, a system of a **MyLinkedList** and two **Queues** was implemented to hold when the input and output time of a certain boolean value. The LinkedList acted as the permanent times that the gate would take the boolean values of its inputs. A special LinkedList was created because sorting could be done in a mergesort type of fashion from the Data level up towards the output Gate of a circuit. In fact working from bottom up made this perfect, as working your way towards a circuit’s output could result in a power of 2 more delays than the previous layer, just like a merge sort. The queues acted as expendable resources during a state transition. Knowing if a gate should recalculate its value required knowing its next calculation time, so queues provided a “next” time way of storing the data directly from the LinkedList. The inputDelay Queue would hold the times that the gate would calculate an output, while the outputDelay Queue would hold the time that the gate would set its output to that calculated output. To us, this seemed like the best way to make out circuit function with the delay of real life circuits.
9. The **Detector** used the **Circuit** to find any glitches, this directly solving our problem statement. Every possible transition was represented by an **array** of **Queues**. Since N inputs to a circuit have 2^n states, there are 2^N-1 transitions to another state from a current state (the last transitions would be a state to itself, which can never result in a glitch). Therefore, we had an array, with each index representing the current state of the circuit (we would convert the base 10 to base 2 to get the state). For each one of those states, a queue with numbers 1 to (2^N-1) held what should be added to the state to get to another state. To make sure we didn’t go over or miss states with lower values than our current one, we would modulo this answer by 2^N.
10. Using each of these transitions, we would test a single state change (the current state would be based on the values of our data, and the next would use that value+the next value in the queue of that index). Each are stored in our **Binary Tree** class called **BTree**. Before beginning a test, the circuit’s Gate array would be sorted based on inputDelay times using a BubbleSort as to speed up the check. A single transition would go through every possible important moment in the circuit, updating the internal values at the inputTime and setting them at the outputTime for each gate. While doing so, any changes in the output of the circuit would be recorded. After all gates were fully updated, if the circuit’s output changed but ended as it started, there was a glitch. Any glitches are noted with a transition and stored in the tree.
11. The **Menu** class allows the user to easily navigate and use all these various classes. It provides explanations to usage, and is completely text based.

****

**Outputs for circuit with glitch… Whole thing (what you would normally do)**

****

**References: Writing to a file, converting to base 2**