CS 355 PA5 – Verilog Simulation Part 1

1 Introduction

In part 1 of this programming assignment you will parse and 'compile' a structural gate-level netlist in Verilog. In part 2 you will simulate the circuit given a set of input test vectors.

This programming assignment should be performed **in teams of 2 or individually**. You may re-use portions of code from previous examples and labs provided in this class, but any other copying of code is **prohibited**. We will use software tools that check all students' code and known Internet sources to find hidden similarities.

2 What you will learn

After completing this programming assignment you will:

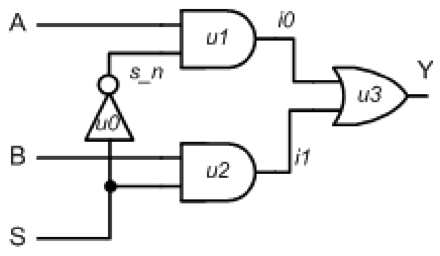
1. Use file and string streams to help parse a text file
2. Use inheritance and polymorphism concepts to model components
3. Use the STL map class to quickly index other objects
4. Perform topological sort/ DFS search algorithms
5. Have an understanding of the complexity of software CAD tools used in the practice of electrical engineering

3 Background Information and Notes

Verilog: Verilog is a popular hardware-description language used to describe digital circuits at various levels of complexity.

- RTL (register-transfer level) Verilog constructs allow high level operators and control structures to describe the operation of the circuit.
- Structural Verilog describes a circuit at the component level (i.e. as a set of black-boxes and their interconnections) and can use Verilog's built-in support for primitive logic gate types: **and**, **or**, **nand**, **nor**, **xor**, and **not**.

```
// Comment - this design a 2-to-1 mux
module mux21(a, b, s, y);
  input a;
  input b;
  input s;
  output y;
  wire s_n;
  wire i0;
  wire i1;
  not u0(s_n, s);
  and #2 u1(i0, s_n, a);
  and #2 u2(i1, s, b);
  or u3(y, i0, i1);
endmodule
```



2-to-1 Mux Design

In our subset of Verilog, we will only have one **Design** (module) per file. Within the design there can be **Gate** and **Net** (i.e. a wire that connects a gate output to another input) objects. Thus, the primary task in the first half of this project is to parse a Verilog design file and create the associate **Design** object which will have **Gate** and **Net** objects associated with it. Note that Net objects in Verilog are expressed in one of three ways: **inputs** (a.k.a. PI's or Primary Inputs which are the overall inputs to the design), **outputs** (a.k.a. PO's or Primary Outputs which are the overall outputs of the design), and **wires** (internal nets within the design). However, they are all just Net objects.

Net objects have **drivers** (gates that output to that net) and **loads** (gates that take input from that net). In addition, nets have a single value associated with it (while driver gates simply determine this value). Though we normally think of digital signals as simply '1' or '0', we will add a third value of 'X' (undetermined). During simulation of a design, all nets start with a value 'X' and then take on a '1' or '0' as input signals flow through the gate network. However, in the case of an improper design with multiple gates driving one net it is possible that one driver produces '0' while another produces '1'. In this case, the value of the net should also be 'X'. A net is referred to with a text identifier that serves as its name. This name must be stored and can be used to index any Net objects that are created.

Gates have a **type** (and, or, etc.), optional delay (defaults to 1 if not specified), as well as an **instance name** to refer to that gate and differentiate it from others. Gate objects also have a single output and some number of input nets associated with it. The output net is the first in the port list provided and the number of input nets to the gate is then determined by the number of nets provided after the output net (**note we can have ANY number of inputs…not just 2-input gates**).

