An Object-Oriented Circuit Simulation Library

Zeal Jagannatha

April 24, 2012

Abstract

In many computer science programs, the field of digital electronics is introduced within computer organization or computer architecture courses, with the help of graphical tools. However, little attention is given to text-based approaches. The C++ Circuit Simulation Library (CCSL), attempts to present an intuitive text-based interface for those with experience with C++, while still giving advanced users an efficient framework for digital circuit design and development.

1 Introduction

1.1 Purpose

The project is a logic simulator that is entirely programmed in C++. This logic simulator is meant to be used as a solid foundation and intuitive interface for digital circuit development in C++. The library should be easily usable by users from a variety of experience levels from as simple as wiring up a pulse generator to as complicated as building an ALU. In addition, it seeks to represent the interactions of real digital circuits as closely as possible. To this purpose, the project simulates circuit creation, connection and evaluation with evaluators and circuit bread-boxes. In addition, tools have been added that allow easy generalization and expansion of circuits, such as black-box components.

When pieced together correctly, basic components allow us to abstract out the ground-level components and focus on the functionality of modules of these components. These modules are the main focus of digital electronics within computer science. Computer scientists ignore the resistors, capacitors and other details, and focus purely on the modules, e.g. **and**, **or**, and **not** gates.

CCSL users are assumed to have a basic familiarity with C++ programming. Users wishing to use the library to design new digital components must have an understanding of C++ inheritance[7]. Finally, in order to fully understand use all aspects of the project, an understanding of algorithms and data structures is necessary. Specifically hash tables, garbage collection, and algorithms related to graphs are of particular interest.

In the remainder of this paper, words that are marked in `typewriter font` are class names and always begin with a capital letter. Words that are in *italics* are generally function names, and also begin with capital letters.

1.2 Intended Audience

Although the idea behind the project should be understandable to those without programming experience, the intended audience of this project is software engineers who wish to design and simulate digital circuits in C++. However, the user's level of experience limits what they are able to do with this project. There are three experience levels that have different methods of interacting with the project: introductory users, advanced users, and expert users.

Introductory users are users who are able to create and use simple circuits using pre-defined components but are unable to create their own components or extend the libraries in any way. This group of users has only a vague understanding of the traversal methods and design principles used by the project, and so, are unable to extend it completely. This group of users are around the level of introductory computer architecture or hardware courses.

Advanced users are users who can both construct simple circuits using predefined components, as well as create their own components by either extending existing components, or by creating brand new components from scratch. These users can also extend the existing iterators to their own uses. In addition to being able to create components, they have a good understanding of the principles used in the design of the project and can use this knowledge to extend their components and iterators. These users would be students who have completed courses in algorithms, project design, and computer hardware.

Expert users are users who are able to construct circuits from components, both predefined and of their own making, can extend existing iterators to create new ways of evaluating and interacting with circuits, and have a deep understanding of the project's design principles. These users are those who understand not only the principles which the project was design with, but have a good understanding of the internal mechanisms of the project as well.

1.3 Similar Projects

A variety of similar projects have been produced:

- Logisim: A graphical circuit simulator written in Java. The project was thorough and implemented many of the features which CCSL implemented including delay-based evaluation and black-box components.[4]
- KLogic: A graphical circuit simulator developed in C++ with the QT interface. KLogic allows black-box components, which were a crucial feature of CCSL.[1]
- Verilog: A hardware description language that can be used to model everything from digital circuits to computer chip organization. Verilog is a

professional development tool used by many companies that produce electrical components; as a result, it is very well developed and quite extensive. Virtually every feature of CCSL is represented in this language.[3]

- LibLCS: A library for digital circuit simulation written in C++, like CCSL. This project implements many of the features of CCSL, but differes in a number of ways, especially in the way components interact.[2]

2 Project Design

2.1 Organization

The organization of the project is based on the concept of a breadboard into which a user plugs in components which they wish to use. This model allows for arbitrarily complex circuits, and is intuitive for those without an understanding of digital circuits. In addition, certain methods can be used by the implementation of the breadboard to make accessing and using components easier.

