Java Digital Breadboard Simulator:

A simulator for an educational electronics environment

by
Nicholas Glass

13150 words as calculated by the Microsoft Word word count function.
This includes all sections of the report.

Table of Contents

1 Abstract 2
2 Introduction 2
2.1 Simulation 2
2.1.1 Advantages of Computer Simulation 2
2.1.2 Disadvantages of Computer Simulation 3
2.2 Simulation Methods 3
2.2.1 Continuous Modelling 3
2.2.2 Discrete Modelling 3
2.2.2.1 Discrete Event-Driven Simulation 4
2.2.2.2 Discrete Cycle-Based Simulation 4
2.3 Electronics Simulation 4
2.4 Comparison of Existing Programs 6
2.4.1 The Applications 6
2.4.2 The Interface 6
2.5 Project Aims 8
3 Program Design 9
3.1 The Tools 9
3.2 Graphical User Interface 10
3.2.1 The Breadboard Metaphor 10
3.2.2 What's in a wire. Part 1. Drawing Connections 11
3.3 Simulation Engine 13
3.3.1 Nodes in the circuit 13
3.3.2 What's in a wire. Part 2. Simulating Connections 13
3.3.3 The Device Model 14
3.3.4 The Chip Model 15
3.3.5 The Discrete Simulation 15
4 Implementation 17
4.1 User Interface 17
4.2 The Simulation 19
4.3 Chip Models 20
4.4 Signal Tracing 21
4.5 Loading and Saving 23
5 Evaluation of Program 25
5.1 Simulation Validation 25
5.1.1 Connection Verification 25
5.1.2 Devices 25
5.1.3 Circuits 25
5.1.4 Timing 26
5.1.5 Other Behaviour 28
5.2 Interface Evaluation 29
6 Conclusions 30
6.1 Future Work 31
7 References 32
8 Appendices 34
8.1 User Guide 34
8.2 Example Chip Implementations 37
8.2.1 SN7400 Quadruple 2-Input Positive-Nand Gates 37
8.2.2 SN7470 And-Gated J-K Positive-Edge-Triggered Flip Flops with preset and clear 40
8.3 Test Circuits 44
8.3.1 2-Input Exclusive-Or 44
8.3.2 Not-S Not-R Flip-flop 44
8.3.3 Positive Edge-Triggered D-Type Flip-flop 45
8.3.4 (Clock NAND (Not Clock)) 45
8.3.5 Asynchronous Four-bit Binary Counter 46
8.3.6 Wired-Or 46

// 1 Abstract
The University of York Computer Science Department requires an electronics simulator package to complement its laboratory time. Current solutions are shown to be inadequate for the task, so a new simulator is created. An innovative graphical interface is developed and a cross-platform solution is implemented with the use of Java. Extensibility of the application is ensured through the use the use of a plug-in component architecture. The solution is thoroughly tested and found to be reliable for use in the design and testing of student circuitry.

2 Introduction

The Computer Science Department at the University of York offers courses in electronic systems design and implementation. Projects involve designing and creating digital microprocessor-controlled systems where interfacing of input and output devices is sometimes necessary.

Students working in departmental electronics laboratories have to be supervised by experienced students or staff, therefore there are time limitations on the available sessions. A program such as an electronic simulator would allow students to design and test systems outside of the laboratory, reducing time pressures.

The Computer Science laboratory computers run Linux whereas the majority of the University computers feature Microsoft Windows, hence any simulation software must run on both of these operating systems. This suggests that a simulator design ensuring ease of portability or choice of implementation language is necessary, such that the application will run in both environments.

2.1 Simulation

To get a better understanding of the software available or to design a new application, the concept of simulation must first be understood. The Collins English Dictionary [1] defines to simulate and simulator as:

> **simulate** *vb* **–lating**, **-lated** **1** to pretend to feel or perform (an emotion or action); imitate: *I tried to simulate anger.* **2** to imitate the conditions of (a situation), as in carrying out an experiment: *we can then simulate global warming*. **3** to have the appearance of: *the wood has been painted to simulate stone.* **simulated** *adj* **simulation** *n*
>
> **simulator** *n* a device that simulates specific conditions for the purposes of research or training: *a flight simulator.*

Hence, a computer simulation is a program that imitates or models an object or system. A system is defined as a collection of interacting elements that function together for some purpose [2].

2.1.1 Advantages of Computer Simulation

The main advantage of simulation is that it enables testing and design of a system before moving onto construction. By identifying faults in the design stage, it may be possible to avoid costly errors from appearing later in the product cycle. It may also be possible to determine the efficiency and accuracy of the design and allow the user

to experiment with several different approaches without ever having to build anything.

Simulation is also advantageous where design or operational failure has potentially life threatening risk. Examples include Nuclear reactor testing and pilot training neither of which can be performed safely on real systems.

2.1.2 Disadvantages of Computer Simulation

The disadvantages of computer simulation lie mostly in the fact that it is inherently computationally expensive to accurately model most real world systems as there are potentially infinite factors which could be considered. As such, it may take long periods of time to determine the result of an interaction, that in reality happens in fractions of a second. This is a particularly important issue if a simulator is supposed to operate in real time, such as in a flight-training program.

To reduce complexity, assumptions or simplifications are introduced to the simulation program. However, although this can speed up the simulator, any simplification made decreases the accuracy of the simulation making the results less reliable.

2.2 Simulation Methods

When designing or implementing a simulator, various techniques can be used to model a system. Traditionally, these are divided into continuous and discrete methods. The technique used depends on how the system changes with respect to time.

2.2.1 Continuous Modelling

Continuous simulation models dynamic systems with the use of differential equations.[3] To achieve this, the basic relationships between chosen state variables, need to be known in order to model the elements of the system. The dynamic behaviour of the system is then calculated by either solving the differential equations or iterating from some start condition to produce data for the system over time.

The relationships used for continuous modelling are typically convenient natural laws such as Newton's laws of motion of Kirchhoff's voltage laws. Thus, a model of this form will be widely accessible and easily understood by people from the associated fields.

Unfortunately, a continuous model is unsuitable for systems with more than a few thousand variables, as it is just too computationally expensive to calculate in any reasonable time. Thus to simulate models of higher complexity a coarser level of detail must be used.

2.2.2 Discrete Modelling

Discrete models can be used whenever a system's state changes instantaneously at discrete time intervals. In reality, this is rarely the case as changes are not instantaneous, however if this can be approximated, discrete simulation offers considerable speed advantages over continuous.

There are two main techniques used for discrete modelling, event-driven and cycle-based, (also known as event-by-event and epoch-by-epoch respectively) [4]. There are several variations that exist based on these ideas.

2.2.2.1 Discrete Event-Driven Simulation

An event is a change in state that happens at a particular time. In discrete event-driven simulation, the emphasis lies on updating the state of the system only upon the occurrence of events. This is a good strategy, as it means that time is not wasted evaluating sections of the system or periods of time where nothing is happening.

The main feature of an event-driven system is that of a global event queue. As events are generated by the system elements, they are added to this queue for later processing.

At each stage in the simulation, the events that occur next are removed from the front of the queue and processed. Any elements that are affected by these changes from the events are then re-evaluated to produce further events.

The event-driven method also has the effect that it automatically orders the evaluation of simulation elements due to the queuing system. In a system where this ordering is important such as in analysing the behaviour of electronic circuits, elements are only evaluated after the previous ones have generated events.

2.2.2.2 Discrete Cycle-Based Simulation

Unlike event-driven simulation, which updates at the system at the occurrence of events, cycle-based simulation updates the entire system at regular intervals. This has the advantage that the speed at which the simulation proceeds is directly related to time. Additionally, since no central queuing process is required, this type of discrete simulation lends itself well to parallel or distributed architectures, with each process evaluating a single simulation element. This has resulted in the development of hardware simulation accelerators which utilise this method.

Unfortunately, since each simulation element is only evaluated at each cycle, events that occur at other times may be ignored or lost. Hence, a cycle-based simulation is unsuitable for a system involving lots of asynchronous behaviour, such as some electronic circuits.

2.3 Electronics Simulation

Electronics simulation is the process of simulating an electronics circuit. Four basic levels of electronics simulation have been identified [5]: circuit level, gate level, register transfer level, and system level. A simulation that combines more than one level is called a mixed-mode simulation.

Circuit Level Simulation is the ideal form of electronics simulation as it aims to represent the analogue-waveform response of a circuit. However, such an approach is highly computationally expensive. Circuit-level simulations typically run as a continuous simulation.

Components at circuit level, that comprise a logic gate, may be simplified in a gate level model which represents them as a single element. This decreases complexity in several ways. Firstly, the simulator now models fewer components and secondly, logic gates only respond to discrete logic levels thus considerably simplifying the representation of voltage.

Register transfer, again groups components, this time assimilating gates into registers and thus incorporating the idea of state into the circuit elements. Register transfer models will also respond to the same voltage levels as gate-level models and as such, the two are often grouped as logic-level simulations. As the transitions between states in a logic-level simulation are nearly instantaneous, they can be assumed so, and are therefore suitable for discrete simulation.

Logic-level simulation can be further subdivided into switch, gate, functional and behavioural levels [6].

Switch level falls just beneath the level of abstraction offered by gate-level simulation and considers circuits as collections of transistors and wires. Transistors are then approximated as simple switches, with propagation delay in switching, and wires are modelled as zero-delay conductors.

Functional-level models map onto the architectures functional blocks, and as such will be able to accept and produce the same values as the actual hardware. It differs from gate models in the way that it can process data internally. For example a comparator comprised of four single bit comparators must examine each binary bit to determine the larger of its two inputs, a functional model can just read the inputs as integers and carry out a single comparison.

Behavioural level simulation only seeks to mimic the behaviour of a block of hardware and doesn't necessarily have to achieve this in the same way as the hardware. They give a general overview of the function provided by the hardware and allow the designer to experiment with higher-level solutions.

System level simulation, unlike gate and register transfer levels, does not compose its models by grouping components from a lower level. Instead, it is derived largely by abstraction of elements of the actual system, as such it is a form of behavioural simulation. In terms of system level simulation, Breuer [6] describes the system as 'any work-performing entity or set of connected entities in the computer system, up to and including the latter.'

In general, the systems built for the University of York's electronic courses comprise of mostly digital systems, with analogue circuitry used only for the inputs and outputs. Thus, a logic-level simulation would be most useful with perhaps some interfacing to a circuit-level system for IO. As the major concern is the testing of circuits themselves, a behavioural or system-level simulation would not be of particular use here as the level of abstraction is too high.

2.4 Comparison of Existing Programs

The capacity required from the simulator at the University, does not justify the licensing cost of commercial software. Instead, free software is examined to meet the requirements.

It is felt that it would be advantageous if the program offered an intuitive graphical interface instead of opting for textual circuit description, as the emphasis of the course is the circuit development, not the use of the associated simulator. As such, ESIM [8] and similar programs will be discounted.

There are already several freely available graphical logic-level simulators [9 – 24]. Current research [9] has compared these systems however, this has mainly been concerned with the interface function and further inspection is necessary.

2.4.1 The Applications

All of the identified freely available simulation applications offer a two or three state logic simulation, mostly using discrete event processing. Therefore, there is not much that separates the relative power of the individual programs.

The range of components offered by each package is also similar and include clocks, switches, light emitting diodes and flip-flops as well as the basic gates that are to be expected in such a program. Multimedia Logic [21] stands out, as it features some interesting higher-level components such as a keyboard and terminal, however it does not have the range of components necessary for general circuit design.

With the exception of Digital Simulator [15], which offers a plug-in architecture, none of the programs allow for easy addition of components via any method other than recompiling of the application. This is an important feature for an electronics simulator, due to the large number of available electronic devices, it should be easy to add new ones to a program when required, and should not require understanding of the entire program to do so.

Of the sixteen applications under scrutiny, about half of these are able to run on multiple platforms, mostly through the use of Java. Unfortunately, the only one that features addition of devices through a plug-in architecture runs solely on Microsoft Windows. Thus if an easily extensible platform independent simulator is required, it will need to be designed from scratch.

2.4.2 The Interface

As stated previously the investigation into simulation applications [9] had already looked at the interface. The research examined factors such as wire representation, variation with respect to numbers of gate inputs, operations such as saving and printing, and tried to gauge the ease of use offered by the various interfaces. However, the question was not raised as to whether the graphical metaphor, common to all the simulators was the most appropriate for the task.

