Digital Circuits

Andras Botar

Programming project report

May 2022

Abstract

A C++ program was written to represent and simulate digital circuits. It contains classes for digital components, and circuits made from these. It is able to create and simulate circuits made of many components, connected in an arbitrarily complex way. It provides utilities to generate truth tables for components, turning circuits into components or lambda functions, drawing a circuit diagram, and various other functionality. It provides a text based, menu interface and tutorial for users, however it may also be used as an imported library.

1 Introduction

Digital electronics and computers form the foundation of the modern information economy. With the decreasing size and cost of digital components, the complexity of digital circuits and the challenge of designing them has exploded. One clear application of computers, is to the design of digital electronics themselves.
Modern Hardware design packages can represent a large variety of component types, can perform complex optimizations and simulations. This program is beginning approximation for performing these functions. Digital circuits are usually constructed from silicon transistors, however here we are interested in higher level, logic-gate representations. Where one component acts as a Boolean function on the outputs of other Boolean functions. Here logic gates are defined as digital components with only one boolean output. These components are then collected into a single circuit with complex time dependent behaviour.
We wish to create classes to represent such circuits and components, with the ability to calculate their outputs. This is done in discrete time-steps, by calling an update function belonging to each component in a circuit. Physical components have propagation delay, and race conditions might occur depending on these, however here a uniform "clock signal" is implicitly assumed. The model used is equivalent to a "Flow-based programming" approach.

2 Code design and implementation

Two primary classes are defined: One is a component class, representing individual AND gates, XNOR gates, NOT gates..., and the other is a circuit class, representing the combination of these components. The class hierarchy of the program is illustrated in Figure 1.
A variety of logic gates are implemented, such as AND, OR, XOR gate and the inversions of these. NOT gates, buffers and constant inputs. All of these are generalized to arbitrary number of inputs. Majority function is also implemented: a function which return false if half or more of its inputs are false,

2.1 Core classes

The state of a circuit is stored as a std vector of Booleans. Where each boolean represents a single "wire", that is the output of a component, which may by the input of one or more other components. The use of bit-sets was considered, however they did not provide sufficient advantages. It is notable that the std::vector¡bool¿container actually behaves significantly differently than other std::vector specializations. This is due to the container sorting each Bool not in its own byte, but as parts of a single large word in memory which cannot be independently accessed. Instead of returning a bool, the container returns a proxy object which can access the bool indirectly.
The components that compromise a circuit are stored as a vector of base-class-pointers which allows the use of polymorphic component classes without slicing i.e. loss of non-inherited func-

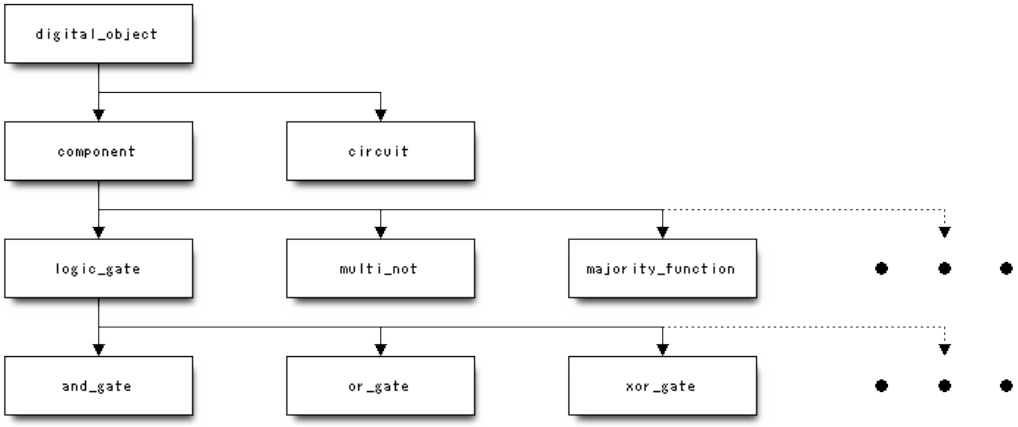

Figure 1: Hierarchy of classes in program.

tions. Specifically each circuit class has an std::vector of unique-pointers, pointing to components or objects derived from the component class. The use of smart pointers assures the deletion of pointers when they go out of scope, helping avoid memory leaks. Since unique pointers cannot be copied to multiple owners, care must be taken to use std::move when passing them around. The behaviour of unique pointers and the non-standard behaviour of vector¡bool¿ makes the use of certain features, such as iterators, difficult.