2.2 Evaluation Model

At the start of the project, I had a conceptual idea of how components and wires should interact. Components I have always thought of being something akin to functions, and wires bearing similarities with values. In this way, evaluating a circuit would conceptually produce a series function calls passing along different values by calling and returning from the components' associated functions. However, this is a narrow view of components and only really allows for combinatorial circuits. Naively implemented, this method would cause infinite loops when simulating sequential circuits.

Although narrow, this view of things is not without its advantages. In particular, it gives you a very good way of conceptualizing the interactions between components and wires that expands easily to sequential circuits; components are function-like things that perform operations on wires, which store the states of the circuit. This is the method for interaction between wires and components that I have chosen to adopt for this project.

The other main component of evaluation is the evaluation method. There are two main ways of approaching this problem. The first, and simpler approach, is to treat the components as vertices in a graph with the wires designating edges. This leads us to two main ways of evaluating the circuit, based on graph traversal techniques. Depth-first traversal is generally unhelpful for simulating circuits, so we use breadth-first traversal to evaluate the components. This gives us the first method for circuit traversal. The second method is based on the behavior of actual circuits.

In actual digital circuits, each component has a certain amount of delay that it introduces before it propagates its resulting values to the components that use these values. This delay is ignored by the depth-first traversal method, but

in order to fully simulate the behavior of real digital circuits, we cannot ignore this propagation delay. Our second method takes this delay into account and evaluates the components in an order equivalent to evaluating actual digital components. This is primarily the evaluation method of interest, since breadth-first traversal doesnt mirror the behavior of actual circuits in a vast number of cases.

2.3 Design Concepts

When designing the project, I employed several programming idioms and design patterns to simplify and generalize the layout of the project:

2.3.1 Iterator

To generalize the various ways in which a circuit can be evaluated or traversed, I made use of the iterator design pattern. This allowed me to scalably implement evaluators for both traversal patterns as well as other iterators with other purposes.

2.3.2 Observer

For simulating probes and detecting feedback loops, I used the observer design pattern. This pattern is not as heavily used as either of the other design concepts.

2.3.3 Handle-Body

In order to present a clean interface to users of the project, I used the handle-body idiom to generalize circuit components and allow the interface to be pointer-free, which increases its readability.

2.4 Potential Problems

2.4.1 Infinite Oscillation

When simulating circuits with feedback, there is the problem of looping. While some sequential circuits loop infinitely, others do not, and it is important to not disallow sequential circuits due to their pivotal importance to computers. In general, deciding if a particular circuit has a set of input states that create an infinite oscillation when evaluated is NP-complete. As a result, a heuristic is applied that works well in most cases, but is not general enough to detect loops in arbitrary situations. This heuristic is to keep track of how many times each wires state has been changed. If this number exceeds some pre-defined amount, we assume that the circuit has entered an infinite loop and immediately cease evaluation of the circuit. Implementation

3 Implementation

The next section details the projects implementation, including the implementation of various aspects of circuit simulation, organization, and interaction of components.

3.1 Library Structure

The structure of the source code is based on each modules primary use and the most related code. The major modules are:

3.1.1 BaseCircuits

Contains definitions for the built-in components. These built-in components include components for basic logic, Input/Output, Flip Flops and Latches, and simple Arithmetic.

3.1.2 Circuit

Contains the class files for the Circuit class, which implements a breadboard circuit. Components can be added to the circuit with the addition of an identifier, or string name. This allows them to be referenced by this identifier, eliminating the need to explicitly reference the component. The Circuit class makes use of a hash table to efficiently store these components by name.

3.1.3 CircuitEvaluator

Contains the definitions for the Evaluator classes. Evaluators for Debugging, Normal (BFS), Delay, and Circuit Refreshing (a helper evaluator for modules of the project) are included.

3.1.4 CircuitIterator

Contains the definitions for the two Iterators, BFSIterator and DelayCircuitIterator. The former is used to model Breadth First Iteration, and latter is used to simulate Component Delays.