All the investigated non-commercial programs represent the circuits using either standard logic symbols (Figure 1 (a)) or ANSI/IEEE Std. 91-1984 symbols [26] (Figure 1 (b)). This is useful in the design stages of circuit modelling, as these symbols are recognised internationally so make for good documentation of the circuit.

Hence, this type of interface is well suited to the simulation of circuitry for design testing. However, this system is less applicable to a practical course as it does not show the actual form of the circuitry used in the laboratory.

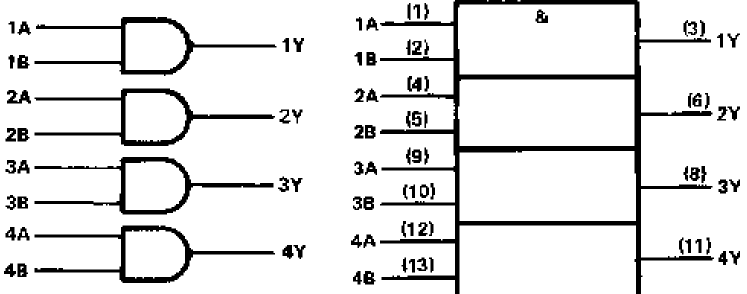


Figure 1: From left to right: (a) Logic diagram and (b) ANSI/IEEE Std. 91-1984 symbol for a SN7408 And-Gate Package

Virtual Vulcan [25], a commercial circuit trainer simulated in software, experiments with a different graphical metaphor. This piece of software uses computer representation of a real world objects and enables the user to utilise the software just like the real world trainer (Figure 2). However although it does offer a reasonable selection of chips to experiment with, the program is limited to just a single breadboard which is insufficient for the circuits created in the Computer Science courses.

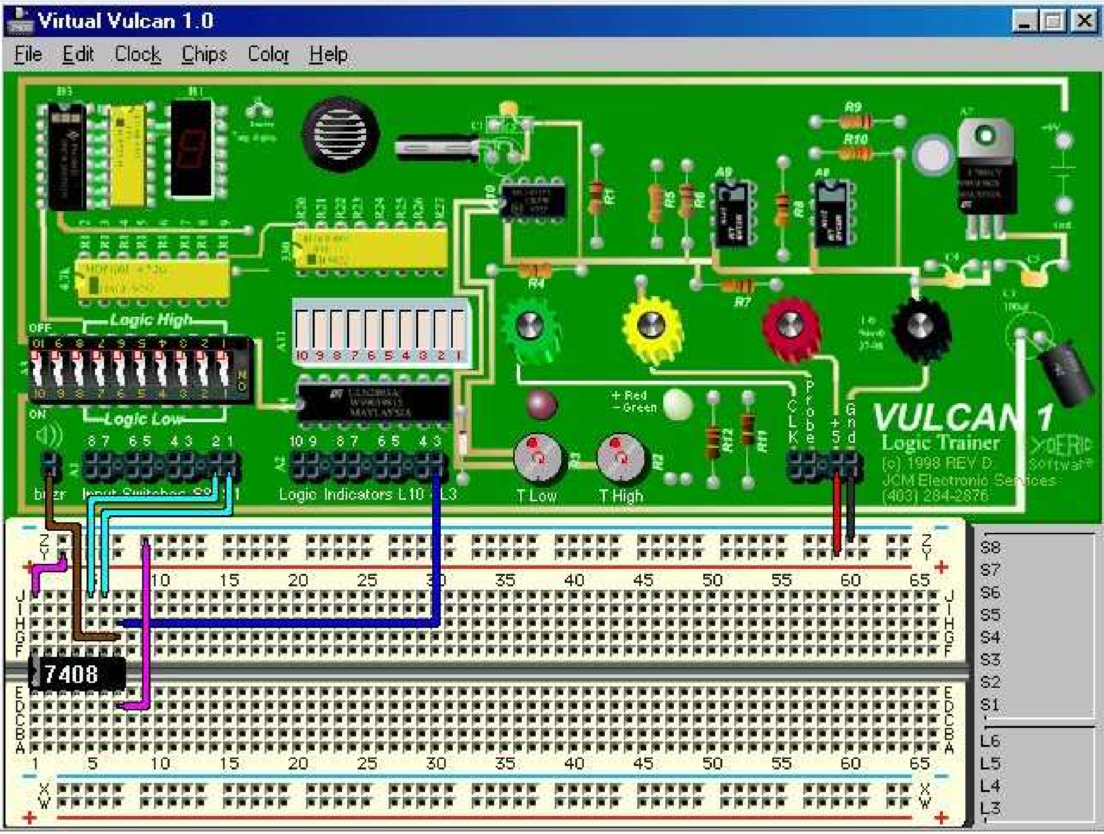


Figure 2: Virtual Vulcan Interface

The circuits designed and built for the University courses, although potentially complex, generally consist of a microprocessor and associated off the shelf

component packages. Using either the standard logic symbols or the ANSI/IEEE representations, these devices tend to be represented by rectangular blocks. Thus, an interface featuring representations of real world objects would not lose too much comprehensibility and instead, would produce a standard work environment between the laboratory and the software which reduces the amount of time it takes to switch between the hardware and the simulator.

2.5 Project Aims

The aim of the project is to design an electronics simulator for use in testing circuitry designs for students on the Chips to Systems course at the University of York. To achieve this a discrete, logic-level simulation will be used to model the digital portion of the student circuitry.

The program should run on at least Linux and Microsoft Windows operating systems. It should provide components typical of those used in the lab sessions such as the Texas Instruments SN74 range, light emitting diodes and switches and should feature an extensible system for the addition of further components as necessary. Finally, it should aim to provide a graphical interface mimicking the environment of the electronics laboratory and will be fully tested and evaluated for its applicability to this environment.

3 Program Design

3.1 The Tools

The most important aspect to consider, when choosing the tools to create the program, is how to maintain the cross-platform nature specified by the requirements. There are two main ways of achieving this; choosing tools and designing the program such that the application is easily ported between environments, or selecting an interpretive language such that the same code can be run on each environment.

There are advantages of each method, namely that native code is executed faster than interpretive code, whereas with interpretive code you will only have to maintain the one code base. However, usually it easy enough to design program code such that it is easily ported, thus the execution speed argument will often favour this method. This argument is not so strong for this application.

One of the primary goals is that the application is graphical in nature. Although there are many libraries for providing graphical output to non-interpretive languages such as C, it is much harder to find ones that provide good cross-platform support. Often these require so many levels of indirection to provide a common interface, that the resulting speed would be similar to that of an interpreted language.

The second problem with choosing the porting route is the wish to use some kind of plug-in architecture to provide extensibility to the types of components that the program supports. If the main application is ported then it must be ensured that all the plug-ins are also transferred. Thus, the solution must lie with an interpretive language.

Several interpretive languages exist that also offer graphical toolkits. TCL/TK and Java are probably the two most popular, and as such will be the best candidates for the work as it will be easier to find support, in terms of both documentation and available interpreters. TCL/TK and Java in particular provide another benefit over certain interpretive languages, as it is possible for applications created in both of these languages to run through an appropriately configured web browser. This facilitates program deployment, as only the one copy of the software needs to be kept, being accessible to any number of computers through the Internet.

Java has the advantage over TCL/TK, as it is an object-orientated language, which transfers well to circuit simulation, as it is natural to think of circuit components as objects. It is for this reason that Java is selected in preference of TCL/TK, for although there exists object-orientated extensions to TCL/TK, these are less popular and less well supported than standard TCL/TK and Java.

Java offers two graphical application programming interfaces, AWT and Swing, which offer rich ranges of ready-made graphical components. Swing was chosen for the graphical implementation, for although AWT currently has better support through web browsers, Swing is rapidly growing in popularity and features the more advanced feature set [27].

With the tools chosen, the program can now be developed. A very top-down approach is taken towards this piece of software, starting with the innovative graphical user interface, and moving on to the backend after this is mostly complete.

3.2 Graphical User Interface

Taking inspiration from the Virtual Vulcan interface, it must be extended to support multiple boards and to allow a wider range of components to be placed on the boards themselves, as the laboratories do not use logic trainers as featured in its interface.

In this section, the importance of getting the methods for the placement of wires correct is examined together with details of the various methods tested.

3.2.1 The Breadboard Metaphor

In the student labs circuits are composed on multiple breadboards (also know as plug boards) (Figure 3). These feature internal connections (Figure 4) so that no soldering is necessary thus allowing rapid prototyping of circuits. The breadboards themselves offer a convenient way of partitioning the circuitry, allowing sub-circuits to be built up on separate boards and then combined in the final implementation.

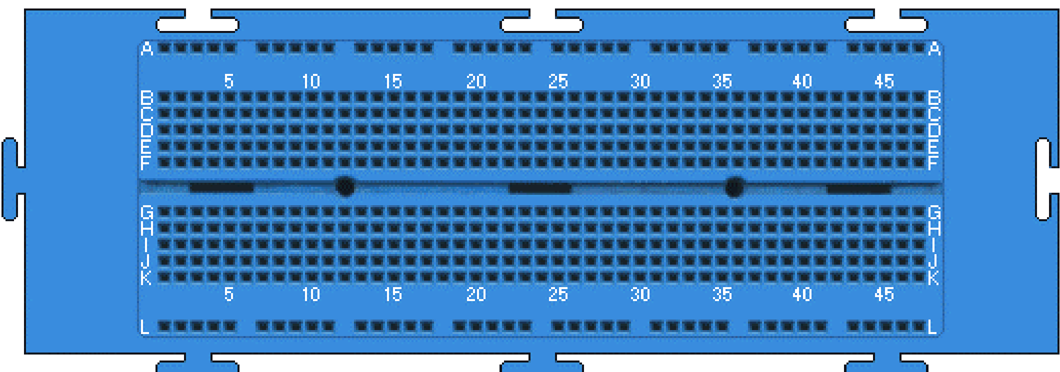


Figure 3: The Breadboard

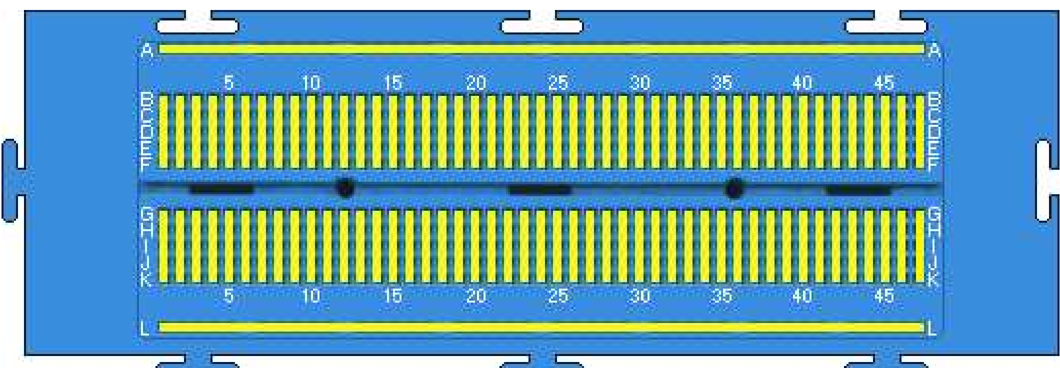


Figure 4: The Breadboard with internal connections shown

An intuitive interface will want to mirror this way of working. As well as requiring multiple breadboards in the workspace at any one time, the boards themselves should

be able to be repositioned within this space so that boards with associated circuitry can be appropriately placed.

When creating circuits; selection, movement, placement and removal are the basic physical operations available to a person for interacting with objects, and these translate easily over to their computer equivalents in a typical mouse driven application.

3.2.2 What's in a wire. Part 1. Drawing Connections

Wires form the primary type of connection in the circuits. With circuits that require many connections, it is important that these can be created and modified easily. There are two important aspects here, the graphical appearance of the wire and the way in which the user is required to interact with the system to add wires to the circuit.

Again, the primary influence was Virtual Vulcan. Unlike the other graphical simulators that draw wires as narrow lines, Vulcan makes them appear closer to real wires, and gives the user choice of colouring, to allow wires of similar function to be visibly grouped. To be consistent with the chosen metaphor, this approach must also be taken.