2.2 Updating

At every time-step, the circuit steps through its list of components calling their update functions. These functions take by reference the current state vector of the circuit and a similar vector representing the new state of the circuit. Which wires of the state vector the update functions read, is determined by a list of indices stored inside each component. Similarly, they output to a certain element of the new state vector based on an internal output index. These indices are set when the components are initialized and represent the topology of the circuit.
Once every component has updated, the old state vector is swapped with the new state vector. This is equivalent to copying the new state into the old state, however this involves only a pointer reassignment, making it an efficient O(1) operation.
Since circuits manage pointers a destructor, copy constructor and copy assignment operator are defined for it. These use the swap and copy idiom, for exception safety.

2.3 Utilities

The circuit class contains 40 different functions for a variety of goals, including labelling, viewing, replacing, changing , deleting or adding the individual components or wires in the circuit; printing information; resting state, etc.

Several utility functions are also defined outside this class, to e.g. display the state of a circuit or to print the truth table for a logic gate. Here templated functions are used to print vectors, find whether an element is in a vector, and to display tables for several different types. To generate truth tables, the program simulates circuits until either they no longer change or a maximum number of evaluations is reached. If the circuit has no feedback, it eventually reaches "steady state", depending on the depth of the circuit. But if feedback is present, oscillatory behaviour may occur, which may cause unpredictable or unexpected results.
The circuit class also has function to return a lambda function representing one given output of the circuit, given its inputs. A component class is provided to convert this or any other boolean lambda function into a valid logic gate component. This allows the user to combine an arbitrary number of components from an arbitrary number of inputs, the package up this whole collection seamlessly as if it were a monolithic logic gate. This also allows for seamless sub-circuits, and for recursively defining some circuits as made from logic gates which are made from other circuits and so on.

2.4 User interface

The program can be imported and used as a flexible library of classes. However and additional command line interface is also implemented. This takes the form of a menu with a series of menus and options. At point a list of possible choices is presented to the user, and they can navigate this menu tree by entering the number corresponding to their choice.
The menu tree is implemented as a state machine , with each sub-menu represented by an integer value for a global state variable. A single circuit is maintained and manipulated by all UI functions.
The root of the menu tree is a "main menu" with the following options: See tutorial, Load existing circuit, Create new circuit, Settings, Help, Exit.
The tutorial option is an interactive subprogram meant for gradually introducing the user to the functions of the program and demonstrating its capability. The create new circuit allows for the creation of new circuits from a blank-empty circuit. Settings include whether to draw frames or indices for truth tables, mathematical symbol selection, what to draw in circuit diagrams, etc. Help displays an attached readme.txt file with some basic information. And "Exit" quits the program.
The above mentioned circuit diagrams refer to the ability of the program to create a basic circuit diagram of a circuit object. It uses the "BlockDiag" program to do this, which may be installed using the "sudo dnf isntall blockdiag" command. It can creates a block diagram with directed edges, labelling each node as a gate. Optionally labelling may be turned on or off and logic gate symbols may be shown on the nodes. For efficient space usage, sometimes wires going to the same logic gate may be collapsed on the diagrams. An example diagram is shown in Figure 2.
To use unicode box-drawing characters, the std::wstring functionality was used instead of std::string in the entirety of the program.