Running your Program: Your program will take two command line arguments: the name of the Verilog file containing the design to parse and a test vector file containing test input cases for simulation purposes.

```
>./gatesim verilog_file input_test_file
```

Verilog Language Grammar: Just as languages like English have a grammar (set of rules) for the construction of larger units (sentences and paragraphs) using simpler ones (words and punctuation), programming and hardware description languages also have a grammar. This grammar defines the syntax of a valid design unit. These grammars are known as **context-free grammars** where there can be no ambiguity as to the grouping and format of expressions. In this project, we will use a simplified and more strict grammar to make your job of parsing easier. A common notation for context free grammars is BNF (Backus Normal Form). These grammars are a set of production rules (translation or rewrite rules) made of up terminal values (i.e. strings expressed as regular expressions) and other non-terminal values (i.e. other rules). Note that these rules can be recursive (put in terms of themselves). Also note that any number of spaces or tabs may occur between tokens/terminals.

```
Simplified CS355 Verilog Grammar in BNF:
<design>      ::= “module” <ID> <port_decl> “;” <EOL>
                <body_list> “endmodule” <EOL>

<ID>          ::= [A-Za-z_][A-Za-z0-9_]*

<num>         ::= [0-9]+

<port_decl>   ::= “(“ <port_list> “)”

<port_list>   ::= <port_list> “,” <ID> |
                  <ID>

<body_list>   ::= <body_list> <body_item> |
                  <body_item> |

<body_item>   ::= “input” <ID> “;” <EOL> |
                  “output” <ID> “;” <EOL> |
                  “wire” <ID> “;” <EOL> |
                  <gate_delay> <ID> <port_map> “;” <EOL> |
                  <EOL>

<gate_delay>  ::= <gate_type> |
                  <gate_type> <delay>

<delay>       ::= ‘#’<num>

<gate_type>   ::= “and” | “or” | “nand” | “nor” | “xor” | “not”

<port_map>    ::= “(“ <ID> “,” <in_list> “)” “;”

<in_list>     ::= <in_list> “,” <ID> |
                  <ID>
```

File I/O and Stringstreams: By now you should have experience processing input text files using file streams. But there are some other helpful functions and classes provided in the standard C++ library that can help to parse your Verilog files.

- The istream operator >> provides a handy way of extracting data from a stream of text characters into the desired variable and format. The input stream used is often ‘cin’ (i.e. the keyboard) or an input file. However, occasionally it may be advantageous to take an internal text string (i.e. just a string variable in memory) and extract portions of it using the ‘>>’ operator. To treat a string of text as a stream that we can read from (‘>>’) or write to (‘<<’) we can create a stringstream object. See the example below:
  ```
  string text;
  int num;
  double val;
  stringstream ss(“Hello 355 2.0”);
  ss >> text >> num >> val;
  ```
  Now text = “Hello”, num = 355 (as an int), val = 2.0 (as a double)

- To read an entire line of text (and not just a single value stopping at whitespace) into a C++ string you can use the global function (i.e. not a member of any object/class) `getline`:
  ```
      istream& getline(istream &is, string &str);
  ```
  Pass this function an input stream (i.e. your input file stream object or a stringstream object) and a C++ string object and it will read all the text character until the end-of-line ('\n') *it will discard the '\n'+ into `str`.

- To read all the text (and not just a single value stopping at whitespace) into a C++ string but STOPPING at a particular character *such as a '(', ')' or ';'+ another version of getline exists where you may pass it the delimiter value you want to stop at:
  ```
      istream& getline(istream &is, string &str, char delim);
  ```
  The text from the input stream will be read up through the first occurrence of 'delim' and placed into `str`. The delimiter will be stripped from the end of `str` and the input stream will be pointing at the first character after 'delim'.

  ```
      int line_no = 0;
      ifstream myfile(...); string line_of_text;
      getline(myfile, line_of_text);
      line_no++;
      stringstream ss(line_of_text);
      string token; ss >> token;
      if(token == "module"){
        string next_token;
        getline(ss, next_token, „(„); // read up through „(„
        string port_list;
        getline(ss, port_list, „)"); // read up through „)"
        // we can even make another stringstream of port_list
        // and extract pieces from it)
        ...
  ```
- You may need to check if the '>>' operator was actually able to read a value [because what if the remaining stream contains only whitespace]. To do this check the fail bit of the streams status with the fail() member function.

  ```
      stringstream ss(line_of_text);
      string token; ss >> token;
      if(ss.fail()){
        // no text on this line
        line_no++;
        continue;
      }
  ```

Circular References and Forward Declarations: Sometimes we need to define two (or more) classes that each reference each other. Thus we can't include one before the other. In this case we can include a forward declaration with the actual class definition following later:

```
class Gate; // forward dec. of
Gate class Net {
  void add_driver(Gate *g);
}
```

```
class Net; // forward dec. of
Net class Gate {
  void add_input(Net *n);
}
```

