Logisimple

Language Reference Manual

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

October 16, 2017

Contents

1 Introduction

Logisimple aims to provide a streamlined, text-based format for simulating digital circuits. With a structure based on nesting gates, users can combine sequential and combinational building blocks to quickly model complex hardware. The completed circuit can be treated as a "black box" during simulation, allowing users to manipulate inputs and view outputs for the entire system. Some of our most important goals in designing this language have been the following:

- Concise syntax
- Small number of types and operators
- Generally object-oriented concept of gates
- Intended for skilled users with knowledge of digital hardware

2 Lexical Elements

2.1 Identifiers

2.1.1 Gate names

Both user-defined and built-in gate names are all uppercase letters. Examples:

```
AND x.in = [0,0];
## Declaring and assigning a new AND gate will always require AND in uppercase letters.
↪  The same applies for OR. ##

MYNEWGATE{
    # gate definition here
}
## When defining a new gate type, always use uppercase names ##
```

2.1.2 Variables

Boolean literals, boolean arrays, and instances of gates may be assigned to variables. Variables must begin with a lowercase alphabetical character but may contain alphabetical characters of any case. Examples:

```
bool myVariableName = 1;
bool[] myArrayVariable = [0,0];
AND myGateVariable.in = [0,1];
```

Variables may not be declared without assignment, except in the case of the header.

2.2 Keywords

2.2.1 in

`in` is a keyword used to define the boolean input arguments to a user-defined or built-in gate.

`in` can be thought of as an identifier for the array of input arguments which is unique to each instance of a gate.

When assigning inputs, `in` is used with the dot syntax on the left side. The right side of the assignment is a boolean array of inputs to the gate:

```
AND example.in = [1,1];
```

Input values can also be accessed by index in the `in` array. The following example declares a new boolean `takeValueOfInput2` and assigns to it the second input argument to the `EXAMPLEINPUT` gate:

```
    EXAMPLEINPUT{
            # define this gate
    }
    EXAMPLEINPUT x.in = [0,0,0];
    bool takeValueOfInput2 = x.in[1];
    # takeValueOfInput2 is now equal to 0
```

2.2.2 out

`out` is a keyword which is used to define the boolean output array of a user-defined gate. Example:

```
MYGATE{
        out = [!in[0]]; ## evaluates to negated value of first input ##
}
```

Example when the output array contains more than one element:

```
MYOTHERGATE{
        out = [1, 0, 1];
}
```

Similarly to the input values, output values can also be accessed by index in the `out` array.
To assign the first element of the output array of an instance of `MYOTHERGATE` to variable `y`:

```
MYOTHERGATE x.in = []; #takes no inputs
bool y = x.out[0];
```

2.2.3 include

`include` is a keyword that is used to define external files to be included in the program. This facilitates the reuse of previously written code.

2.3 Operators

2.3.1 Dot operator

As discussed in Section 2.2, the dot operator is used only in conjunction with the `in` and `out` keywords. When accessing the inputs or outputs of a specific gate, the syntax `gatename.in` or `gatename.out` is used.

2.3.2 Indexing

The bracket operator `[]` is used to index boolean arrays, which represent the inputs or outputs of gates.

```
MYGATE m.in = [0,1,1,0,1];
bool tmp = m.in[2];
# tmp evalutates to 1
```

2.3.3 Assignment

Our syntax supports the use of the `=` operator to set the inputs of a gate or perform another assignment to a variable name. We follow the standard convention for use of this operator: the value of the right-side operand is stored in the left-side operand.

2.3.4 Negation

The exclamation point is considered as a negation operator, and thus when placed directly before a Boolean, provides its complement.

```
bool a = 1;
bool b = !a;
#b now evaluates to 0
```

2.3.5 Operator precedence

In Logisimple, if there are multiple operators in a line, they are evaluated based on precedence. The operators listed below are in order of highest precedence first and evaluate left-to-right (left associative) except for the assignment operator which is evaluated right-to-left (right associative).

```
1. dot operator .
2. indexing []
3. negation !
4. assignment =
```

An example piece of code with an explanation of its evaluation:

```
bool x = 0
AND y.in[0] = !x;
## The dot operator first accesses the in Boolean array of the AND gate y, and
then accesses the first index of that array. After, the Boolean x is negated,
then assigned to the accessed first index of y.in. ##
```

2.4 Separators

2.4.1 Commas

The comma is used to separate identifiers. For instance, if multiple gate names are declared at the same time, they must be separated by commas.

```
MYFLIPFLOP a, b, c;
```

2.4.2 Semicolons

Semicolons are used as statement terminators. Any line that is not a bracket or a header must be terminated with a semicolon.

```
MYINVERTER {        # no semicolon needed here
    out = [!in];    # semicolon needed here
}
```

