

BLIF Netlist

Stephen A. Edwards
Columbia University
sedwards@cs.columbia.edu

Abstract

This package represents a gate-level netlist. Each gate is a generalized AND gate, each of whose inputs and output may be inverted separately. Nets are all single-driver.

Contents

1	Gate, Input, and Netlist Classes	2
1.1	Gate	2
1.2	Input	3
1.3	Netlist	3
2	BLIF reader	3
2.1	getGate	4
2.2	.model	5
2.3	.inputs and .outputs	7
2.4	.names	8
2.5	.latch	10
2.6	readLine	10
3	BLIF Writer	12
4	Dot Printer	13
5	Verilog Writer	15
6	The Simulator	18
6.1	Constructor and Destructor	19
6.2	reset	20
6.3	simulate	21
7	Netlist.hpp and .cpp	22

1 Gate, Input, and Netlist Classes

1.1 Gate

The gate is the fundamental object. It computes the logical AND of its inputs, and each input may be inverting, as may the output. The `newInput` and `newOutput` methods add connections.

The output of a gate with no inputs is 1 if it is non-inverting, 0 if it is inverting.

A latch is a single-input, single-output gate. Its `is_inverting` flag is true if it resets to 1, and false if it resets to 0. Its single input should not be inverting.

```
2  <gate class 2>≡ (22)
    class Gate {
        friend class Netlist;

        Gate(Netlist *nl, unsigned int i, string n, bool inv = false)
            : parent(nl), id(i), name(n), is_inverting(inv),
              is_input(false), is_output(false), is_latch(false) {}
    public:
        Netlist *parent;
        unsigned int id;
        string name;
        vector<Input*> inputs;
        bool is_inverting;
        vector<Input*> outputs;

        bool is_input;
        bool is_output;
        bool is_latch;

        void newInput(Gate *g, bool i = false) {
            assert(g);
            Input *ni = new Input(g, this, i);
            inputs.push_back(ni);
            g->outputs.push_back(ni);
        }

        void newOutput(Gate *g, bool i = false) {
            assert(g);
            g->newInput(this, i);
        }
    };
};
```

1.2 Input

An **Input** object is owned by the gate for which it is an input. Each may be inverting or true. Use the gate **newInput** and **newOutput** methods to create them.

3a $\langle \text{input class 3a} \rangle \equiv$ (22)

```
class Input {
    friend class Gate;
    Input(Gate *d, Gate *g, bool i = false)
        : driver(d), gate(g), is_inverting(i) {}
public:
    Gate *driver;
    Gate *gate;
    bool is_inverting;
};
```

1.3 Netlist

A netlist is a named collection of gates. Each gate in a netlist has a name and a unique ID number, assigned when a new gate is created.

3b $\langle \text{netlist class 3b} \rangle \equiv$ (22)

```
class Netlist {
public:
    string name;
    Netlist(string n) : name(n) {}
    vector<Gate*> gates;
    Gate *newGate(bool = false, string = "");
};
```

The **newGate** method create a new, unconnected gate in the netlist. If the given name is empty, a unique name is assigned to it.

3c $\langle \text{netlist methods 3c} \rangle \equiv$ (23)

```
Gate *Netlist::newGate(bool inv, string name)
{
    if (name.empty()) {
        std::ostringstream longname;
        longname << "g" << gates.size();
        name = longname.str();
    }
    Gate *g = new Gate(this, gates.size(), name, inv);
    gates.push_back(g);
    return g;
}
```

2 BLIF reader

3d $\langle \text{blif reader declaration 3d} \rangle \equiv$ (22)

```
Netlist *read_blif(std::istream &);
```

4a \langle blif reader definition 4a $\rangle \equiv$ (23)

```
Netlist *read_blif(std::istream &i)
{
    try {
        BlifReader r(i);
        if (i.eof()) throw BlifReader::Error(0, "no model found");
        return r.readModel();
    } catch (BlifReader::Error e) {
        std::cerr << e.lineNumber << ':' << e.error << std::endl;
        return 0;
    }
}
```

4b \langle blif reader class 4b $\rangle \equiv$ (22)

```
class BlifReader {
    Netlist *netlist;
    std::istream &inf;
    std::map<string, Gate*> namedGate;
    string line;
    vector<string> word;
    unsigned lineNumber;

    struct Row {
        string andplane;
        char orplane;
        Row(string s, char c) : andplane(s), orplane(c) {}
    };

public:
    BlifReader(std::istream &ii) : netlist(0), inf(ii), lineNumber(0) {}

    struct Error {
        unsigned lineNumber;
        string error;
        Error(unsigned l, string s) : lineNumber(l), error(s) {}
    };

     $\langle$ blif reader methods 4c $\rangle$ 
};
```

2.1 getGate

This returns a gate with the given name or creates a new one.