Class Definitions: The following class definitions are provided for Design, Net, and Gate objects. **<u>They are suggestions and not requirements so you can change them as desired</u>**. Also, note that class 'Gate' is an abstract base class and is meant to be used to derive specific gate types: AND, OR, etc.

```
class Net {
 public:
  Net();
  Net(string n);
  ~Net();
  void addDriver(Gate *g);
  void addLoad(Gate *g);
  vector<Gate *>   *getLoads();
  vector<Gate *> *getDrivers();
  string name();
  void setVal(char v);
  char computeVal();
  char getVal();
  void printDriversLoads();
 private:
  vector<Gate *> *drivers;
  vector<Gate *> *loads;
  char val;
  string netname;
};
```

Net Class Definition

```
class Gate {
 public:
  Gate();
  Gate(string n, int d);
  virtual ~Gate();
  virtual char eval() = 0;
  virtual void dump(ostream &os) = 0;
  string name();
  void addInput(Net *n);
  void addOutput(Net *n);
  vector<Net *> *getInputs();
  Net* getOutput();
  int getNumInputs();
  int getDelay()
 protected:
  vector<Net *> *inputs;
  Net *output;
  string inst_name;
  int delay;
};
```

Gate Base Class Definition
(Specific gate types can be derived from this)

```
enum {AND, OR, NAND, NOR, XOR, NOT};
class Design {
 public:
  Design();
  Design(string n); // contains the name of the design
  ~Design();
  string name(); // returns the name of the design
  void addPI(string n); // add an input‟s name to the list of PIs
  void addPO(string n); // add an output‟s name to the list of POs
  Net *findNet(string net_name); // returns NULL or a pointer to
                                  // the specified Net object
  Gate *findGate(string inst_name); // same as find_net but with Gates
  Net* addFindNet(string n); // returns a pointer to the Net object with
      // the given name or if a net with the given namd does not exist,
      // allocates a new one and returns a pointer to it
  Gate* addFindGate(int gtype, string n); // same as add_find_net but
      //   for gates. gtype is the enum value of the gate type
  vector<Net *> *getPINets();    // Allocates & creates a new vector
                                   // of pointers to the primary input Nets
  vector<Net *> *getPONets();    // Similar to get_pi_nets()
  vector<Net *> *getWireNets();  // Similar to get_pi_nets()
  vector<Net *> *allNets();       // Returns pointers to all Net objects
  vector<Gate *> *allGates();     // Similar to all_nets() but for gates
  void dump(ostream &os);          // Write out the design in Verilog format
                                   //     to the provided ostream
 private:
  string desname;
  map<string, Net *> designNets;
  map<string, Gate *> designGates;
  vector<string> pis;
  vector<string> pos;
};
```

Design Class Definition

4 Requirements

Your program shall meet the following requirements for features and approach:

Part 1

1) Implement a Design class that performs similar functionality to the description provided earlier. You must use a map to associate gate and net names with a pointer to the corresponding object.
2) Net and Gate objects (really the specific derived objects from Gate) must be dynamically allocated) and destructors should free memory when the container classes (vectors) go out of scope.
   a) The list of driver gates must be dynamically allocated and store pointers to Gate objects
   b) The list of load gates must be dynamically allocated and store pointers to Gate objects
3) Implement the non-pure virtual functions of the Gate class that perform similar functionality to the description provided earlier. Implement the virtual destructor to deallocate the vectors you create for each gate.