2.4.3 Comments

The pound symbol is used to denote comments.

```
# Single line comments begin with a single pound symbol.
## Multi line comments are enclosed by two pound symbols
on either side. ##
```

2.4.4 Brackets

Logisimple employs two types of brackets: `[]` (square brackets), and `{}` (curly brackets). Square brackets are used to enclose boolean arrays. User-defined gate definitions use curly brackets to delimit the beginning and end of the definition. Additionally, the entirety of each Logisimple program is surrounded by a pair of curly brackets.

```
  {
       $$ # header

       # some source code

       bool[] myarr = [1,0]; # boolen array is enclosed in square brackets
   }
```

Brackets also control the scope of identifiers. See Section 4 for details.

2.4.5 Whitespace

All tokens must be separated by whitespace.

3 Data types

3.1 Booleans

Booleans are defined as `bool id = e`, where `e` can be either 1 or 0, representing the usual binary notion of the symbols. `e` can also be any valid expression that evaluates to a boolean value– for instance, `e` could be another boolean variable, or the input or output of a gate.

3.2 Boolean arrays

Arrays of boolean values are declared as `bool[] id = [e1, e2, ...]`, where `id` is any identifier for the variable and `e1` are any valid expressions evaluating to booleans. Boolean arrays may also be used anonymously through the square bracket declaration. See below for examples.

```
# Example 1:
bool a = 1;
bool[] one = [a,0];

# Example 2:
AND x.in = [1,0];
bool[] two = [x.in[0], 1];
```

3.3 User-defined combinational logic gates

Gates are the foundational data type, out of which all simulation elements are constructed. Gates can be very large and complex, comprising an entire circuit with many subgates, or very small, performing a single basic operation. The two essential components of a user-defined gate definition are the *output*, which consists of one or more outgoing pins with boolean values, and the *body*, which describes how the output is calculated. The input values are set when instances of a user-defined gate are actually declared. Input values may **not** be reassigned within the definition of the gate. Users may refer to the input in the gate definition by indexing the `<gatename>.in` array. The body of a gate may contain subgates that operate on the gate input; these subgates may be previously defined or defined within the body of the supergate. We provide AND and OR primitive gate types, as well as other common gate types in our standard library.

3.3.1 Output

The last line of a user-defined gate definition must be the output. The output line specifies one or more outputs, which must be enclosed within square brackets since they are interpreted as a boolean array. **This line is mandatory in user-defined gates.** User-defined gates may output constant values, boolean variables, or a combination of both.

```
MYGATE {
    # gate definition begins here
    # the variable x is defined somewhere
    out = [!x];
}
```

3.3.2 Body

Between the input and the output lines of a user-defined gate definition, the body section allows the user to create variables, declare AND and OR primitive gates, and define subgates. The user can also reference any gates or variables defined one scope level above the gate currently being defined (i.e., when defining

a gate within one set of brackets, the definition may reference any global gates or variables). For simple user-defined gates, it is possible that the gate definition body may be empty.

```
# User-defined NOR gate
{
$a,b$  # header
    MYNOR{
        OR x.in = [a,b];  # body
        out = [!x];               # output
    }
}
## Note that even though this is a single-output gate, we still must enclose
the output value in square brackets. ##
```

3.4 Primitive combinational logic gates

3.4.1 AND

AND is built in to Logisimple. An AND gate can be instantiated as follows:

```
AND mygate.in = [1,1];
```

The above line instantiates an AND gate called mygate, and gives inputs 1 and 1 to the AND gate. Calling `mygate.out[0]` would give the result of the logical AND of 1,1 which equals 1.

3.4.2 OR

Similarly to AND, OR is a primitive in Logisimple. It is instantiated the same way as an AND gate, except with the word `OR` instead.

3.5 User-defined sequential logic gates

3.5.1 Flip Flops

While in reality flip flops can be represented as a combination of logic gates, in Logisimple flip flops are represented through a base primitive type called `DFF` (D Flip Flop) and a global clock. All other flip flop types may be constructed from a combination of D Flip Flops and logic gates.

`DFF` models a normal one input, one output D Flip Flop where the input updates the stored value, and the output is the currently stored value. All D Flip Flops initially store 0. Flip flops operate on a shared global clock. Within each clock step they simply output based on the last clock step's inputs. Any given inputs update the stored value for the next step. For example,

```
DFF d1.in = [1]; # Currently stores 0, next step will store 1
AND a.in = [DFF.out[0], 1]; #On first cycle will output 0, second cycle 1
```

3.5.2 Clock

All programs operate on a global clock driving all flip flops at once. See 3.5.1.