4c \langle blif reader methods 4c $\rangle \equiv$ (4b) 5b \triangleright

```
Gate *getGate(string = "");
```

5a $\langle \text{blif reader method definitions } 5a \rangle \equiv$ (23) 6▷

```

Gate *BlifReader::getGate(string n)
{
    if (n.empty() || namedGate.find(n) == namedGate.end()) {
        Gate *result = netlist->newGate(false, n);
        namedGate.insert( std::make_pair(result->name, result));
        return result;
    } else {
        return namedGate[n];
    }
}

```

2.2 .model

5b $\langle \text{blif reader methods } 4c \rangle + \equiv$ (4b) ◁4c 10b▷

```

Netlist *readModel();

```

```

6  <blif reader method definitions 5a>+≡                                     (23) <5a 10c>
    Netlist *BlifReader::readModel()
    {
        readLine();
        if (word.empty() || word[0] != ".model")
            throw Error(lineNumber, "expecting .model, found '" + line + '\n');
        if (word.size() != 2)
            throw Error(lineNumber, "too many words after .model");
        netlist = new Netlist(word[1]);
        if (inf.eof()) throw Error(lineNumber, "empty model");
        readLine();
        do {
            if (word.empty()) throw Error(lineNumber, "empty line?");
            if (word[0] == ".inputs") {
                <read inputs 7a>
            } else if (word[0] == ".outputs") {
                <read outputs 7b>
            } else if (word[0] == ".names") {
                <read names 8>
            } else if (word[0] == ".latch") {
                <read latch 10a>
            } else if (word[0] == ".end") {
                readLine();
                break;
            } else throw Error(lineNumber, "unrecognized '" + word[0] + "'");
        } while (!inf.eof());

        /*
        std::cout << "Line:" << line << "----" << std::endl;
        for (vector<string>::const_iterator i = word.begin() ;
            i != word.end() ; i++ ) {
            std::cout << "'" << *i << "'" << std::endl;
        }
        */

        return netlist;
    }

```

2.3 .inputs and .outputs

These simply create each input or output.

- 7a $\langle read\ inputs\ 7a \rangle \equiv$ (6)
- ```
 for (vector<string>::const_iterator i = (word.begin()) + 1 ;
 i != word.end() ; i++) {
 Gate *new_input = getGate(*i);
 new_input->is_input = true;
 }
 readLine();
 continue;
```
- 7b  $\langle read\ outputs\ 7b \rangle \equiv$  (6)
- ```
    for (vector<string>::const_iterator i = (word.begin()) + 1 ;  
        i != word.end() ; i++) {  
        Gate *new_output = getGate(*i);  
        new_output->is_output = true;  
    }  
    readLine();  
    continue;
```

2.4 .names

This is the real meat: each BLIF node encapsulates a PLA. In our representation, each is expanded to AND gates driving an OR gate.

```

8  <read names 8>≡ (6)
    vector<Gate*> inputs;
    for (unsigned int i = 1 ; i < word.size() - 1 ; i++)
        inputs.push_back(getGate(word[i]));
    Gate *output = getGate(word[word.size()-1]);

    vector<Row> rows;
    readLine();
    while (!(inf.eof() || word.empty() || word[0][0] == '.')) {
        if ( !((word.size() == 1 && inputs.empty()) ||
            word.size() == 2) )
            throw Error(lineNumber, "syntax error in PLA");

        string andplane = (word.size() == 2) ? word[0] : "";
        string orplane = (word.size() == 2) ? word[1] : word[0];
        if (andplane.size() != inputs.size() || orplane.size() != 1)
            throw Error(lineNumber, "PLA wrong width");
        if (andplane.find_first_not_of("01-") != std::string::npos)
            throw Error(lineNumber, string("bad character in AND plane of PLA"));
        if (orplane != "0" && orplane != "1")
            throw Error(lineNumber, string("bad character in OR plane PLA"));
        rows.push_back(Row(andplane, orplane[0]));

        readLine();
    }

    switch (rows.size()) {
    case 0:
        // Empty row: a constant 0
        output->is_inverting = true;
        break;

    case 1:
        // A single row: a single gate
        output->is_inverting = (rows.front().orplane == '0');
        for ( unsigned int i = 0 ; i < rows.front().andplane.size() ; i++ ) {
            char v = rows.front().andplane[i];
            if ( v != '-' ) output->newInput(inputs[i], v == '0');
        }
        break;

    default:
        // Two or more rows: the output gate acts as an OR (or NOR, if the orplane
        // is 0s); the others act as ANDs
        {
            output->is_inverting = (rows.front().orplane == '1');