   a) The list of input nets must be dynamically allocated and store pointers to Net objects
   b) If a delay is not provided in the Verilog file, its default value should be 1 but when we dump the design back to a file, we would not want to print out a delay of 1 (instead just omitting it as in the original file). However, if the delay is not provided, getDelay() should return the default value of 1.
4) Derive And, Or, Nand, Nor, Xor, and Not classes from Gate class.
5) Implement a Net class that performs similar functionality to the description provided earlier.
6) Implement a parse function (or class with appropriate function member) to parse the specified Verilog file. This function should return a pointer to a new Design object that describes the design if parsing is successful and NULL otherwise. It should also:
   a) If the input Verilog file does not meet the specified grammar, output an error message and the line number of where the first parse error occurs, and exit the program taking care to deallocate any necessary memory.
   b) Create net objects and enter them into a map in the Design object associating the net name to a Net object
   c) Create gate objects and enter them into a map in the Design object associating the instance name to the derived gate type: And, Or, Nand, Nor, Xor, Not
   d) Appropriately add the correct nets as the output and to the list of inputs of each gate
   e) Appropriately add the correct gates to the list of drivers and loads for each net
7) Each derived Gate class should implement a virtual dump() function that shall output the gate's type, instance name and port mapping in the correct format/syntax to the specified ostream object.
8) In the Design class, implement a "dump" method that will re-generate the design file (without re-reading the input file or storing it verbatim) and write it to a file "output.v". Obviously the comments and whitespace orientation from the original, input Verilog file do not need to be maintained and the order in which things (e.g. wires or gates) are dumped can be slightly different, but the output file must be able to be read back in by your parser.
9) Write a main program that calls your parse function and if parsing is successful dumps the design to 'output.v'.

5 Procedure

Perform the following.

1. Create a 'pa5' directory in your account on parallel05.usc.edu
   ```
   > mkdir pa5
   > cd pa5
   ```
2. Copy the sample images down to your local directory:
   ```
   > tar xvf ~redekopp/rel/gatesim.tar
   ```
   This will download the following files:

   `gates.h` – Gate base class. Define all the derived classes in this file too.

   `net.h` – Net class

   `design.h` – Design class

   `test.v` – Sample Verilog file 1

   `test2.v` – Sample Verilog file 2
3. You may create a 'Codeblocks' project in this directory and import the .h files, or just create a Makefile. Again you are free to modify these classes as you desire.
4. Complete the parsing and design creation functionality specified in the Requirements section. Be sure your parser detects major parse errors such as:
   - Missing keywords
   - Misspelled keywords
   - Missing output nets on a gate
   - Not enough input nets to a gate (at least one to each gate)

   For more credit, attempt to detect smaller parse errors such as:
   - Missing commas or parentheses.
   - Missing semicolons
   - Misformed delay items

   You may assume EOL (new lines) will be correctly placed in all cases (i.e. gate and wire declarations will not erroneously span multiple lines of the input file).
5. Submit one copy of your completed code including main.cpp in a single ZIP file via Blackboard under just one of your team's accounts. **Indicate both team members name as a comment at the top of gatesim.cpp (your main program)**

Teammate Names: ____

Item	Outcome	Score	Max
Code submitted appropriately in a ZIP file	Yes / No		2
Both team members' name are included at the top of main.cpp	Yes / No		1
Compiles successfully in CodeBlocks	Yes / No		3
Code Design			
• Classes broken into .h and .cpp files	Yes / No		1
• Appropriately derives specific gate types from the Gate base class	Yes / No		2
• Each derived gate class implementing only the functions that provided different behavior for that gate type	Yes / No		3
Design Class			
• Uses map to associate gate name with corresponding Gate object	Yes / No		1
• Uses map to associate net name with corresponding Net object	Yes / No		1
• Dynamically allocates Net and Gate objects	Yes / No		3
• Deallocates Net and Gate objects appropriately	Yes / No		2
Net Class			
• Dynamically allocates vectors to store pointers to the driver and load gates	Yes / No		1
• Deallocates the drivers & loads vectors in the deconstructor	Yes / No		1
• Adds driver and load gates appropriately	Yes / No		2
Gate Class(es)			
• Dynamically allocates vectors to store pointers to input nets	Yes / No		1
• Deallocates the inputs vector in the deconstructor	Yes / No		1
• Appropriate dump() function is implemented for each derived gate type	Yes / No		1
• Adds input and output nets appropriately	Yes / No		2
• Handles delays and default delays appropriately	Yes / No		1
Parsing and Dumping			
• Successfully parses and dumps test.v	Yes / No		5
• Successfully parses and dumps test2.v	Yes / No		5
• Correctly indicates line number of the first parse error, if any	Yes / No		2
• Appropriately detects parse errors (input files that do not meet the specified grammar) [Comments]: ______________________________ ______________________________ ______________________________ ______________________________	_______		6
Well-commented code and neatly formatted	Yes / No		3
Subtotal			50
Late Deductions (-10% per day)			
Total			