Unfortunately, Virtual Vulcan's method for placement of wires is less satisfactory. This requires the user to hold down the mouse button while dragging. The path of the mouse describes the path of the wire. When describing the wire with the motion of the mouse, it is difficult to keep to a straight line causing the results to look untidy. Therefore, other options are investigated.

The most basic notion is that a wire can be represented in a straight line between the two points to be connected. This is easily implemented by rotation and scaling of a wire graphic. However, again there are issues. In a circuit with several wires, it is likely that using this method will result in wires crossing each other as well as the components, which makes the circuit appear cluttered and difficult to understand.

This method also raises an implementation issue. An object in Java that receives input from the mouse will be able to receive it in an area of a rectangle whose extreme corners would be determined from the endpoints of the wire (Figure 5). Therefore, if it is wished to use mouse interaction on the wires as well as other objects, a wire that does not visually conceal a component, may still interfere with the user interacting with it.

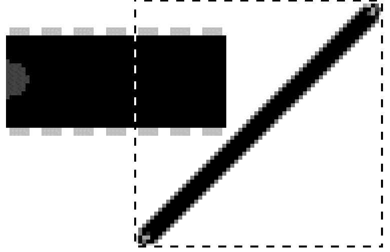


Figure 5: A diagonal wire, its influence on the mouse and how it may interfere with a component

The obvious restriction to make is, to only allow wires to be only completely horizontal or vertical. Connections can then be made by using the internal connections on the breadboard and multiple wires to describe the required path. It now becomes much easier for the user to customise the path of the wiring, allowing circuits to become more readable. However, many times more wires are required to achieve the same task.

Having to create these additional wires is both frustrating to the user and may increase the complexity of the circuit for simulation. Thus, it is necessary to replace these collections of wires with a single wire, which although still travels only horizontally and vertically, can now be bent to switch the direction of travel mid-wire. Conveniently, this is how well disciplined circuits should be created in the laboratories thus returning to the idea of mirroring real life practice in the software.

In fact, it would be possible to go much further, by taking the onus of routing wires away from the user and performing this operation automatically. However, as well as deviating from the metaphor of representing real world practice, this is complicated to achieve as it will require path finding algorithms to avoid components that must be re-evaluated for all the wires when extra components are added or moved. This could also present a problem when a wire has to be repositioned because of a component having been adjusted, as the sudden change of the wire position could confuse a user.

3.3 Simulation Engine

Having decided on the format of the user interface there is a need to design the simulation portion of the program and ensure that it interfaces with the front-end. Decisions are made as to how connections will be represented and what these will link to. There is also a need to formalise the representation of events for the simulation together with how these interface with devices.

3.3.1 Nodes in the circuit

Before dealing with making connections, an understanding of exactly what is being connected is needed. The usual approach, taken with circuit simulators, is that circuits are represented as undirected graphs with devices as nodes, and connections as the graph's arcs.

However, the circuits are created on breadboards, which affects this representation. Wires connect to the breadboard's sockets and not the components themselves. This means it is possible to have portions of circuit that are not yet connected to any devices. These still need to be included in the representation so that they may be probed appropriately.

Thus, the nodes in the circuit graph will represent the sockets on the boards, for a connection will always be to a socket and sockets may then be attached to the devices.

3.3.2 What's in a wire. Part 2. Simulating Connections

Having decided what the basic units in the circuit are, there is now a need to represent connections between them. Connections can be both wires, drawn by the user, and the internal connections on a breadboard.

Each socket will contain, within it, a representation of its electrical potential. It will be assumed that wires have no resistance to keep the model simple. A connection can therefore be modelled as the unification of two nodes potentials.

The obvious data structure for this model, is that of a set, as unification is a standard set operation. However, although the creation of connections then becomes trivial by unifying sockets potentials, when removing connections, information about which sets of sockets were unified, by the deleted connection, will also be needed, so they may be separated again. A solution to the problem may be to create the sets when simulation starts, however, in a circuit with many connections this operation will become computationally expensive, additionally to simulate switches during simulation some form of dynamic structure will be necessary to avoid further reforming as connections are made and broken. Thus, a different data structure is required.

By noting that each connection will always unify exactly two sets, it is possible to represent a connected set of sockets as a binary tree. Each element of the tree has two children, which represent the two sets that are connected. Thus, the leaf nodes will contain the actual sockets, as they are the basic node type in the model. Connections may now be formed by creating a new root node whose children are the two trees to be unified.

Deletion of a connection is now the separation of sub-trees. However, removal in this way will break the connection between the deleted nodes ancestors and its leaves. Thus, after removing a node from the tree its ancestors will have to be updated. This is achieved by storing, at each node in the tree, the two sockets that it physically connects. When a connection is removed, its ancestors in the connection tree will also be removed and then reattached to the new roots of its two sockets' trees and thus correctly rebuilding connections.

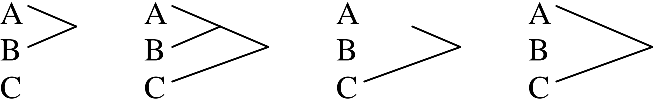


Figure 6: Creating and Removing Connections

Figure 6 shows the proposed solution in action. Initially A and B are connected. The new connection is then made between A and C. This appears in the data structure as a new root that connects C to the parent of A and B. The initial connection is then removed. This leaves A and C connected incorrectly. The ancestors of the removed connection are then updated, repairing the connection between A and C.

The structure can be extended so that each node in the tree stores its own potential. The propagation of the potential up the connection tree to the root, means that when one of the leaves is changed, will result in all nodes on the path to the root receiving a copy of the potential from that source. Thus, connections may now be broken during simulation and leave the newly created sub-trees pointing to the correct potentials, without having to recalculate from all the leaves. This will allow switches to be added without considerably adding to the complexity.

3.3.3 The Device Model

Before design of the simulation can begin, an understanding of the devices to be simulated is needed. A device is an object that performs a function. In terms of electronics, a device also features inputs and outputs for connection to the rest of the circuit. While the function will be different for each device, a common representation is required for these ports.

Formally, the ports of the devices are what connect them to the board sockets. Both the socket and the device will need to register the connection. The socket needs to store the device that is connected to it whilst the device needs to be able to read and write values to the socket.

A port may also take one of several types and initially the following are identified: input, output and input/output. These are interesting for two reasons, firstly, it would be useful to limit undefined behaviour in the simulation by not allowing connections between multiple outputs. Secondly, keeping track of the port types could also be used to check that models added to the system are not performing illegal operations such as writing to an input port.

The port's type can be stored in the sockets and connection trees. By propagating the type up the trees when new connections are made, it is possible to tell if any outputs are already in the connected set so that the user can be informed if it tries to connect multiple outputs. Enforcing this behaviour necessitates the introduction of two

additional port types; not connected and open-collector outputs. The not connected type is required to indicate when a connected set does not connect to any devices by storing this value at sockets with no connection. Open-collector outputs are a special type of output that do not produce undefined behaviour upon connection, thus the distinction from the regular output type is required to allow circuits with connected open-collectors.

The device model can be implemented as a Java interface. By forcing the actual devices to extend from this, it is ensured that there is a common method for access to all of them.

3.3.4 The Chip Model

A plug-in based architecture is required for the chips so that the program is easily extensible. However, to ensure that the user can interact in the same way with each of the chips their device models must be identical. Therefore, an extra layer to parameterise each chip and provide its function is required, known as the chip model.

Modular simulation programs often use a Hardware Descriptive Language (HDL) for specifying components. There are many different HDLs but the most commonly used are Verilog and VHDL (Very high speed integrated Hardware Description Language). Both have open specifications allowing for an interpreter to be added to the program, but HDL implementations of devices, although vast in number, are often commercially based and costly. Additionally, both languages are large so writing a full interpreter for either is beyond the scope of this project.

Instead of using a HDL, Java was chosen. This removes the necessity for producing an interpreter for the plug-ins, as the Java compiler will do this. The result can then be loaded using Java's dynamic linking facility. This method is also beneficial, as it simplifies the design and implementation of the program as a common language is used throughout. A further advantage is that if a HDL is required at a later stage, then an interpreter can be created as a Java plug-in. This opens up the possibility of supporting multiple HDLs, and therefore, providing greater numbers of components for the application.

3.3.5 The Discrete Simulation

The discrete simulation must be of an event-driven type because it cannot be guaranteed that circuits created by students will not exhibit asynchronous behaviour. Additionally, to enable accurate behaviour of the circuit elements with respect to time, a multiple delay model is required to model the various gate delays.

Therefore, the basis of the simulation is a time ordered event queue. The nanosecond is chosen for the unit of time as inspection of typical device datasheets [28] indicates delays of multiples of nanoseconds.

The simulation will initially only represent three states, low and high logic values together with a value representing that the node is not connected. Not connected is an unusual value in logic simulations, as generally circuit components are automatically connected to power supplies, so always output a legal value. However, because of the chosen graphical interface, components must be connected manually to power rails by

the user, which provides a condition where devices are not necessarily connected and their outputs must be able to reflect this.

To start the chain of events processing initial events are required to initialise the circuit. By modelling power supplies, initialising these is enough to start the event process, as it will cause all devices that are connected to the supplies to be evaluated.

4 Implementation

The Java Digital Breadboard Simulator comprises of over six thousand lines of code and over 27 classes for the main program alone. As such, the entire implementation will not be covered here and instead the focus will be on the more interesting and important aspects of the program.

4.1 User Interface

Figure 7 shows the implementation of the main program window, including at least one of each type of component.

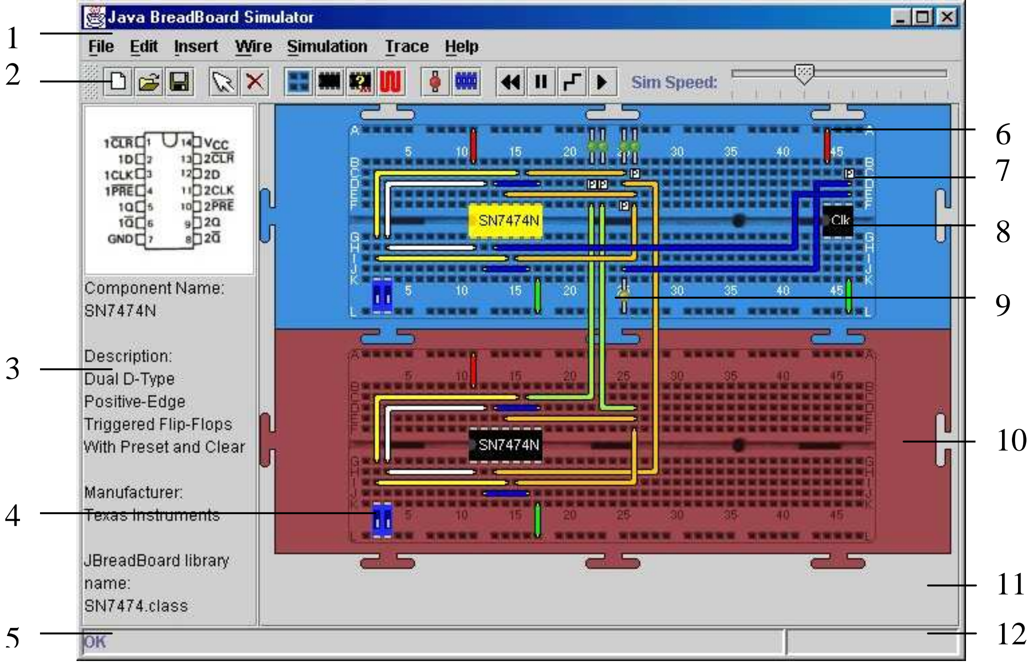


Figure 7: The main program window

Almost all interaction in the program is through this main window, with the only exceptions being the dialogs to select chips, colours of wires, and the help system. Thus, the main window is the most important graphical interface of the program and is described in further detail below.