3.6 Gate decorators

Gate decorators allow for the creation of more general gate abstractions. They are, broadly speaking, functions that take gates as inputs and output gates. This allows for generic changes to any input gate, i.e. putting together four gates to make a four bit version of the gate, etc. A decorator may be named any name containing one capital letter followed by an arbitrary number of lower case letters, and then a parenthetical, comma separated list of IDs, corresponding to the expected gate inputs, with the body surronded in curly braces. Essentially, C-like syntax. Decorators are evaluated at compile time and expanded into the resulting gate expression, akin to a C or LISP macro.

```
Makegated(g) {
    AND a1.in = [in[0], in[2]];
    AND a2.in = [in[1], in[2]];
    g.in = [a1.out[0], a2.out[1]];
    out = g.out;
}
GATEDXOR = Makegated(XOR);
```

Would expand to

```
GATEDXOR {
    AND a1.in = [in[0], in[2]];
    AND a2.in = [in[1], in[2]];
    XOR x.in = [a1.out[0], a2.out[1]];
    out = x.out;
}
```

4 Scoping

An identifier's scope is the curly brackets which enclose it, and any brackets those brackets enclose, for any level of depth. Parameters which become populated via command line arguments are defined within the entire source program because they are only enclosed in the outermost brackets.

Example:

```
{
$a,b,c$
    AND x.in = [a,b];
    MYCIRCUIT{
        OR y = [a,c];
        out = [!y.out[0]];
    }
}
```

In the above example, a, b, c and x are declared throughout the entire source program. y is only defined within MYCIRCUIT. Note that a, b, and c are defined within their enclosing brackets, and the brackets those brackets enclose (i.e. the brackets for MYCIRCUIT).

5 Standard Library

The Standard Library provides commonly used gates in addition to the existing AND and OR primitives.

5.1 NAND

```
NAND {
    AND x.in = in;
    out = [!x];
}
```

5.2 NOR

```
NOR {
    OR y.in = in;
    out = [!y];
}
```

5.3 XOR

```
XOR {
    OR o.in = in;
    NAND na.in = in;
    AND a.in = [o.out[0], na.out[0]];
    out = a.out;
}
```

6 Program structure

6.1 File format and extension

All source programs must be enclosed in curly braces to denote the start and end of the source code. The file containing the source code requires file extension `.sim`.

6.2 Executable file

The compiler builds an executable file which simulates the behavior of the circuit. This executable file is `<filename>.simx`. The executable file must be run with command line arguments which serve as the boolean inputs to the program. The source code requires a header which specifies the identifiers of the input arguments necessary to run the simulation. The header is specified between two dollar signs, and may be empty.

```
{
    $x,y,z$
    # some code here
}
```

If `example.sim` has header `$x,y$`, then `example.simx` is run from the command line as follows:

```
./example.simx 1 0
```

In the example case, x = 1 and y = 0 are assigned at compile time.

6.3 Using the standard library

The `include` keyword makes it simple for the user to access the standard library. Including an external file containing the standard library code effectively copies and pastes the standard library code into the current file, similar to the `include` keyword in Python.

7 Sample program

```
# Implementing a Half-Adder in Logisimple
{
$in1, in2$

# These gates could be implemented in the standard library

NAND {
    AND a.in = in;
    out = [!a.out[0]];
}

XOR {
    OR o.in = in;
    NAND na.in = in;
    AND a.in = [o.out[0], na.out[0]];
    out = a.out;
}

AND a.in = [in1, in2];
XOR x.in = [in1, in2];

bool carry = a.out[0];
bool result = x.out[0];

out = [result, carry];
}
```

8 Grammar

8.1 Production Rules

Below are the production rules for deriving Logisimple programs. This context-free grammar is a work in progress in that it does not yet support sequential logic. Additionally, some nonterminals such as "user defined type" are placeholders that have not yet been defined in more detail.

r → { s }

s → HEADER BODY

BODY → | EXPR; | DEFINEOUT; | BODY ; BODY | ϵ;

HEADER → n;

EXPR → NEWGATE | USERDEFBODY | DEFBOOLS

USERDEFBODY → Id{ EXPR ; DEFINEOUT;}

NEWGATE → Type Id.in = [i,i]

DEFINEOUT → out = [OPTIONALNEG Id]

OPTIONALNEG → ! | ϵ

DEFBOOLS → bool Id = OPTIONALNEG IDACCESS | bool Id = OPTIONALNEG i | bool[] Id = [n]

Type → OR | AND | (* user defined type *)

b → 1|0

IDACCESS → Id.in[i] | Id.out[i]

i → in[INT] | b | Id

INT → (*integer*)