```



```

for ( vector<Row>::const_iterator i = rows.begin() ;
      i != rows.end() ; i++ ) {
    const string &andplane = (*i).andplane;
    int uniqueInput = -1;
    for ( string::const_iterator j = andplane.begin() ;
          j != andplane.end() ; j++ ) {
        if (*j != '-') {
            if (uniqueInput == -1)
                uniqueInput = j - andplane.begin();
            else
                uniqueInput = -2;
        }
    }

    if (uniqueInput >= 0) {

        // Single unique input: connect it to the output gate directly
        // and invert it, since it is not an OR gate
        output->newInput(inputs[uniqueInput], andplane[uniqueInput] == '1');

    } else {
        // Multiple care inputs: generate an AND gate and connect it to
        // the output
        Gate *andplanegate = getGate();
        andplanegate->is_inverting = false;
        output->newInput(andplanegate, true);
        for ( string::const_iterator j = andplane.begin() ;
              j != andplane.end() ; j++ )
            if (*j != '-')
                andplanegate->newInput(inputs[j - andplane.begin()], (*j) == '0');
    }
}
break;
}

```

2.5 .latch

The single-line .latch directive creates a latch.

```

10a  <read latch 10a>≡ (6)
      if (word.size() < 3)
        throw Error(lineNumber, "too few arguments to .latch: need at least two");
      if (word.size() > 6)
        throw Error(lineNumber, "too many arguments to .latch: no more than five");
      if (word.size() == 5)
        throw Error(lineNumber, "wrong number of arguments to .latch");
      Gate *latch = getGate(word[2]);
      latch->is_latch = true;
      latch->newInput(getGate(word[1]), false);
      if (word.size() == 4)
        latch->is_inverting = (word[3] == "1");
      if (word.size() == 6)
        latch->is_inverting = (word[5] == "1");
      readLine();
      continue;

```

2.6 readLine

This reads a single line into the `line` field. Multiple spaces are condensed into one, and comments (start at #, end at newline) are ignored. A “followed immediately by a newline is treated as a space.

```

10b  <blif reader methods 4c>+≡ (4b) <5b
      void readLine();

10c  <blif reader method definitions 5a>+≡ (23) <6
      void BlifReader::readLine()
      {
        <read line 11a>
        <tokenize 11b>
      }

```

The first task is lexical analysis on the whole line: discard comments, fuse spaces, and merge \-continued lines

11a *<read line 11a>*≡ (10c)

```

char c;
line.resize(0);
while ( c = inf.get(), !inf.eof()) {
    if ( c == '\\') {
        if ( (c = inf.get()) == '\\n') {
            lineNumber++;
            c = ' ';
        }
    } else if ( c == '#' ) {
        do {
            c = inf.get();
            if (inf.eof()) goto done;
        } while (c != '\\n');
    }
    if ( c == '\\n' && !line.empty() ) {
        lineNumber++;
        break;
    }
    switch (c) {
    case '\\n':
        lineNumber++;
        // FALLTHROUGH
    case ' ':
    case '\\t':
    case '\\f':
        c = ' ';
        if ( line.empty() || *(line.rbegin()) == ' ' )
            continue; // condense multiple spaces
        // FALLTHROUGH
    default:
        line += c;
    }
}
done:

```

Next, the line is broken into words by alternately skipping whitespace and scanning over words.

11b *<tokenize 11b>*≡ (10c)

```

word.resize(0);
string::const_iterator i = line.begin();
string::const_iterator lastWordStart;
do {
    while (i != line.end() && *i == ' ') i++;
    lastWordStart = i;
    while (i != line.end() && *i != ' ') i++;
    if (i != lastWordStart) word.push_back(string(lastWordStart, i));
} while (i != line.end());

```

3 BLIF Writer

This is much easier than reading: walk through each node, printing a simple truth table for each. Note that read followed by write generally does not produce a textually identical file because of additional, unnamed gates.