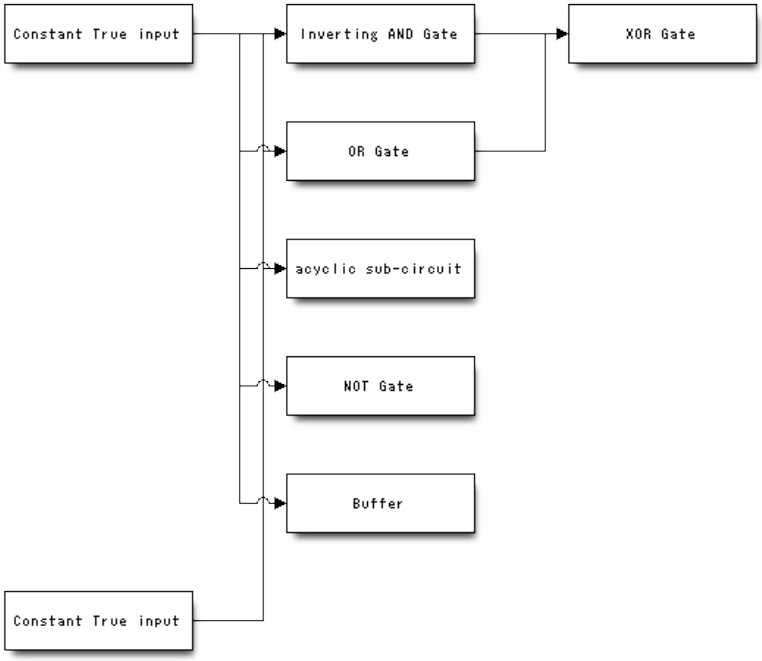

Figure 2: Example circuit diagram, with labels and no symbols based on settings.

3 Results and conclusion

The program is able to perform the stated goals of simulating, combining and analyzing circuits and logic components. It is able to link any number of components of differing types, in serial or parallel. The program functions for cyclic and acyclic circuits, i.e. circuits with and without feedback. And it has dedicated functions for detecting if it has feedback, and if any sub-circuits in it have feedback. It provides a user friendly menu for CLI users and organized classed for library use . It is able to create truth tables, circuit diagrams and functions representing circuits. And it uses a variety of advanced C++ features such as templates, lambda functions and smart pointers.

4 Discussion

4.1 Further work

There are a large number of avenues for further development, some of these include: collecting truth table data and functions into single truth table class; designing less cumbersome interfaces for adding components in serial and parallel; and simply adding more component implementations.

Also exporting and importing circuits as verilog modules or other file formats, this would make

it easier to store example and user defined circuits but also allow for interfacing with the large ecosystem of existing tools;
For drawing, the edges could be colored based on the state of the circuit, and with a series of such images an "animation" could be created of how the circuit propagates information. A new, blockdiag plugin could be created to change shapes of boxes to that of logic gate symbols or to change routing behaviour. Or an entirely new, more diagramming approach would likely be the most fruitful.

4.2 Mathematical analysis

What here is called a "wire-state" is called an atom in boolean logic, representing a single "variable" that can be true or false. And combinations of these atoms with logic gates is a boolean formula. A formula is called a Closed-Normal-Form(CNF) formula if it consists of a conjunction (AND) of one or more clauses, each of which is a discjunction (OR) of atoms or their negations; n AND-of-OR-s. The opposite, an OR-of-AND-s is called Disjunctive-Normal-Form(DNF). Every boolean formula can be converted into equivalent CNF and DNF formulas. Notice, that if one collects all the elements from a truth table where a logic gate is on (the minterms), one just and together terms in one column of the truth table, and or these conjunctions together to create the DNF representation for any circuit. This may be interesting because boolean simplification can rapidly be done on DNF forms, such as with the Quine–McCluskey algorithm. Note also that constructing a circuit object from a DNF description is not difficult. So if the above mentioned steps were to be implemented, a powerful circuit simplification functionality cold be added. Of course there are eve more efective methods, such as extending the qm algortihm with Petrick's method, using And-Invert-graphs for deeper circuits or by using other algorithms such as the espresso heuristic minimizer.
Another possibly useful aspect is in terms of boolean satisfiability, meaning is there some combination of input which produces a true output. This was the first problem to be proven NP-complete, and plays an important role in computational theory. Calculating the truth table of a circuit and checking if one of the elemenets is true, is the brute force method, whose resource requirements scale extremely rapidly with circuit size. So functionality to directly translate circuit objects into boolean expression and then DNF or CNF form wold provide a wealth of new options. A DNF form of a circuit can be transformed into CNF form using a Tseytin transformation. And a CNF form specifically is useful, becuase many boolean satisfiability solvers work with CNF form primarily. Or the inverse, allowing a user to create a circuit from a boolean expression or a truth table might be useful, and is an active are of research [1].

References

[1] Z. Fu and S. Malik, "Extracting logic circuit structure from conjunctive normal form descriptions," in *20th International Conference on VLSI Design held jointly with 6th International*

Conference on Embedded Systems (VLSID'07), pp. 37–42, 2007.