3.1.5 Component

Contains the definitions for the Component classes. This is the most pivotal part of the project as most of the other classes are groundwork or wrappers for this class. Handle body is used to allow pointer-less interaction with Component class instances.

3.1.6 Exceptions

Contains the definitions for the Exceptions used in the project. This includes exceptions for Circuit Evaluation Errors, Component Errors, and Wire Errors.

3.1.7 Observer

Implementation of the Observer design pattern. This is used to implement probes which can be attached to circuits.

3.1.8 Wire

Defines the interface for wires. Primarily this is used internally, and users will nearly never interact with these directly.

3.1.9 gc

Implementation of the Conservative Garbage Collector, which is used to include garbage collection in the project.[6]

3.1.10 sparsehash

Implementation of the Sparse Hash Library, which is used to include hash tables in the project.[5]

3.2 Wire States

The wire states for the project are implemented as a pair of the timestamp the wire was last updated, and the state it was given at that update. This allows us to keep track not only of the states being passed along wires, but also the time delay introduced by components in the circuit. The primary class for this implementation is the State class, which includes the timestamp, state pair.

The timestamps passed around are implemented as quads (integers of 64 bits) to allow for very large timestamps for larger circuits that take much longer to run. In addition, the unit for these timestamps is defined as the amount of time it takes a negation to operate. I chose this as the unit since it is the smallest and simplest component that does some form of computation, unlike a splitter, for example, which simply passes its value along to its outputs.

3.3 Component Classes

3.3.1 Built-in Components

Built-in component definitions are used to implement certain elementary components for users to use.

For basic logic, binary `And`, `Or`, `Nor`, and `Xor` are included. Additionally, the elementary `Not` gate is included. Since wires are defined simply as an input/output pair, they do not allow a single input value to be passed to several possible components. For this purpose, the splitter has been included. It propagates a single value to a variable number of components.

Input and output to circuits and components is done with instances of the `CircuitInput` and `CircuitOutput` classes. These two classes provide a generalized interface for interacting circuits via their input and output components.

From these two generalized interfaces, more specific classes have been derived. For input, the components are:

- A `Button` allows boolean input from standard input via cin.
- The `StreamInput` device is like a `Button`, but reads from an arbitrary steam.
- A `Toggle` is a device that is toggled, rather than assigned a specific state.

For output, the predefined components are:

- An `LED` functions by printing a value to standard output via cout.
- A `StreamOutput` is like an LED but prints to any steam.

In addition to these components, simple components have been created to demonstrate other capabilities of the system. These include: a 1-bit `FullAdder`, which is included in the *Arithmetic* folder; input and output components for busses, `BitVectorInput` and `BitVectorOutput`, which are in the *BitVectors* folder; and an `SRlatch` and `JKflipflop` which are located in the *FlipFlop* folder.

3.3.2 Black-box Components

Beside pre-defined components, CCSL attempts to make it as simple as possible for users to define their own components and use them in larger, more complex circuits. To this purpose, the concept of black-box component has be included. This concept allows users to create components from circuits they have already built, which can then be used in other circuits. The class `BlackBox` implements these features. In order to use this class, a user need only define a class with whatever name they would like to give their component, inherit from `BlackBox`, provide the *Delay* function, and within the constructor create the circuit as they would within the main function of a circuit, using the result of the internal function as the top-level circuit. As an example of this, the pre-defined `JKflipflop` component has been created in this way.

3.4 Class Interaction

Within the project, classes interact in a variety of ways, depending on what this interaction is to accomplish. The goals for this interaction can be either attempting to add a component to the circuit breadboard, querying the state of a particular component, evaluating a circuit, and many other operations that can be performed on circuits. In addition, advanced to expert users can define their own such operations on circuits. For the built-in operations, the interaction is fairly simple and the structure of the library has been designed to make these operations easy to understand. What follows below is a description of the various interaction mechanisms for the project, with examples, where suitable.