12a $\langle \text{blif printer declaration 12a} \rangle \equiv$ (22)

```
void print_blif(std::ostream &, const Netlist &);
```

12b $\langle \text{blif printer definition 12b} \rangle \equiv$ (23)

```
void print_blif(std::ostream &o, const Netlist &n)
{
    o << ".model " << n.name << '\n';
    o << ".inputs";
    for ( vector<Gate*>::const_iterator i = n.gates.begin() ;
          i != n.gates.end() ; i++ )
        if ( (*i)->is_input ) o << ' ' << (*i)->name;
    o << '\n';

    o << ".outputs";
    for ( vector<Gate*>::const_iterator i = n.gates.begin() ;
          i != n.gates.end() ; i++ )
        if ( (*i)->is_output ) o << ' ' << (*i)->name;
    o << '\n';

    for ( vector<Gate*>::const_iterator i = n.gates.begin() ;
          i != n.gates.end() ; i++ ) {
        Gate &g = *(*i);
        if ( !g.is_input && !g.is_latch ) {
            o << ".names";
            for ( vector<Input*>::const_iterator j = g.inputs.begin() ;
                  j != g.inputs.end() ; j++ )
                o << ' ' << (*j)->driver->name;
            o << ' ' << g.name << '\n';
            for ( vector<Input*>::const_iterator j = g.inputs.begin() ;
                  j != g.inputs.end() ; j++ )
                o << ((*j)->is_inverting ? '0' : '1');
            o << ' ';
            o << (g.is_inverting ? '0' : '1') << '\n';
        } else if ( g.is_latch ) {
            assert(g.inputs.size() == 1);
            o << ".latch " << g.inputs.front()->driver->name << ' ' << g.name << ' '
              << (g.is_inverting ? '1' : '0') << '\n';
        }
    }
    o << ".end" << std::endl;
}
```

4 Dot Printer

This prints the netlist in a form suitable for the *dot* tool, part of the AT&T *graphviz*. It is very straightforward: each gate becomes a node. Arcs indicate wires, and bubbles at the beginning and ending of arcs indicate the phase of the gates and their inputs.

13 $\langle \textit{dot printer declaration 13} \rangle \equiv$ (22)
 `void print_dot(std::ostream &, const Netlist &);`

```

14  <dot printer definition 14>≡ (23)
    void print_dot(std::ostream &o, const Netlist &n)
    {
        o << "digraph " << n.name << " {\n"
        "rankdir=\"LR\"\n"
        "node [shape=\"box\"]\n"
        "size=\"10,7.5\"\n"
        ;

        for ( vector<Gate*>::const_iterator i = n.gates.begin() ;
              i != n.gates.end() ; i++ ) {
            Gate &g = *(*i);
            if (g.is_latch) {
                o << 'g' << g.id << "q [label=\"" << g.id << ':' << g.name
                << " Q\" style=filled color=beige]\n";
            }
            o << 'g' << g.id << " [label=\"" << g.id << ':' << g.name;
            if (g.is_inverting) o << "~";
            o << g.name;
            if (g.is_latch) o << " D";
            o << '\n';
            if (g.is_input)
                o << " style=filled color=palegreen1 shape=house orientation=-90";
            else if (g.is_output)
                o << " shape=house orientation=-90 style=filled color=pink1";
            else if (g.is_latch)
                o << " style=filled color=green1";
            o << "]\n";
            for ( vector<Input*>::const_iterator j = g.inputs.begin() ;
                  j != g.inputs.end() ; j++ ) {
                Input &input = *(*j);
                o << 'g' << input.driver->id;
                if (input.driver->is_latch) o << "q";
                o << " -> g" << g.id << " [arrowtail=";
                if (input.driver->is_inverting) o << "invodot"; else o << "inv";
                o << " arrowhead=";
                if (input.is_inverting) o << "odot"; else o << "none";
                o << "]\n";
            }
        }
    }

    o << "}\n";
}

```

5 Verilog Writer

This dumps the netlist in a structural Verilog netlist form.