1. The Menu. A standard component of most graphical user interfaces, the menu provides the user with the possible commands for the application. Keyboard accelerators are provided for the more common options to speed the user in their working. The functions provided by the menu are documented in the program's user guide available through the help menu. The user guide can be found in appendix 8.1

2. The Toolbar. Another common component in many graphical interfaces the toolbar provides easy access to the more common menu options. The toolbar also provides a slider for controlling the speed of the simulation. The toolbar is detachable and can be placed appropriately for the users convenience. The

button actions are documented in the program's user guide available through the help menu. The user guide can be found in appendix 8.1

3. The Selection Pane. This panel provides useful information about the currently selected component, including pin diagrams for the various chips. The selection pane makes the creation of circuits much easier by providing a useful on screen reference.

4. A Dipswitch Component. These are the only types of switch available in the Java Breadboard Simulator, although they come in several sizes. They are controlled by the mouse.

5. The Status Bar. The status bar is used to communicate messages to the user. These include things like on screen instructions for wire drawing and to show the current time in the simulation.

6. Wires. Wires are created with the mouse by selecting wire mode, from the menu or toolbar, then clicking on a hole on a breadboard. They may only be drawn horizontally or vertically but can be bent to provide any required path. The wires colour is selectable from the menu and is customisable with a colour chooser.

   Unlike devices, which are added onto the boards, wires are drawn on a layer above the boards allowing them to leave the board's boundaries and connect to other boards.

   The wire drawing method, as described in the design section, was extended to allow mistakes to be more easily corrected. During the drawing of a wire, pressing the escape key undoes the last bend that was inserted in the wire. Further presses unravel more wire allowing any part to be repositioned.

   A second extension to the wire drawing method enables wires, that have already been placed, to be editable via a double click on either end. This then reinitiates the wire drawing mode continuing from the selected end.

7. Probe. A device that produces a trace of the simulation. The output from probes can be imported into a statistical package, such as Microsoft Excel, to assist in analysing the circuit.

8. Chip. The chip component can draw a chip with any number of pins and two different widths according to its underlying model. A variety of chips are initially provided.

9. LED. Light Emitting Diodes are the main output devices for the simulator. Like real LED these are useful for viewing the status of parts of the circuit. Unlike real LED they do not require current limiting resistors to protect them. LEDs are available in red, yellow and green.

10. The Breadboard. Breadboards are used to build circuits on. As many breadboards as required may be used. They are placed automatically upon

insertion in the circuit, but providing they are not connected to other boards by wires, they can be moved freely.

For convenience, breadboards come pre-connected to the power supplies with +5V (logic high) along the top rail and 0V (logic low) on the bottom.

11. The Circuit Pane. This is the main working area for the Java Breadboard simulator. When the circuit gets too large for the panel, scrollbars are added to enable further expansion.

    All devices including boards can be selected with a mouse button press and can be moved by dragging the mouse. No assumptions are made to the dominant hand of the user or the number of buttons featured on their mouse, therefore all mouse interactions in the circuit pane can be performed with any mouse button.

12. The Logic Probe bar. During the simulation, clicking the mouse on the board sockets will produce a reading in this area. This is useful for debugging circuits, by reducing the need to add LED to all points in the circuit.

4.2 The Simulation

It has been shown [29] that with the chosen model it is possible for a new event to be scheduled before an existing event at the same node. In such a case, the old event must be cancelled to prevent its processing leaving the circuit in an incorrect state. To account for this effect, a queuing algorithm has been implemented, to first remove all events that occur after the new event that are scheduled at its node.

The event-driven simulation allows for three modes of operation: Stepping and animating continuously are controlled by the user via the interface whilst running for a specified time assists in the creation of animation.

Stepping through the simulation is the most basic type of event operation and is used in building the other two modes. Stepping the simulation results in only the next events being processed, so allows for the most finely spaced events to be watched.

The ability to run the simulation for a specified duration is not an option available to the user, but instead is used to build the continuous animation mode. The simulation is stepped until the specified duration has passed or there are no more events to be processed. Simulation runs in less than real time, thus simulating a long (in the order of seconds) will therefore take a considerable time to process causing the program to appear to freeze. A time-out of one millisecond is imposed to keep the animation flowing.

Animating the circuit allows the user to view the operation of the circuit over time. This method utilises the Java Timer class to run the simulator for a variable length of time. The rate of simulation is specified by the slider component in the graphical user interface. Animation relies of regular calls to the run-for-duration method described above, so if the speed is set too high, such that the computer cannot cope with the rate

of simulation, the run method will time out and instead of achieving the selected speed the simulation will proceed only as fast as is possible for the host computer.

4.3 Chip Models

As previously mentioned, chip models are implemented in Java. Once compiled the models can be loaded into the program with a call to Class.forName(String), which finds the specified class file and links in the code. Thus, the implementation is concerned with creating an interface for these Java class files.

The interface consists of sixteen Java methods that must be implemented for each model, which control the simulation, appearance, description and derivatives of the chip as shown below:

```
package jbreadboard.v1_00;
public interface ChipModel {
        public void setAccess(jbreadboard.ChipAccess a);

        //simulation
        public void simulate();
        public void reset();
        public String getPinType(int i);

        //appearance
        public int getNumberOfPins();
        public boolean isWide();
        public String getChipText();

        //description
        public String getDescription();
        public String getManufacturer();
        public String getDiagram();

        //derivatives
        public String[] getDerivatives();
        public int getDerivative();
        public String[] getPackages();
        public int getPackage();
        public void setDerivative(int t);
        public void setPackage(int p);
}
```

There are three methods concerned with simulation. The simulate method is most important for the interface as its body provides the function of the chip. It is called each time the signal at one of the chip's input pins changes.(determined by the getPinType(int) method)

The reset method is called when the simulation is reset. This method allows for any state variables associated with the model to be initialised appropriately at the start of simulation.

The interface provides three methods for limited control of the appearance of the chip. It is possible to specify the number of pins, one of two different widths and the text to appear on top of the chip. Specifying the number of pins is particularly important, as

the simulation will be checking that the model is only accessing valid pins during simulation.

The description methods provide the information seen in the selection pane of the application. The diagram returned from the getDiagram() function is especially useful as it provides the diagram for the selection pane that enables the user to see how the chip should be connected in the circuit.

By allowing chips of the same basic type, with possibly different pin assignments or timing in the same class, the derivatives methods vastly reduce the number of implementations required to model a range of chips.

To provide the chip implementer with a standard method of accessing the program interface a second class was created. This is called the ChipAccess class and is passed to the chip model via its setAccess method shortly after an instance of the chip is created. As well as providing methods to allow the chip implementer access to write values through the pins, the ChipAccess class also provides them with the current simulation time and methods for loading and saving data.

In addition to providing methods for the implementer, the ChipAccess class also features a more subtle secondary role. The ChipAccess class is the only object from the main application that the chip model can identify and it does not provide any further way of accessing the rest of the application. The result is that the ChipAccess class insulates the application from the effects of a rogue chip model.

The procedure for creating new chips is therefore as follows. Each chip is created as a separate class that must implement the ChipModel interface. This file must be declared in package chips to allow the simulator to be able to find the file once compiled, which is achieved by adding the line "package chips;" to the top of the file. This file may then be compiled with the command:

```
javac –classpath jbreadboard.jar NEWMODEL.java
```

where NEWMODEL.java is the name of the chip model Java file to be compiled and jbreadboard.jar is the jar archive containing the ChipModel interface which can be found in the Java Breadboard Simulators distribution package. The successful result of this operation is the production of a class file. By placing this in the chips directory of the Java Breadboard Simulator it will be found when the application is restarted.

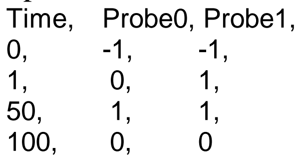
To accompany the initial release of the Java Breadboard Simulator, forty chips from the Texas Instruments SN74 range, typical of those used in the laboratories, were implemented. Two example implementations, those of the SN7400 and SN7470, can be found in appendix 8.2. The SN7470 implementation is of particular interest as it shows how both state and edge-detection can be implemented.

4.4 Signal Tracing

Producing a waveform output of the circuit is a useful way of getting a good view of how it changes over time. The Java Breadboard simulator does not feature its own graph drawing code and instead relies on other applications to draw traces for it.

To achieve this, it outputs a comma delimited file suitable for import into most spreadsheet packages. The first line of the file gives column labels for time followed by the probes in the circuit. The following lines start with the time value, then the value for each probe at that time in the order they were specified in the first line. The current probe values can be –1,0,1 for not connected, low and high respectively.

An example trace file is therefore:



```
Time,   Probe0, Probe1,
0,      -1,     -1,
1,       0,      1,
50,      1,      1,
100,     0,      0
```

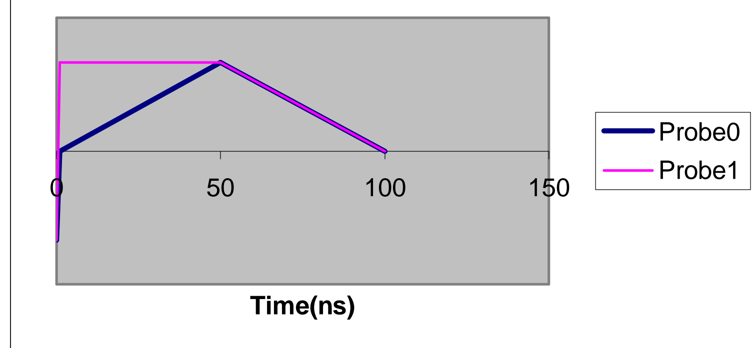


Figure 8: Example trace from Microsoft Excel

As can be seen in Figure 8, Microsoft Excel does not produce the sharp transitions expected in a digital circuit with this data. Therefore, an option was added to the program to modify the trace output such that both the previous and current values are given at each time step.

The new trace file is thus:

```
Time, Probe0, Probe1,
0,      -1,     -1,
1,      -1,     -1,
1,       0,     -1,
50,      0,      1,
50,      1,      1,
100,     1,      1,
100,     0,      0
```

Figure 9 shows the resulting graph.

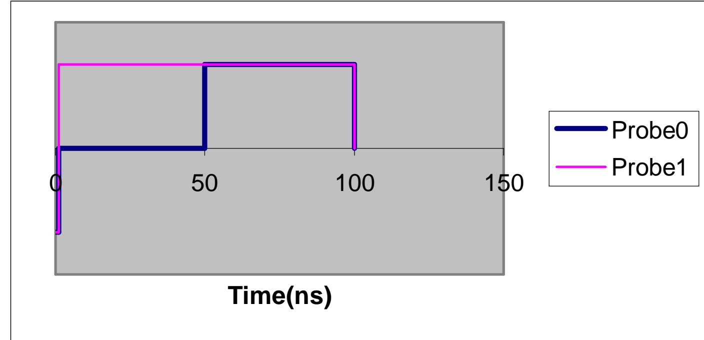


Figure 9: Improved example trace from Microsoft Excel.

4.5 Loading and Saving

By default, the Java Plug-in will not allow an applet to access data from the system on which it is running. This is to ensure that malicious code cannot harm the user's system. However, this also means that the simulator cannot perform operations such as loading and saving without each individual user granting permissions for the application. Java provides us with the ability to check if the relevant permissions are set for the application and so users can be told how to enable loading and saving if the permissions are insufficient.

Java permissions are granted with the use of a policy file. This is called ".java.policy" and is placed in the users home directory. Adding the following lines to the policy file will grant the Java Breadboard Simulator access to the file system:

```
grant codeBase "http://www-student.cs.york.ac.uk/~njg103/-" {
  permission java.util.PropertyPermission "user.home", "read";
  permission java.io.FilePermission "<<ALL FILES>>", "read, write";
  permission java.lang.RuntimePermission "modifyThread";
};
```

Although strictly the only the FilePermission is necessary to read and write files, the other two permissions are required by the Swing JFileChooser component.

Not all users will necessarily trust the application with file access so the Java Breadboard Simulator provides an alternative. With different permissions, the user can grant the Simulator the ability to read and write to the system clipboard, which is a potentially less dangerous operation. The policy entry for this is:

```
grant codeBase "http://www-student.cs.york.ac.uk/~njg103/-" {
  permission java.awt.AWTPermission "accessClipboard";
};
```

If the program detects that it has clipboard access but not file access then instead of presenting the user with a file chooser to load or save the file, a text component is shown to copy and paste the content of files. If both options are available the simulator defaults to the file chooser method.