3.4.1 Circuit Evaluation

Perhaps the most important mechanism for a single run of a project is the evaluation of the circuit. The interaction of the various components within a circuit is what gives the circuit its abstract meaning. For example, a SR-latch is defined in terms of the smaller components, the XOR gates and splitters that compose it, and the interaction of these components gives the circuit its particular behaviour. To fully simulate the interactions for circuit evaluation, the project includes the `CircuitIterator` class hierarchy. This class is the base class for all classes that implement evaluation of a circuit. An abstract base class is used to allow users to define their own evaluators that extend or alter pre-defined evaluation techniques. Three pre-defined circuit evaluators have been provided that allow different ways of modeling circuit interaction, `NormalEvaluator` which implements BFS evaluation, `DelayEvaluator` which implements delay-based evaluation, and `Debugger` which allows a user to interact with the circuit as a delay-based evaluation takes place. These evaluators are discussed in more detail below.

The interaction between the classes themselves is fairly simple. Upon a call to a circuits evaluate function, the circuit creates an evaluator that matches with the circuits evaluation method and then uses this evaluator to evaluate the circuit. An sequence diagram of this interaction is used below to detail class interactions. This diagrams shows the function calls and returns across classes with arrows. Each solid arrow represents a function call from one class to another. Dotted arrows represent function return from previous calls. In each diagram, there is an implicit user on the far left, making the left-most calls through either a main function, or within a black-box component.

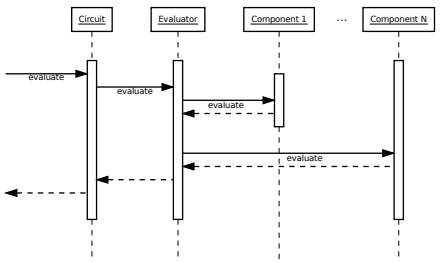

Figure 1: Interactions between the circuit, evaluator, and components when evaluating a circuit.

Circuit evaluation is done by some derived class which is based on the `CircuitIterator` class. This `CircuitIterator` ensures that the evaluators share a similar interface, but provides little other restrictions on their behavior.

As a result of this, evaluators can choose any order in which to evaluate the components. In addition, there is nothing disallowing components from being evaluated more than once, in fact this is guaranteed by some evaluation methods. In particular, all the built-in evaluators can evaluate components many times. Due to this lack of restriction, it is the responsibility of the evaluator to determine in which order the components are evaluated and keep track of them internally.

The two built-in evaluation methods are breadth first evaluation and delay-based evaluation. Both of these methods are provided as a way of simulating real circuits, although each has their own advantages and disadvantages. Of the two, delay-based evaluation comes closer to the behavior of actual circuits, but breadth first evaluation has the advantage of being easy to understand from the point of view of the component interactions, as well as the implementation, since a plain queue can be used, rather than a priority queue.

There is a third implemented evaluator that is based on a modified version of delay-based evaluation called the `Debugger`. This evaluator evaluates the circuit using delay-based evaluation, but stops at each component to print output about what the components are doing, and prompt the user for input. This allows the user to see what is happening and interact with the circuits evaluation, as it is being evaluated.

For any given circuit, the user can select which of these two evaluation methods are to be used by calling either of the functions *UseGateDelays* or *IgnoreGateDelays*, which enable or disable using gate delays, respectively. Alternatively, using a `Debugger` to evaluate the circuit can be done by calling a circuits *Debug* function.

Regardless of which type of evaluation the user selects, the evaluator needs to know which components indicate the inputs, or beginning of the evaluation. This is done with the *AddInput* function, which allows the user to indicate which components should start the circuits evaluation.

3.4.2 Circuit Creation

Circuit creation is also done via the `Circuit` class. This allows `Circuit` to be a single interface for all the necessaries of component interaction.

The diagram below demonstrates the interactions between the user (far left), `Circuit`, and hash table. Before a `Circuit` can be used, it must be created. This can be done by the use of its default constructor indicated by section **A** in the figure below. No additional work is needed to create the circuit, as the constructor does all the necessary setup. For creating a component, the `Circuit` class provides the *AddComponent* function, which adds a component to the Circuit with a particular name; represented by section **B** on the figure below. The *AddComponent* function requires a `Component`, passed by pointer, and a string for the name. The `Component` is generally passed by creating a new Component, since it is garbage collected. The string has no restrictions except that each components name must be unique in a given `Circuit`. For accessing that component later, the function *Lookup* is provided, which looks a

component up by name; represented by section **C** in the figure.