15 $\langle \textit{verilog writer declaration 15} \rangle \equiv$ (22)
 void print_verilog(std::ostream &, const Netlist &);

```

16  <verilog writer definition 16>≡ (23)
    string &renw(string &s) // rename wire
    {
        int lc = s.length() - 1;
        if(s[0]=='[' && s[lc]==']'){
            s[0]=s[lc]='_';
        }
        return s;
    }

    void print_verilog(std::ostream &o, const Netlist &n)
    {
        o << "module " << n.name << "(clk, rst";

        for ( vector<Gate*>::const_iterator i = n.gates.begin() ;
              i != n.gates.end() ; i++ )
            if ( (*i)->is_output ) o << ", " << (*i)->name;

        for ( vector<Gate*>::const_iterator i = n.gates.begin() ;
              i != n.gates.end() ; i++ )
            if ( (*i)->is_input ) o << ", " << (*i)->name;

        o << ");\n";

        o << "input clk, rst;\n";

        for ( vector<Gate*>::const_iterator i = n.gates.begin() ;
              i != n.gates.end() ; i++ )
            if ( (*i)->is_output ) o << "output " << (*i)->name << ";\n";

        for ( vector<Gate*>::const_iterator i = n.gates.begin() ;
              i != n.gates.end() ; i++ )
            if ( (*i)->is_input ) o << "input " << (*i)->name << ";\n";

        o << '\n';

        for ( vector<Gate*>::const_iterator i = n.gates.begin() ;
              i != n.gates.end() ; i++ )
            o << "wire " << renw((*i)->name) << ";\n";

        o << '\n';

        unsigned int instnum = 0;

        for ( vector<Gate*>::const_iterator i = n.gates.begin() ;
              i != n.gates.end() ; i++ ) {
            if ((*i)->is_input) continue;
            else if ((*i)->is_latch) {
                assert((*i)->inputs.size() == 1);
                o << "d_ff" << ((*i)->is_inverting ? '1' : '0') << " u" << instnum++

```



```

        << "(rst, clk, " << renw((*i)->name) << ", "
        << renw((*i)->inputs.front()->driver->name) << ");\n";
    } else {
        // Normal gate: becomes an assignment
        o << "assign " << renw((*i)->name) << " = ";
        if ( (*i)->inputs.size() == 0 ) {
            // No inputs: a constant
            o << ((*i)->is_inverting ? '0' : '1');
        } else {
            // Some form of AND or OR gate
            for ( vector<Input*>::const_iterator j = (*i)->inputs.begin() ;
                j != (*i)->inputs.end() ; j++ ) {
                if ( j != (*i)->inputs.begin() )
                    o << ' ' << ((*i)->is_inverting ? '|' : '&') << ' ';
                if ( (*j)->is_inverting != (*i)->is_inverting ) o << '!';
                o << renw((*j)->driver->name);
            }
        }

        o << ";\n";
    }
}

o << "\nendmodule\n";
}

```

6 The Simulator

This is a very simple simulator for these netlists. It operates in binary and requires the network to be acyclic when latches are removed.

To use, set the values of the inputs using `setInput()`, call `simulate()`, and fetch the values of the outputs using `[[getOutput()]`. The next states of the latches can be obtained by calling `getLatch()`.

```

18a  <blif simulator class 18a>≡ (22)
      class Simulator {
        Netlist &n;

        unsigned int next;
        Gate **topoorder; // Array of gate pointers in topological order

        bool *currentState; // State of latches, indexed by gate number
        bool *nextState; // State of inputs and gates/latches after simulation

        void dfs(Gate*, set<Gate*> &);
      public:
        vector<Gate*> inputs;
        vector<Gate*> outputs;
        vector<Gate*> latches;

        void setInput(Gate* g, bool v) { currentState[g->id] = v; }
        bool getOutput(Gate* g) { return currentState[g->id]; }
        bool getLatch(Gate* g) { return nextState[g->id]; }

        Simulator(Netlist &);
        ~Simulator();

        void reset();
        void simulate();

        static bool debug;
      };

      Debugging is disabled by default. A caller may enable this.
18b  <blif simulator definitions 18b>≡ (23) 19a>
      bool Simulator::debug = false;

```

6.1 Constructor and Destructor

The constructor takes a netlist, builds some fixed arrays, enumerates the inputs, outputs, and latches, and uses the `dfs` method to topologically sort the gates.

```
19a  <blif simulator definitions 18b>+≡ (23) <18b 19b>
      Simulator::Simulator(Netlist &nl) : n(nl)
      {
          unsigned int ngates = n.gates.size();
          topoorder = new Gate*[ngates];
          currentState = new bool[ngates];
          nextState = new bool[ngates];

          // Populate the inputs, outputs, and latches vectors
          for (vector<Gate*>::const_iterator i = n.gates.begin() ;
              i != n.gates.end() ; i++) {
              if ((*i)->is_input) inputs.push_back(*i);
              if ((*i)->is_output) outputs.push_back(*i);
              if ((*i)->is_latch) latches.push_back(*i);
          }

          // Topologically sort the gates
          next = 0;
          set<Gate*> visited;
          for (vector<Gate*>::const_iterator i = n.gates.begin() ;
              i != n.gates.end() ; i++ ) dfs(*i, visited);
          assert(next == n.gates.size());

          reset();
      }
```