Although inconvenient for basic file operations, the Java security model is particularly useful for the simulator's chip plug-in system. In terms of granting permissions, separate policies may be created for each plug-in, allowing new untrusted plug-ins to be safely tested on the system before their widespread use is allowed.

The ability to save via clipboard operations dictates that the circuit file format is textual rather than binary because binary data may not be reliably sent via the clipboard. Saving the circuit is used to preserve the elements of that circuit, so the file does just that, by listing the components and their positions. Each line of the file specifies a separate component.

The line starts with one of four basic types, Board, Device, Probe and Wire, followed by their parameters. Boards and devices are parameterised both by the Java class for the object and their x and y coordinates, with extra parameters on a per class basis. Probes have a label and coordinates. Parameters for wires are their red, green and

blue (RGB) colouring followed by pairs of numbers indicating the coordinates that the wire passes through. Devices are specified after the board that they are on and Wires are the last objects in the file.

The following file sets up a clock on a breadboard with a LED and a probe. The line numbers are added for the description that follows.

```
1    Circuit
2    Board jbreadboard.BreadBoard 0 0
3    Device jbreadboard.Chip Clock 0 7 76 72
4    Device jbreadboard.LED 84 128 0
5    Probe "probe0" 84 104
6    Wire 255 0 0 76 16 76 40
7    Wire 0 255 0 92 152 92 128
```

Line 1 specifies that this is a circuit file.
Line 2 indicates for a board implemented by jbreadboard.BreadBoard to be added to the circuit at (0,0).
Line 3 adds a chip device with model Clock.class and derivative 0 package 7 to the coordinates (76,72) on the above board.
Line 4 adds an LED of type 0 (red) to the above board at (84,128).
Line 5 adds a probe labelled “probe0” to the breadboard at (84, 104).
Line 6 specifies a wire with RGB colour 255,0,0 (red) that starts at (76,16) and ends at (76,40)
Line 7 is another wire, it has RGB colour 0,255,0 (green), starts at (92,152) and finishes at (92,128)

5 Evaluation of Program

Evaluating the program was broken down into two main tasks: ensuring the correctness of the simulation and assessment of the user interface.

5.1 Simulation Validation

Testing of the simulation end of the application started with the most basic elements; the connections and the devices, and then gradually built up to more interesting circuits.

5.1.1 Connection Verification

Ensuring that connections can be properly made is a trivial, but important, stage of the testing procedure. If the connections do not function correctly then it is not possible to rely on any other tests carried out, for even the most basic of circuits will require connections.

In order to test the connections, wiring and switches were added to a board and connected to both power rails and other wiring components. It was necessary to ensure that when the simulation was initialised that the values at the rails could be read at the other ends of the wiring connected them and that toggling switches made and broke further connections. Results of numerous combinations showed that the connections behaved as expected allowing any number of wires to reliably function.

It was also necessary to test that the program would not be able to introduce undefined behaviour into the circuit by allowing connections between different outputs. This was verified by trying to add wires and switches such that the supply and ground rails are linked. The application did not allow such connections, removing wires and not allowing a switch to toggle, if this resulted in outputs being connected.

Satisfied that the connection mechanisms seemed reliable the various devices available to the user were then tested.

5.1.2 Devices

A large number of devices were tested however, as the current selection of chips does not feature many inputs, it was possible to thoroughly test all input combinations of each of the 40 chips. The results of this test showed correct operation was achieved.

Another consideration with the device models was that they exhibit internal consistency as well as functional correctness. If a pin is either declared as an input or not connected, the model should then not be able to write to it as an output. The results of testing showed the program was able to detect this type of error, putting a warning on the screen detailing the inconsistency, and terminating the simulation run.

Testing of all input combinations has shown the program accurately highlighted any inconsistencies.

5.1.3 Circuits

By using circuits of well-documented behaviour as tests, it was possible to verify that the simulator functions on a larger scale, with correct timing and behaviour. As such,

a number of test cases were created to assess various aspects of the simulation. Circuit diagrams and screen shots of the test circuits can be found in appendix 8.3 illustrating the correct function of the program.

A basic combinatorial circuit was first used to test the simulator. A two-input exclusive-or circuit was set-up using the program. By running the simulator and trying all four input combinations (00, 01, 10, 11), it was possible to determine the state table (Appendix 8.3) for the circuit and verify that it matched that of the exclusive-or function.

Having verified basic combinatorial operation, more complicated circuits, which are harder to analyse, were then tested. This meant testing circuits that incorporated feedback, so flip-flops were created to test how the simulator coped.

The first feedback circuit examined was that of cross-coupled nand gates. This forms the most basic type of bistable the not-s not-r flip-flop. Running the simulation, it was again shown that the derived state table is that of the specified circuit. More interestingly, it is known that the not-s not-r can be made unstable by setting both inputs low and then high simultaneously creating an oscillation in the circuit. The simulation also confirmed this behaviour.

The second circuit featuring feedback tested, was a positive edge-triggered d-type flip-flop. This circuit combines three not-s not-r bistables into the single d-type flip-flop and testing it ensured that the simulation still behaves well with both feedback and increased complexity.

5.1.4 Timing

Having tested the simulators ability to handle complex behaviour it was also necessary to check that timing is modelled correctly. These tests were conducted using circuits whose timing properties are easy to analyse. Circuit diagrams and screen shots for these circuits can be found in appendix 8.3.

The first circuit tested had the function output = (Clock NAND (NOT Clock)). This function should always evaluate to true, however due to the propagation delay through the Not gate the actual circuit will not. When the clock rises from logic low to logic high both the clock and the output of the not gate will be high until after the not gate switches low. This will in turn send the Nand gate low. This behaviour was visible in the simulation in terms of the LED flicking off briefly. A plot of the saved trace (Figure 10) confirmed this behaviour showing the delays on the gates.

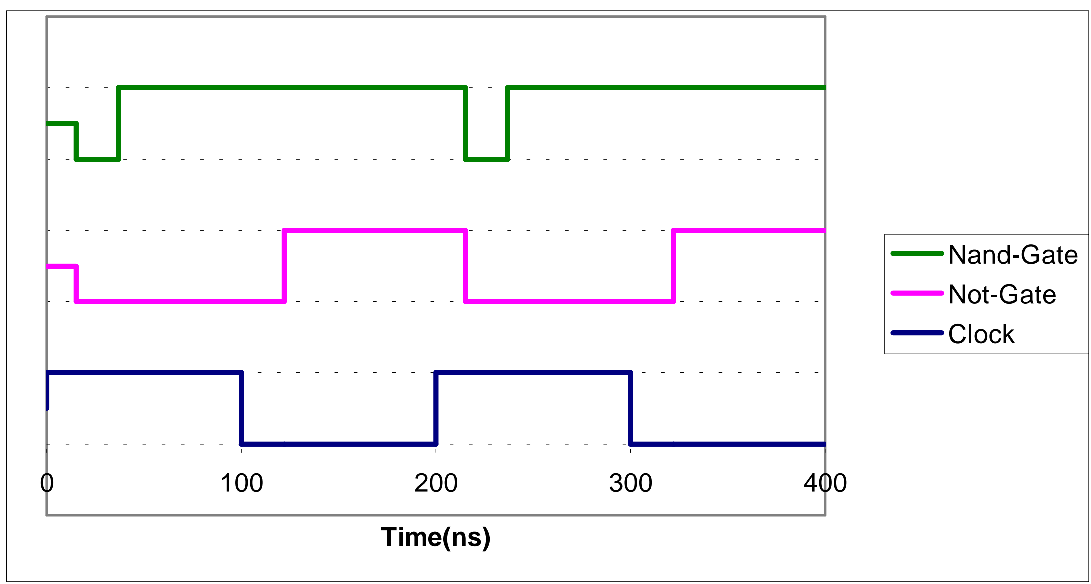


Figure 10: Timing analysis of circuit

The second timed circuit created an asynchronous four-bit binary counter. The circuit was created with four identical SN7470 JK-flip-flops set up to toggle and therefore divide the clock input by two. The propagation delay for one of these is 50ns, thus the propagation delay through all four flip-flops, for the most significant binary digit, is four times this. For the circuit to achieve the correct state while the input clocking pulse is still high the clock must be high for at least the length of this delay. Therefore, the minimum input clock period, assuming an even mark-to-space ratio, is 400ns giving a maximum clocking frequency of 2.5MHz.

Setting the input clock to 2MHz (Figure 11) and stepping through the simulation the circuit always achieved the correct output while the clock was high. However, by increasing the frequency to 3MHz (Figure 12) the most significant bit did was not set until after the clock input had dropped as expected.

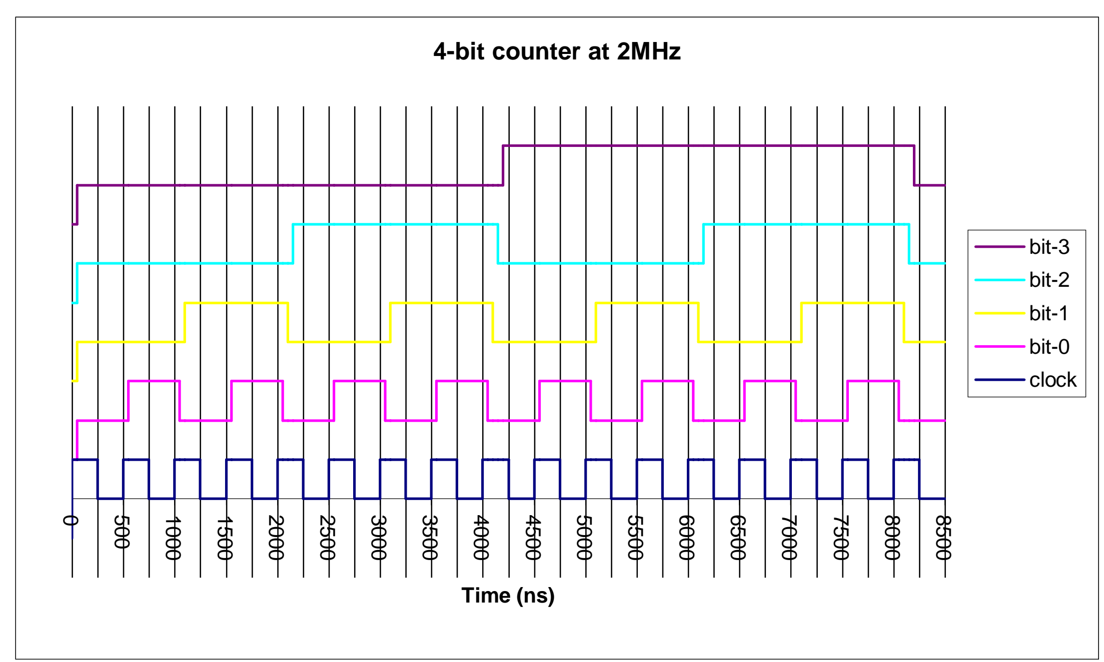


Figure 11: 4-bit counter with 2MHz input

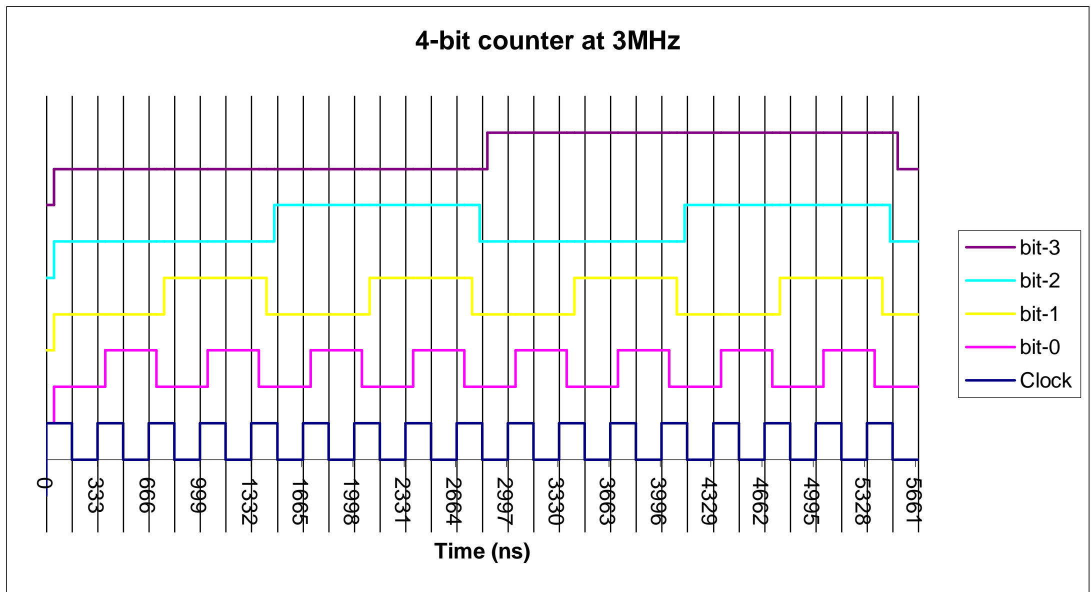


Figure 12: 4-bit counter with 3MHz input

5.1.5 Other Behaviour

The design had allowed for device models that can include both open-collector and IO ports. None of the tests already covered had checked that these can be simulated successfully. Unfortunately, no devices that required IO were initially identified to be implemented so it was not be possible to verify that IO ports connected together will function correctly. This will require further work in the implementation of devices that feature IO to be tested.