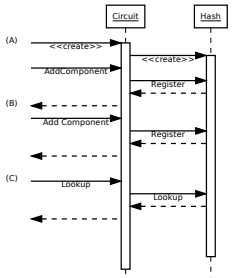

Figure 2: Interactions between a `Circuit` and hash table when creating components.

3.4.3 Circuit Connection

Being able to create and evaluate circuits is important, but means nothing without the ability to connect components. This allows the user to specify the way in which the components interact. Like component creation and evaluation, connection is done with the `Circuit` class. The class provides the member function *Connect*, which takes four parameters: the input component, the input pin number, the output component and the output pin number. This function creates a `Wire` and associates the wires input with the specified pin number of the input component, and the Wires output with the specified pin number of the output component. The diagram below demonstrates the interaction between a `Circuit` and the newly created `Wire`.

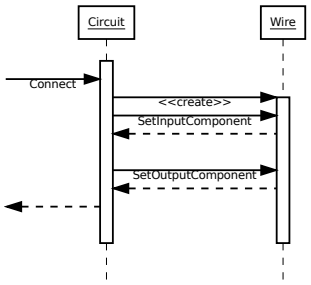

Figure 3: Interactions between a `Circuit` and `Wire` when connecting components.

4 Future Work

There were a few tasks that the project sought to accomplish, but were left incomplete. The first of these tasks is a proof that detecting circuit oscillation is NP-complete. This proof would be a proof by reduction, as proofs of NP-completeness generally are. The outline of the proof is that detecting oscillation would require keeping track of every possible state in which the circuit has been. While finite, the number of possible states is likely to be exponential in worst-case, so such an algorithm would be at best exponential in space, which implies exponential in time.

In addition, the concepts of circuit iterators and circuit evaluators arent necessarily distinct at this time. If given more time, the two concepts could be made into disjoint but related ideas and that difference could be made more clear in code. In particular, iterators should be objects that iterate over components in a circuit without necessarily the goal of evaluating them, e.g. an iterator could be used to search over all components to find a particular component by name. On the other hand, evaluators would be designed with the specific purpose of evaluating a circuit in mind. This would lead them to take either the delay-based or breadth-first approaches, as discussed above. For example, in addition to the three existing evaluators, one could be added that computes the delays for circuits, eliminating the need to define the *Delay* function in custom built components.

Finally, although busses were thought of as a desirable feature of the project, they were not a focus of the design. As a result, they can be made to work with for loops and similar techniques, but the approach is fairly ad-hoc and doesnt fit with the interface for most other interactions to the library. In the future, busses could be made to work by generalizing the concept of wires to wire groups or busses. This would allow users to link wire groups between components, rather than single wires. Changing the interaction in this way would still allow users to

use single wires - they would merely be interacting with unary wire groups - but allows simple interfaces for busses. Once this change had been made, it would be fairly easy to allow components to name ports, allowing users to reference them by name, rather than by an integer reference. This would greatly simplify something like an SR-latch, where each input and output has a specific name, but their ordering has been assigned arbitrarily.

References

[1] Klogic: Digital circuit simulation. http://www.a-rostin.de/.

[2] libLCS: A Logic Circuit Simulation Library in C++. http://liblcs.sourceforge.net/.

[3] IEEE 1364-2005. IEEE Standard for Verilog Hardware Description Language. *Institute of Electrical and Electronics Engineers*, April 2006.

[4] Carl Burch. Logisim: A Graphical Tool for Designing and Simulating Logic Circuits. http://ozark.hendrix.edu/ burch/logisim/index.html.

[5] Craig Silverstein and Mark Weiser. sparsehash: An extremely memory-efficient hash_map implementation. http://code.google.com/p/sparsehash/.

[6] Hans J. Boehm and Alan Demers and Mark Weiser. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[7] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2000.