The destructor deletes the static arrays created in the constructor.

```
19b  <blif simulator definitions 18b>+≡ (23) <19a 20a>
      Simulator::~~Simulator()
      {
          delete [] nextState;
          delete [] currentState;
          delete [] topoorder;
      }
```

The `dfs` method topologically sorts the gates in the netlist, ignoring gate inputs that come from latches.

```
20a  <blif simulator definitions 18b>+≡ (23) <19b 20b>
      void Simulator::dfs(Gate *g, set<Gate*> &visited)
      {
        if (visited.find(g) != visited.end()) return;
        visited.insert(g);
        for ( vector<Input*>::const_iterator i = g->inputs.begin() ;
              i != g->inputs.end() ; i++ )
          if (!(i->driver->is_latch) dfs((i->driver), visited);
        topoorder[next++] = g;
      }
```

6.2 reset

Reset: initialize the latches to their initial values and clear all the inputs to 0.

```
20b  <blif simulator definitions 18b>+≡ (23) <20a 21>
      void Simulator::reset()
      {
        for ( vector<Gate*>::const_iterator i = latches.begin() ;
              i != latches.end() ; i++ )
          nextState[(i->id)] = (i->is_inverting);
        for ( vector<Gate*>::const_iterator i = inputs.begin() ;
              i != inputs.end() ; i++ ) currentState[(i->id)] = false;
      }
```

6.3 simulate

The main workhorse: copy the inputs of the latches to their outputs, then recompute every gate in topological order.

```

21  <blif simulator definitions 18b>+≡                                     (23) <20b
    void Simulator::simulate()
    {
        // The clock: copy the Ds of the latches to the Qs

        for (vector<Gate*>::const_iterator i = latches.begin() ;
            i != latches.end() ; i++ )
            currentState[(i->id)] = nextState[(i->id)];

        // Propagate the new current state information throughout the circuit

        for (unsigned int i = 0 ; i < n.gates.size() ; i++) {
            Gate *g = topoorder[i];
            if ( !(g->is_input) ) {
                bool state = true;
                for (vector<Input*>::const_iterator j = g->inputs.begin() ;
                    j != g->inputs.end() ; j++)
                    state =
                        state && ( currentState[(j->driver->id)] != ((*j->is_inverting) ) );
                if ( g->is_latch ) {
                    // Write the output of the latch to the next state array
                    nextState[g->id] = state;
                    if (debug)
                        std::cout << "latch " << g->name << '='
                                    << ( nextState[g->id] ? '1' : '0') << '\n';
                } else {
                    // Write the output of a normal gate
                    currentState[g->id] = (state != g->is_inverting);
                    if (debug)
                        std::cout << g->name << '='
                                    << ( currentState[g->id] ? '1' : '0') << '\n';
                }
            }
        }
    }
}

```

7 Netlist.hpp and .cpp

Boilerplate:

```

22  <BLIF.hpp 22>≡
    #ifndef _BLIF_HPP
    #   define _BLIF_HPP

    #   include <string>
    #   include <vector>
    #   include <assert.h>
    #   include <iostream>
    #   include <map>
    #   include <set>

    namespace BLIF {
        using std::vector;
        using std::string;
        using std::set;

        class Input;
        class Gate;
        class Netlist;

        <input class 3a>
        <gate class 2>
        <netlist class 3b>

        <dot printer declaration 13>

        <blif reader declaration 3d>
        <blif reader class 4b>

        <blif printer declaration 12a>
        <blif simulator class 18a>

        <verilog writer declaration 15>
    }

    #endif

```

```
23  <BLIF.cpp 23>≡
    #include "BLIF.hpp"
    #include <sstream>

    namespace BLIF {
        <netlist methods 3c>
        <dot printer definition 14>
        <blif reader definition 4a>
        <blif reader method definitions 5a>
        <blif printer definition 12b>
        <blif simulator definitions 18b>
        <verilog writer definition 16>
    }
```