To verify open-collector functionality SN7407 buffers were used because they feature open-collector outputs. By connecting the outputs together, a “wired-or” circuit was created performing the function of a standard Or-gate (Appendix 8.3.6). As with

testing the devices, this was checked by going through all input combinations. Furthermore, this was confirmed to be extensible to multiple open-collector connections by then increasing the number of buffers in the circuit.

The results showed that with the exception of IO behaviour, for which it was not possible to test, the Java Breadboard Simulator behaved correctly for all types of circuit that it is possible to create.

5.2 Interface Evaluation

It is impossible for a designer to personally evaluate the interface of a piece of software as they will already be aware of all the various intricacies of the program and are thus biased towards it. Therefore, to properly assess the software students that have already completed the relevant courses were asked to evaluate the interface.

The students were asked to comment on ease of use and the feature set and to report any problems. They were also asked whether they felt the application would have been of use in their electronics courses.

In general, the program was well received and few bugs were reported. Problems that were mainly related to extensions in the wire drawing procedures have since been fixed.

Several students reported that they had no difficulties in creating and testing circuits, although on-screen instructions were appreciated for wire drawing, which was initially considered less intuitive, though with a little practice became easy to use.

Some problems were registered in realising that the breadboards came automatically connected to power supplies, as this is not actually stated until the user guide is read.

Board power supplies were also seen to be a problem to students who liked to connect separate board rails together to ensure that they receive common power and grounding. The application sees the rails as separate outputs and therefore does not allow them to be connected.

Some complaints were made over the operation of the switches. Users suggested preference towards different switch interaction methods such as dragging the switch, clicking on the switch instead of the gap or registering any click on switch or gap as a toggle.

Circuit probes were found to be the hardest components to use as they were only documented in the on-line help pages, which the students seemed reluctant to use. Those that managed to get as far as exporting probe data did not find that spreadsheets were well suited to producing the right type of graph easily.

Feature-wise no comments were received so it is assumed that the general opinion was that the application offers an acceptable feature set, which can only grow as more components are developed.

Many comments were received stating that if the application had been ready for this years courses that it would have been a useful tool for those students.

6 Conclusions

There are several strengths associated with the Java Breadboard Simulator.

Firstly, the extensibility of the chip model by the easy addition of further devices using a plug-in method allows the program to be highly applicable to the University's courses, permitting changes to course structure and the introduction of newly developed chips. This will ensure the long term usefulness of the program.

The interface is designed to be intuitive which again makes it applicable for University use as does not require expertise and keeps the focus on the actual system design, the fundamental part of the electronics course. The ease of use was confirmed by the testing conducted by students.

The simulation was shown, by testing, to work correctly and will therefore be a reliable method of testing circuits.

Further applicability to University use was provided by the ability of the program to be run through a web page. This prevents the need to install the program on vast numbers of student workstations.

The simulator should also be of use to other academic institutions, running similar courses, where licensing costs are not viable for the purchase of commercial software. Also, based on the research conducted as part of the study it appears as though web based Java systems are relatively unique in commercial environments and hence the simulator may be useful for retailers of electronic components, allowing their prospective customers to experiment with their products on websites before purchase.

Several problems were identified. Firstly, the lack of a graphical facility within the program reduced the efficiency of the application, as production of graphs which is vital for analysis of circuit performance, can only be achieved by transfer of data to a statistical package which is both complicated and time consuming.

A problem highlighted by testing was the dislike, by some students, of the current method in which interactions with switches occurs. This is to be expected due to the number of different methods available [30] however, a consideration for further work might be a method by which the interaction mode with switches could be selected.

The lack of copy and paste commands within the program could cause problems in large circuits in which particular features are regularly repeated. The consequence of this is the increased time necessary to test circuit designs and the potential for introduction of errors in multiple productions, of what should be identical sections.

Finally, because the current chip catalogue did not provide any chips with IO capacity for testing, the correct simulation of such devices cannot be confirmed without further work, which must be undertaken to ensure accurate working of the simulation in this area.

6.1 Future Work

Although the Java Breadboard Simulator is a useful tool to the Universities electronics courses, there are several areas in which could be improved by further work.

The biggest letdown in terms of the main program is that it relies on external packages for the plotting of waveforms. The ability to view these waveforms is very important to the analysis of a circuit and so the feature should be incorporated into the program proper.

To further the metaphor for representing the real world techniques in the program additional components need to be developed to provide power supplies, signal generators and cathode-ray oscilloscopes.

In addition to more chips from the Texas Instruments range, the addition of microprocessor and memory devices to the program would enable the simulation of the entire digital core of the student electronics systems and would be a unique feature for simulators of this class.

To provide access to many more device models, it may be useful to add an interpreter for the Verilog or VHDL hardware description languages.

It may be useful to provide a method to automatically translate the graphical representation used in the Java Breadboard Simulator to or from IEEE representation to assist in the documentation of circuitry or allow for easy transition between the two representations.

Finally, to be able to cover the entire range of circuitry used in the laboratories a circuit-level simulation needs to be developed, either as part of this application or as an accompanying program. The main reason for the development of the Java Breadboard Simulator was to allow development and testing of circuits outside of laboratory time and until the testing of all possible circuitry is possible, the requirements will not have been completely met.

7 References

[1] *Collins Paperback English Dictionary.* 4th ed. HarperCollins Publishers, Glasgow, 1999

[2] Roberts N. et al. *Computer Simulation: A system dynamics modelling approach.* Addison-Wesley, 1983

[3] Korn GA and Wait JV. *Digital Continuous-System Simulation.* Prentice Hall, New Jersey, 1978

[4] Hartley MG. *Digital Simulation Methods.* Peter Peregrinus Ltd., Stevenage, 1975

[5] Breuer MA. *Recent Developments in Design Automation.* Computer Vol 5, May/June 1972

[6] Rubin SM. *Computer Aids for VLSI Designs*. 2nd ed. Static Free Software, http://www.staticfreesoft.com, 1994

[7] Breuer MA. *Digital System Design Automation Languages, Simulation & Data base.* Computer Science Press, Rockville, 1975

[8] Miller E and Squire J. *esim: A Structural Design Language and Simulator for Architecture in Education.* Workshop on Computer Architecture Education, 2000

[9] Burch C. *Logisim: A graphical system for logic circuit design and simulation.* To appear in Journal of Educational Resources in Computing, March 2002, available http://www.cburch.com, 2001

[10] Arase K. *Simcir.* http://www.tt.rim.or.jp/~kazz/simcir, 2000

[11] Boothe B. *Probe.* http://www.scit.wiv.ac.uk/~cm1970/probe/webpage, 1999

[12] Craig D. D*igiTCL.* http://www.cs.mun.ca/~donald/digitcl, 1997

[13] Eck D. *xLogicCircuits.* http://www.math.hws.edu/TMCM/java/xLogicCircuits/index.html, 2000

[14] *LogicSimulator.* Gordon College. http://www.cs.gordon.edu/courses/cs111/module7/logicsim/example1.html, 2001

[15] Herz A. *DigitalSimulator r3.2.* http://www.ttl-simulator.de, 2001

[16] Karweit A. *Circuit Builder.* John Hopkins University. http://www.jhu.edu/~virtlab/logic/logic.htm, 2000

[17] Knaian A. *Digital Simulator 1.1.* http://web.mit.edu/ara/www/ds.html, 1994

[18] Masson A. *LogicSim.* http://wuarchive.wustl.edu/edu/math/software/mac/logic/LogicSim, 1996

[19] *DigitalWorks.* Mecanique. http://www.mechanique.co.uk/digital-works, 2001

[20] Easysim. Research Systems Pty Ltd. http://www.research-systems.com, 2001

[21] *Multimedia Logic.* Softronics Inc. http://www.softronix.com, 1997

[22] Tetzl A. *LogicSim.* http://www.tetzl.de, 2001

[23] van Rienen I. *DigSim.* http://www.bart.nl/~ivr, 1996

[24] Zidar F. *Digital Circuit Simulator.* http://www.rocketdownload.com/Details/Home/925.htm, 2001

[25] *Virtual Vulcan.* Yoeric Software. http://www.yoeric.com, 2000

[26] *Overview of IEEE Standard 91-1984: Explanation of Logic Symbols.* Texas Instruments, http://www-s.ti.com/psheets/sdyz001a/sdyz001a.pdf, 1996

[27] Walrath K. and Campione M. *The JFC Swing Tutorial: a guide to constructing GUIs,* 3rd Printing. Addison-Wesley, 2000

[28] Texas Instruments SN74XX range, www.ti.com.

[29] Agrawal P. and Daily W. *Algorithms for accuracy enhancement in a hardware logic simulator.* Proceedings of the 26th ACM/IEEE Design Automation Conference, 1989.
[30] Benest ID and Olivier PL. *User-Interface Design Engineering Course Slides.* 3rd ed. Department of Computer Science, University of York.

8 Appendices

8.1 User Guide

The user guide details the function of the various menu and toolbar entries. The toolbar entries are denoted by the icon next to the equivalent menu item.

File Menu

File-New (Ctrl-N)	Clear current circuit.
File-Open (Ctrl-O)	Open a circuit file. See Help-Loading and Saving for more details on enabling loading and saving.
File-Save (Ctrl-S)	Save the current circuit. See Help-Loading and Saving for more details on enabling loading and saving.
File-Exit	Quit the program.

Edit Menu

Edit-Selection mode (Insert)	Switch to Selection mode. In this mode, you can move boards and components.
Edit-Delete (Delete)	Delete currently selected object, or the wire that is currently being drawn. Will not work in simulation mode.

Insert Menu

To insert anything but a breadboard, the circuit must contain at least one breadboard that has been selected (coloured blue).

Insert-Breadboard	Insert a Breadboard. The breadboards are placed automatically but can be moved as long as they are not connected by wires to another board. The Breadboards power rails are connected automatically with 5v along top and 0v along bottom.
Insert-Chip	Brings up a dialog of all the available chips. The selection will be placed onto the currently selected board, but can be moved to others. See here for information on creating additional chips.
Toolbar-Chip	Insert a chip identical to the last one added.
Insert-Dipswitch	Insert a dipswitch. The menu will allow you to select the size of the dipswitch. The toolbar just defaults to the last value used. The Java Breadboard Simulator tries hard not to let you connect separate outputs together. If switching a dip on will connect multiple outputs, the application will not let you turn it on. Dipswitches do not have effect immediately. The simulation must be advanced after changing values. This is to allow more than one input change at any given time. It is probably not wise to connect a dipswitch to the positive power rail. This is because all the current chip inputs float high. This will result in the input always being high whether the switch is on or off. Instead try using the negative power rail and combining the switches with a buffer or an inverter depending on the direction of the switch you desire.
Insert-LED-Red	Inserts a red LED.

Insert-LED-Yellow	Inserts a yellow LED.
Insert-LED-Green	Inserts a green LED.
Toolbar-LED	Inserts a LED identical to the last one added.