Id → (*identifier*)

n → m | ϵ

m → Id | Id, m

8.2 Example derivations

In each of the examples below, we provide a short Logisimple program, and then show the steps for a rightmost derivation of that program.

```
# Example 1
{
$a,b$
NOR{
    OR x.in = [in[0],in[1]];
    out = [!x];
}
NOR mynor.in = [a,b];
}
```

r
s
HEADER USERDEFBODY
HEADER Id NEWGATE ; out = [OPTIONALNEG Id] ;
HEADER Id NEWGATE ; out = [!x] ;
HEADER Id Type Id.in = [Id,Id] ; out = [!x] ;
HEADER Id OR x.in = [in[0],in[1]] ; out = [!x] ;
HEADER Id OR x.in = [in[0],in[1]] ; out = [!x] ;
HEADER Id OR x.in = [in[0],in[1]] ; out = [!x] ;
HEADER Id OR x.in = [in[0],in[1]] ; out = [!x] ;
n Id OR x.in = [in[0],in[1]] ; out = [!x] ;
a,b; Id OR x.in = [in[0],in[1]] ; out = [!x] ;

```
# Example 2
{
$$
    AND myand.in = [1,1];

    bool y = myand.out[0];

}
```

r
s
HEADER BODY
HEADER BODY ; DEFBOOLS;
HEADER BODY ; bool Id = OPTIONALNEG IDACCESS;
HEADER BODY ; bool Id = OPTIONALNEG Id.out[i];
HEADER BODY ; bool Id = OPTIONALNEG Id.out[0];
HEADER BODY ; bool Id = OPTIONALNEG myand.out[0];
HEADER BODY ; bool Id = myand.out[0];
HEADER BODY ; bool y = myand.out[0];
HEADER NEWGATE ; bool y = myand.out[0];
HEADER Type Id.in = [i,i] ; bool y = myand.out[0];
HEADER Type myand.in = [i,i] ; bool y = myand.out[0];
HEADER AND myand.in = [i,i] ; bool y = myand.out[0];
n; AND myand.in = [i,i] ; bool y = myand.out[0];
$ $; AND myand.in = [i,i] ; bool y = myand.out[0];

```
# Example 3
{
$a,b,c$
    AND x.in = [a,b];
    MYCIRCUIT{
        OR y = [a,c];
        out = !y;
    }
}
```

r
s
HEADER BODY
HEADER BODY ; BODY
HEADER BODY ; USERDEFBODY
HEADER BODY ; Id EXPR ; DEFINEOUT;
HEADER BODY ; Id NEWGATE ; DEFINEOUT;
HEADER BODY ; Id NEWGATE ; out = [OPTIONALNEG Id];
HEADER BODY ; Id NEWGATE ; out = [OPTIONALNEG y];
HEADER BODY ; Id NEWGATE ; out = [!y];
HEADER BODY ; Id Type Id.in = [i,i] ; out = [!y];
HEADER BODY ; Id Type Id.in = [a,c] ; out = [!y];
HEADER BODY ; Id OR y.in = [a,c] ; out = [!y];
HEADER BODY ; Id OR y.in = [a,c] ; out = [!y];
HEADER NEWGATE ; Id ; OR y.in = [a,c] ; out = [!y];
HEADER Type Id.in = [i,i] ; Id ; OR y.in = [a,c] ; out = [!y];
HEADER Type Id.in = [a,b] ; Id ; OR y.in = [a,c] ; out = [!y];
HEADER Type x.in = [a,b] ; Id ; OR y.in = [a,c] ; out = [!y];
HEADER AND x.in = [a,b] ; Id ; OR y.in = [a,c] ; out = [!y];

n; AND x.in = [a,b] ; Id ; OR y.in = [a,c] ; out = [!y];
m; AND x.in = [a,b] ; Id ; OR y.in = [a,c] ; out = [!y];
Id,m; AND x.in = [a,b] ; Id ; OR y.in = [a,c] ; out = [!y];
Id,Id,m; AND x.in = [a,b] ; Id ; OR y.in = [a,c] ; out = [!y];
Id,Id,Id; AND x.in = [a,b] ; Id ; OR y.in = [a,c] ; out = [!y];
a,b,c; AND x.in = [a,b] ; Id ; OR y.in = [a,c] ; out = [!y];

```
# Example 4 (not practical, but a valid program)
{
$a,b$
    bool x = !a;
}
```

r
s
HEADER BODY
HEADER EXPR;
HEADER DEFBOOLS;
HEADER bool Id = OPTIONALNEG i;
HEADER bool Id = OPTIONALNEG Id;
HEADER bool Id = OPTIONALNEG a;
HEADER bool Id = ! a;
HEADER bool x = ! a;
n; bool x = ! a;
m; bool x = ! a;
Id,m; bool x = ! a;
Id,Id; bool x = ! a;
Id,b; bool x = ! a;
a,b; bool x = ! a;