Logisimple

A Combinational Hardware Description Language

Yuanxia Lee (yl3262): yl3262@columbia.edu
Sarah Walker (scw2140): scw2140@columbia.edu
Kundan Guha (kg2632): kundan.guha@columbia.edu
Hannah Pierce-Hoffman (hrp2112): hrp2112@columbia.edu

Introduction

Most digital circuits contain some type of combinational logic, in which a series of gates operate to perform Boolean algebra on one or more inputs. Together with sequential logic, combinational logic constitutes a fundamental pillar of hardware design. The design and simulation of combinational circuits can rapidly become very time-consuming as the size of the circuit increases. Common means of simulating combinational circuits tend toward the highly visual (such as Logicly or Logisim) or the highly abstract (such as Boolean algebra solvers). Logisimple aims to provide a middle ground by allowing the user to create, connect, manipulate and test combinational hardware via lines of code. This language is fundamentally object-oriented in its concepts of gates and circuits, but incorporates features of mutability and automatic calculation intended to facilitate the process of design.

Intended Uses

- Model and test combinational logic without using actual hardware
- Educational purposes
- Produce truth tables for combinational circuits

Feature Overview

Types

- *Primitives*
    - *Boolean*
        - Basic boolean type, either 0 or 1
    - *Boolean[]*

        - Array with a maximum size of 8
        - Elements can be:
            - a) booleans
            - b) other boolean arrays
            - c) gates (which resolve to booleans during parsing)

- *Static Classes*
    - These are essentially "fancy operators" that take inputs and process them; we conceptualize them as classes to make them easily manipulatable
    - *Gate*
        - Can be constructed of primitive gate types AND, OR, and NOT (all other logic gates can be implemented in Standard Library through these primitives)
        - Attributes: inputs and outputs (both arrays)
        - Another attribute: the actual type of gate it is
        - When outputs are accessed, gate does internal (private) operation (AND, OR, or NOT) on inputs to produce output
        - A gate can be a collection of nested gates
    - *Circuit*
        - Wrapped gate system that is called a circuit for differentiation

Gate Syntax: User Representation

- Users declare Gate name followed by curly brackets. Inside curly brackets, they should define inputs, outputs, primitives, and operations.
    - If input is not specified, it will default to 0
- Arrows (->) connects a gate's outputs to the input of another gate
    - If the second gate already has inputs, arrow overrides existing inputs
- Indices of gates access outputs
- Dots of gates access inputs

Gate Syntax: Internal Representation

- S-expressions will be used internally for nested parsing
    - All arrays are of size 2
    - Example internal representations of gates:
        - AND[1,0]
            - This resolves to 0
        - OR[AND[0,0], 1]
            - This resolves to 1
- Operators (internal only, not available for user)
    - *AND* becomes *&* during parsing
    - *OR* becomes *|*
    - *!* becomes -

Simulation

- Users will be able to call a function that outputs the truth table for any circuit or gate system (including a set of gates within a larger system)
- The purpose of each gate's automatic internal calculation is to incorporate some ongoing simulation into the basis of the language
- A set of "get" methods allows the user to place a "probe" at any place within the imagined gate system and capture the boolean value at that location

Optimization

- Depending on our progress, we may implement a feature in our language that performs optimizations via circuit minimization (e.g., the sort of minimization that might be done with a Karnaugh map).
- A user would be able to call an optimization method or methods to automatically generate the minimized version of a given circuit or gate system.

Mutability

- Inputs can be determined in the command line for variability
- Can wrap a system of gates into a "circuit" for ease of use with functions
- swap(x,y) function which changes the places of two gates or circuits relative to each other
- remove(x) function that removes either a circuit or a gate and takes the inputs to the circuit or gate and directly sends those inputs to the gate the original gate outputted to

Syntax Table

Syntax	**Explanation**
a -> b.in1, b.out -> c, ...	Connects inputs and outputs (like a wire)
a[i] -> b[j]	Indexes pin-outs to pin-ins
;	Ends lines
Gate x {}	Define new Gate type x
$	Single line comment
$$	Multiple line comment
geti(x), geto(y), geta(z)	Gets x's input array, gets y's output array, gets both arrays of z

gett(a)	Get truth table of gate system a
Inputs: x, y, ...	Declares variable names corresponding to pin ins. Each pin-in also corresponds to an array position in order of declaration.
Outputs: x, y, ...	Like Inputs: but for pin outs.

Sample Program

```
$ Implementing a Half-Adder in Logisimple
Outputs: res, carry $ The “pin-outs” of the circuit

$ These gates could be implemented in SDL
Gate NAND {
  Inputs: In1, In2; $ Maps an array of inputs to variables
  Outputs: out;
  AND a, NOT n;
  $$
   This is a multi-line comment
   The code below shows multiple connections in
   one statement. Also, AND and NOT both only
   have one output, so their output can be explicitly
   declared, as for a.out, or implicitly, as in n->out
  $$
  In1 -> a.in1, in2 -> a.in2, a.out -> n.in, n -> out;
}

Gate XOR {
  Inputs: In1, In2;
  Outputs: out;
  AND a, NAND na;
  In1 -> na.in1, In2 -> na.in2, In1 -> o.in1, In2 -> o.in2;
  na -> a.in1, o -> a.in2, a -> out;
}

AND a, XOR x, bool[2] b = [1,0];
b[0] -> a.in1, b[1] -> a.in2, b -> x, a -> carry, x -> res;
$ b -> x == b[0] -> x.In1, b[1] -> x.In2
$ Will output [1,0] (res is 1 and carry is 0)
```