Wire Menu

Wire-Add Wires	Switch to Wiring Mode. In wiring mode clicking on a board will start drawing a wire. Wire sections can only be horizontal or vertical, but a single click while drawing will insert a bend in the wire. Double clicking on a board will terminate the wire. Wires must start and end in valid positions on a board. The rest of the wire is not constricted, and wires may cross boards. In any mode except for simulation, double clicking on the end of a wire will also switch the program into wiring mode, extending from the selected wire.
Wire-Cancel Wire Segment (Escape)	While drawing a wire, this option will undo the last corner. If there are no corners to undo then the wire is removed.
Wire-Hide Wires (Ctrl-H)	Wires can go anywhere, including over components, so it is easy to lose things under wires. Hiding wires will hide all the wires in the circuit so that objects under wires can be selected again.
Wire-Unhide Wires (Ctrl-U)	If the circuits wires are hidden this menu item will show them again.
Wire-Colours	The rest of the items in the wire menu control the colour of the next wire to be drawn. The custom option will bring up a colour chooser allowing selection of any colour for a wire.

Simulation

Simulation-Reset Simulation (Backspace)	Reset simulation, all components and probe traces.
Pause (Ctrl-T)	Pause the currently running simulation.
Simulation-Step Simulation (Enter)	Steps the simulation by executing all events at the next event time. If there are no events left in the simulation, the time will be advanced by 1ns.
Simulation-Run (Ctrl-R)	Animate the simulation.
Speed Slider	Adjust the speed of the running simulation. The slider follows a logarithmic scale with the markers denoting powers of 10 nanoseconds with 10^0=1ns on the left and 10^9=1,000,000,000ns=1s on the right. If the speed is set too high the application may not be able to run at the selected speed and will instead just go as fast as it can.

Trace Menu

Java Breadboard Simulator does not provide any graphing tools of its own, however it can export data that can be imported into most spreadsheet programs to produce wave outputs.

Trace-Insert Probe	Inserts a probe device.
Trace-Save Trace	Saves the trace up to the current position in the simulation. The format is that of a comma delimited file, with each time on a new line. The potentials are represented by values with -1=NC, 0=LOW, 1=HIGH eg. Time,NameOfProbe0,NameOfProbe1,... 0,0,0,... 10,1,0,... ... See Help-Loading and Saving for more details on enabling loading and saving.
Trace-Dual data at Time	If enabled modifies the output of the next trace such that each time the trace is updated the previous value is output at the current time, then the current value is output at the current time. eg Time,Probe0, ... 10,0, 10,1, 15,1, 15,0, ... This may help your graphing program produce traces with vertical transitions .

Help Menu

Help-User Guide	Brings up this page
Help-Loading and Saving	Brings up a page detailing how to enable loading and saving with the application
Help-About	Brings up the about box

8.2 Example Chip Implementations

8.2.1 SN7400 Quadruple 2-Input Positive-Nand Gates

```
/*
 * @(#)SN7400.java      1.0 14/03/02
 *
 * Copyright 2002 Nicholas Glass. All Rights Reserved.
 *
 */

package chips;

public class SN7400 implements jbreadboard.v1_00.ChipModel {
        private jbreadboard.ChipAccess chip;

        //Chip Information
        private final String Description   ="Quadruple 2-Input Positive-Nand Gates";
        private final String Manufacturer="Texas Instruments";

        //Diagram
        //NB. The SN5400W has a different pinout to the rest of the range.
        private String Diagram              ="SN7400.gif";
        private String Diagram5400W   ="SN5400W.gif";

        //Chip Size
        private final int NumberOfPins   =7;
        private final boolean Wide         =false;


        //Pin Types
        private final String[] pintypes      ={"IN","IN","OUT","IN","IN","OUT","IN",
                                               "OUT","IN","IN","OUT","IN","IN","IN"};
        private final String[] pintypes5400W={"IN","IN","OUT","IN","OUT","IN","IN",
                                               "OUT","IN","IN","IN","IN","IN","OUT"};

        //Derivatives
        private int Derivative                                   =3;       //7400
        private int PackageType                                  =0;       //N
        private final String[] Derivatives ={"SN5400",
                                               "SN54LS00",
                                               "SN54S00",
                                               "SN7400",
                                               "SN74LS00",
                                               "SN74S00"};

        //Package Types
        //Corresponds to the Derivative above
        private final String[][] PackageTypes={{"J","W"},
                                               {"J","W"},
                                               {"J","W"},
                                               {"N"      },
                                               {"D","N"},
                                               {"D","N"}};

        //Switching Characteristics
        private int Tplh() {
                if(Derivative==0 || Derivative==3) return 22;
```

```
		else if(Derivative==1 || Derivative==4) return 15;
		else return 5;
	}

	private int Tphl() {
		if(Derivative==0 || Derivative==3) return 15;
		else if(Derivative==1 || Derivative==4) return 15;
		else return 5;
	}

	//Simulation
	public void reset() {}

	public void simulate() {
		boolean powered=false;

		//pins
		int vcc=13;
		int gnd=6;
		int a1=0, a2=3, a3=8, a4=11;
		int b1=1, b2=4, b3=9, b4=12;
		int y1=2, y2=5, y3=7, y4=10;

		if(getChipText().equals("SN5400W")) {
			vcc=3;
			gnd=10;

			a1=0; a2=5; a3=8; a4=11;
			b1=1; b2=6; b3=9; b4=12;
			y1=2; y2=4; y3=7; y4=13;
		}

		if(chip.readTTL(vcc).equals("HIGH") &&
			chip.readTTL(gnd).equals("LOW")) powered=true;

		//Using !equals(low) instead of equals(high) causes inputs to float high
		if(powered) {
			if(!(!chip.readTTL(a1).equals("LOW")&&
			    !chip.readTTL(b1).equals("LOW")))
				chip.write(y1,"HIGH",Tplh());
			else chip.write(y1,"LOW",Tphl());

			if(!(!chip.readTTL(a2).equals("LOW") &&
			    !chip.readTTL(b2).equals("LOW")))
				chip.write(y2,"HIGH",Tplh());
			else chip.write(y2,"LOW",Tphl());

			if(!(!chip.readTTL(a3).equals("LOW") &&
			    !chip.readTTL(b3).equals("LOW")))
				chip.write(y3,"HIGH",Tplh());
			else chip.write(y3,"LOW",Tphl());

			if(!(!chip.readTTL(a4).equals("LOW") &&
			    !chip.readTTL(b4).equals("LOW")))
				chip.write(y4,"HIGH",Tplh());
			else chip.write(y4,"LOW",Tphl());
		} else {
			//standard power off events
			for(int i=0;i<getNumberOfPins()*2;i++) {
				String type=getPinType(i);
```

```
                        if(!(type.equals("IN") || type.equals("NC")))
                                chip.write(i,"NC",0);
                }
        }
}

public void setAccess(jbreadboard.ChipAccess a) {
        chip=a;
}

//Chip Information
public String getChipText() {
        return Derivatives[Derivative] + PackageTypes[Derivative][PackageType];
}

public String getDescription() {
        return Description;
}

public String getManufacturer() {
        return Manufacturer;
}

//Diagram
public String getDiagram() {
        if(getChipText().equals("SN5400W")) return Diagram5400W;
        else return Diagram;
}

//Chip Size
public int getNumberOfPins() {
        return NumberOfPins;
}

public boolean isWide() {
        return Wide;
}

//Pin Types
public String getPinType(int i) {
        if(getChipText().equals("SN5400W")) return pintypes5400W[i];
        else return pintypes[i];
}

//Derivatives
public int getDerivative() {return Derivative;}
public int getPackage() {return PackageType;}
public String[] getDerivatives() {
        return Derivatives;
}
public String[] getPackages() {
        return PackageTypes[Derivative];
}

//select the particular derivative
public void setDerivative(int t) {
        Derivative = t;
}
//select the particular package
public void setPackage(int p) {
```

```
		PackageType = p;
	}
}
```

8.2.2 SN7470 And-Gated J-K Positive-Edge-Triggered Flip Flops with preset and clear

```
/*
 * @(#)SN7470.java	1.0 14/03/02
 *
 * Copyright 2002 Nicholas Glass. All Rights Reserved.
 *
 */

package chips;

public class SN7470 implements jbreadboard.v1_00.ChipModel {
	private jbreadboard.ChipAccess chip;

	//Chip Information
	private final String Description	="And-Gated J-K Positive-Egde Triggered Flip Flops"
						+" With Preset And Clear";
	private final String Manufacturer="Texas Instruments";

	//Diagram
	//NB. The SN5470W has a different pinout to the rest of the range.
	private String Diagram			="SN7470.gif";
	private String Diagram5470W	="SN5470W.gif";

	//Chip Size
	private final int NumberOfPins	=7;
	private final boolean Wide		=false;


	//Pin Types
	private final String[] pintypes	={"NC","IN","IN","IN","IN","OUT","IN",
						 "OUT","IN","IN","IN","IN","IN","IN"};
	private final String[] pintypes5470W={"IN","IN","IN","IN","IN","NC","IN",
						 "IN","IN","OUT","IN","OUT","IN","IN"};

	//Derivatives
	private int Derivative				=1;	//7470
	private int PackageType			=0;	//N
	private final String[] Derivatives = {"SN5470",
						 "SN7470"};

	//Package Types
	//Corresponds to the Derivative above
	private final String[][] PackageTypes={{"J","W"},
						{"N"}};

	//Switching Characteristics
	private int Tplh				=50;
	private int Tphl				=50;



	//Simulation
```

```
//state variables
boolean state=true;
boolean previousclk=true;

public void reset() {
        state=true;
        previousclk=true;
}

public void simulate() {
        boolean powered=false;

        //pins
        int vcc=13;
        int gnd=6;

        int j1=2, j2=3, nj=4;
        int k1=9, k2=10, nk=8;
        int q=7, nq=5;
        int clr=1, pre=12;
        int clk=11;


        if(getChipText().equals("SN5470W")) {
                vcc=3;
                gnd=10;

                j1=6; j2=7;  nj=8;
                k1=0; k2=13; nk=12;
                q=11; nq=9;
                clr=4; pre=2;
                clk=1;
        }

        if(chip.readTTL(vcc).equals("HIGH") &&
        chip.readTTL(gnd).equals("LOW")) powered=true;

        //Using !equals(low) instead of equals(high) causes inputs to float high
        if(powered) {

                boolean j = (!chip.readTTL(j1).equals("LOW") &&
                             !chip.readTTL(j2).equals("LOW") &&
                              chip.readTTL(nj).equals("LOW"));

                boolean k = (!chip.readTTL(k1).equals("LOW") &&
                              !chip.readTTL(k2).equals("LOW") &&
                               chip.readTTL(nk).equals("LOW"));

                if(chip.readTTL(pre).equals("LOW") &&
                chip.readTTL(clr).equals("LOW")) {
                        //pre and clr
                        chip.write(q,"LOW",Tphl);
                        chip.write(nq,"LOW",Tphl);
                } else if(chip.readTTL(clk).equals("LOW")) {
                        if(chip.readTTL(pre).equals("LOW")) {
                                //pre
                                state=true;
                                chip.write(q,"HIGH",Tplh);
                                chip.write(nq,"LOW",Tphl);
                        } else if(chip.readTTL(clr).equals("LOW")) {
```

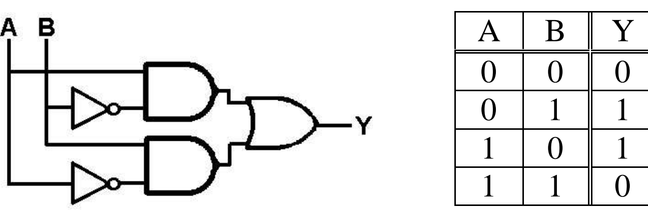
```
					//clr
					state=false;
					chip.write(q,"LOW",Tphl);
					chip.write(nq,"HIGH",Tplh);
				}
			} else if(!chip.readTTL(clk).equals("LOW") &&
			previousclk==false &&
			!chip.readTTL(pre).equals("LOW") &&
			!chip.readTTL(clr).equals("LOW")) {

				//positive going clock edge found
				//and pre and clr are not set

				      if(!j && !k) state=state;
				else if( j && !k) state=true;
				else if(!j &&  k) state=false;
				else if( j &&  k) state=!state;  //toggle

				if(state) {
					chip.write(q,"HIGH",Tplh);
					chip.write(nq,"LOW",Tphl);
				} else {
					chip.write(q,"LOW",Tphl);
					chip.write(nq,"HIGH",Tplh);
				}
			} else if(chip.readTTL(q).equals(chip.readTTL(nq))) {
				//found unstable state, so come out of it
				if(state) {
					chip.write(q,"HIGH",Tplh);
					chip.write(nq,"LOW",Tphl);
				} else {
					chip.write(q,"LOW",Tphl);
					chip.write(nq,"HIGH",Tplh);
				}
			}
			//update previous-clock store
			previousclk=(!chip.readTTL(clk).equals("LOW"));
		} else {
			//standard power off events
			for(int i=0;i<getNumberOfPins()*2;i++) {
				String type=getPinType(i);
				if(!(type.equals("IN") || type.equals("NC")))
					chip.write(i,"NC",0);
			}
		}
	}

	public void setAccess(jbreadboard.ChipAccess a) {
		chip=a;
	}

	//Chip Information
	public String getChipText() {
		return Derivatives[Derivative] + PackageTypes[Derivative][PackageType];
	}

	public String getDescription() {
		return Description;
	}
```

```
	public String getManufacturer() {
		return Manufacturer;
	}

	//Diagram
	public String getDiagram() {
		if(getChipText().equals("SN5470W")) return Diagram5470W;
		else return Diagram;
	}

	//Chip Size
	public int getNumberOfPins() {
		return NumberOfPins;
	}

	public boolean isWide() {
		return Wide;
	}

	//Pin Types
	public String getPinType(int i) {
		if(getChipText().equals("SN5470W")) return pintypes5470W[i];
		else return pintypes[i];
	}

	//Derivatives
	public int getDerivative() {return Derivative;}
	public int getPackage() {return PackageType;}
	public String[] getDerivatives() {
		return Derivatives;
	}
	public String[] getPackages() {
		return PackageTypes[Derivative];
	}

	//select the particular derivative
	public void setDerivative(int t) {
		Derivative = t;
	}
	//select the particular package
	public void setPackage(int p) {
		PackageType = p;
	}
}
```

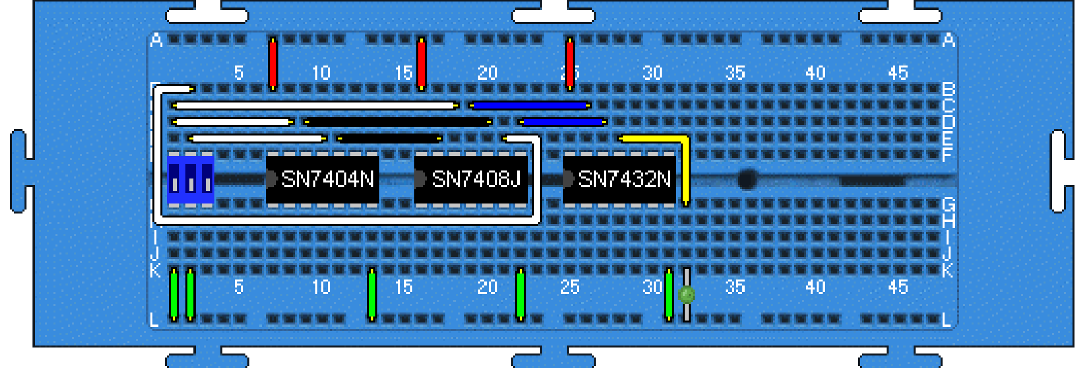
8.3 Test Circuits

The following pages contain the circuits that were used to test the Java Breadboard Simulator. A logic diagram, state-transition table and screen capture is given for each one. In the state-transition tables 0 represents a logical low, 1 represents a logical high.

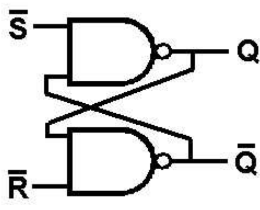
8.3.1 2-Input Exclusive-Or



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

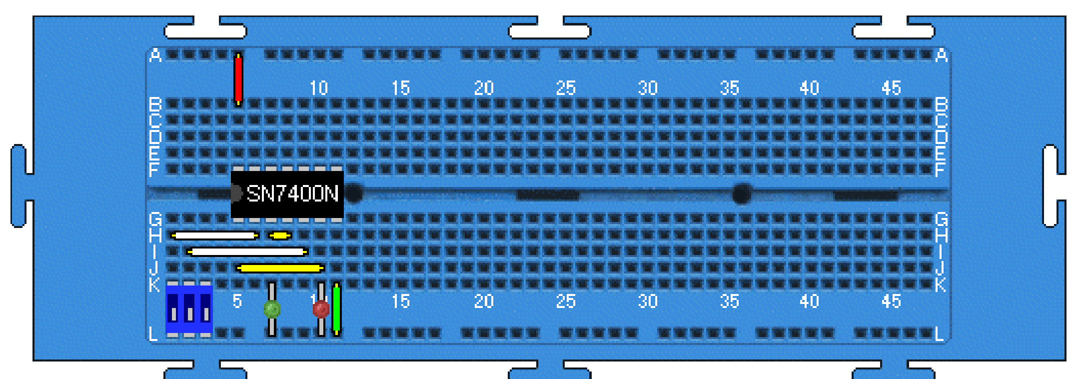


8.3.2 Not-S Not-R Flip-flop

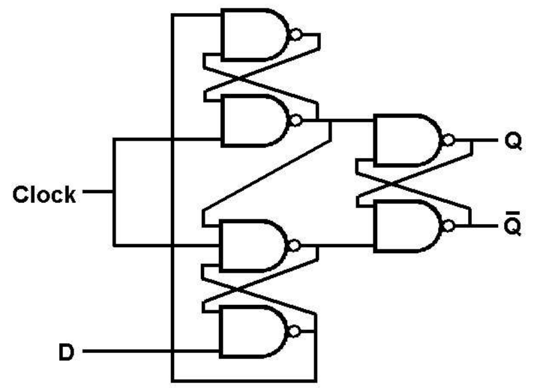


S	R	Q(t+1)	Not-Q (t+1)
0	0	Q(t)	Not-Q(t)
0	1	0	1
1	0	1	0
1	1	1*	1*

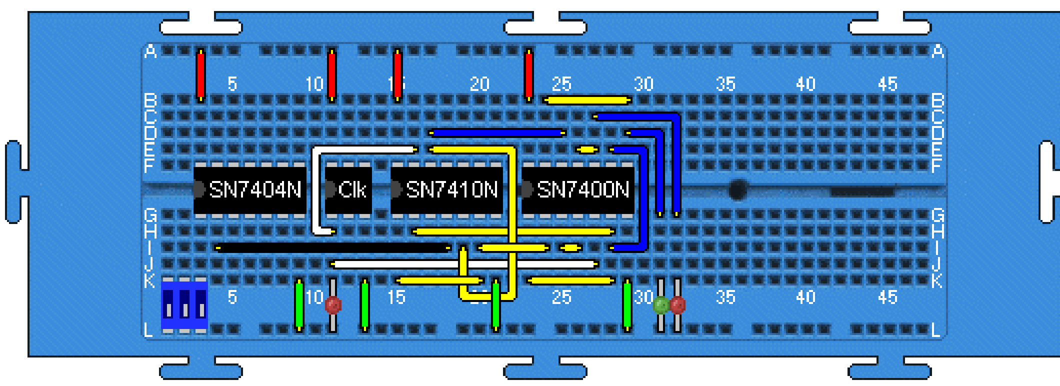
* If from this state S and R are set to 0 simultaneously Q and Not-Q will oscillate between 00 and 11.



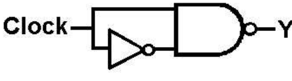
8.3.3 Positive Edge-Triggered D-Type Flip-flop



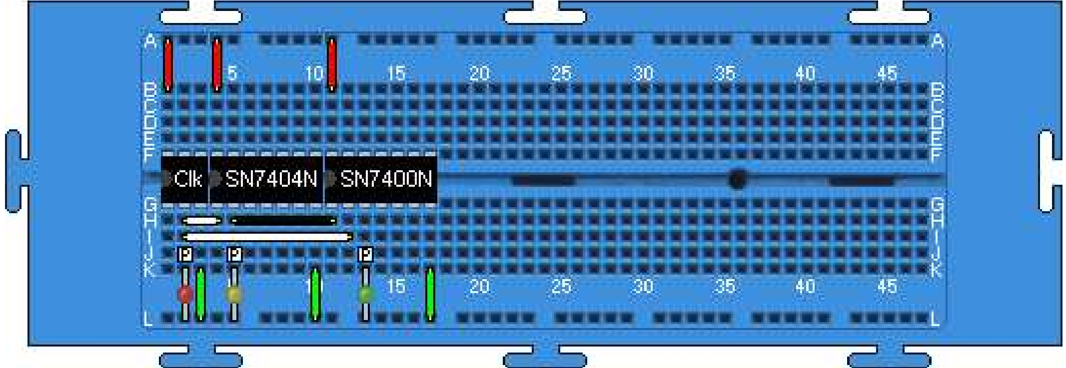
D	Clock	Q
0	0 to 1 transition	0
1	0 to 1 transition	1



8.3.4 (Clock NAND (Not Clock))

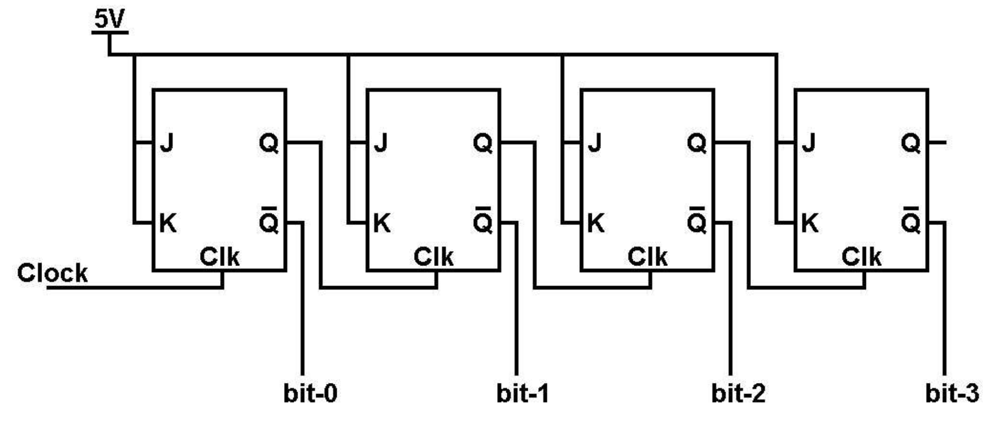


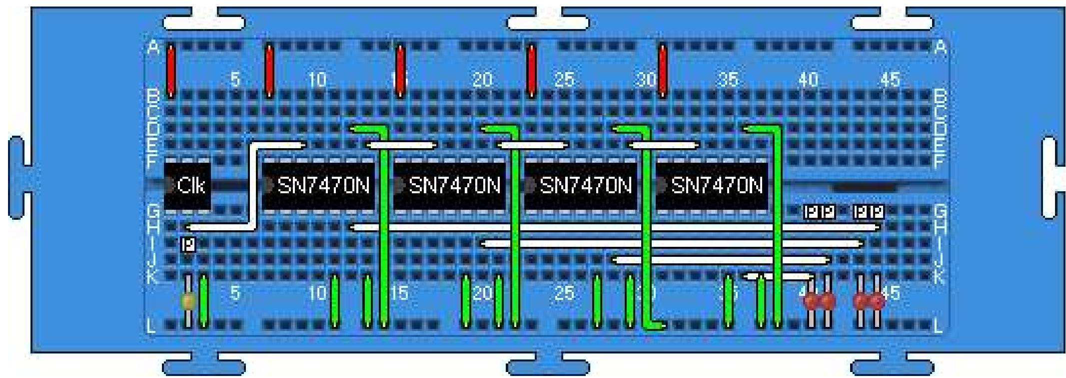
Clock	Y
0	1
1	1



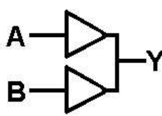
8.3.5 Asynchronous Four-bit Binary Counter

Outputs loop counting 0-15 in binary.





8.3.6 Wired-Or



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1