EE 355 PA6 – Verilog Gate 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 assignment has a "simpler" option where you can use our provided parser and still get full credit or a "harder" option where you implement the parsing yourself for extra credit.

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:

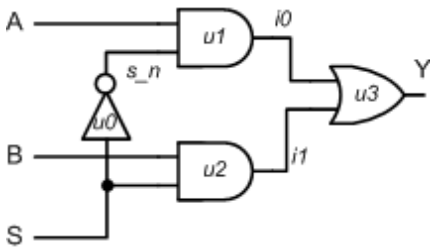
1. Use file and string streams to help parse a text file
2. Use inheritance and polymorphism concepts to model components
3. Use the STL map class to quickly index other objects
4. Perform topological sort/ DFS search algorithms
5. Have an understanding of the complexity of software CAD tools used in 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 first is the name of the Verilog file containing the design to parse. The program should then output the design you just read to a new file given as the 2nd command line argument.

```
$ ./gatesim verilog_input_file verilog_output_file
```

Parsing, File I/O and Stringstreams (For Extra Credit): We have provided you a full featured parser specified as a FLEX (verilog.lex) and Bison (verilog.y) grammar. It will read in the input files and use your class definitions to create object models of the design.

Verilog Language Grammar: The provided grammar is derived from an open-source software release (PythonGND-0.35 - www-cad.eecs.berkeley.edu/~pinhong/scriptEDA/pythongnd-0.35.tar.gz). It is under the GPLv2 license so any work you create from it is also open-source (sorry, you can't start your own company using this software).

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 ee355 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>
```

However, for extra credit you can implement your own “parser” with ifstreams, stringstreams, etc. If you choose this approach you do NOT have to parse the Verilog files using a grammar or by trying to implement the rewrite rules above. Instead you can just write code that will parse and check for the syntax that the above rules describe (i.e. your parser should implement the same behavior but can do it using any method of your choice). 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>You may add helper functions 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(string n);
  ~Net();
  void addDriver(Gate *g);
  void addLoad(Gate *g);
  vector<Gate *>* getLoads();
  vector<Gate *>* getDrivers();
  string name() const;
  char getVal() const;
  void setVal(char v);
  char computeVal();
  int computeDelay();
  int getDelay() const;
  void setDelay(int d);
  void printDriversLoads();
 private:
  vector<Gate *> *drivers;
  vector<Gate *> *loads;
  char val;
  string netname;
};
```

Net Class Definition

```
class Gate {
 public:
  Gate(string n); // name only
  Gate(string n, int d); // with delay
  virtual ~Gate();
  string name() const;
  int getDelay() const;
  virtual char eval() = 0;
  virtual void dump(ostream &os) = 0;
  void addInput(Net *n);
  void addOutput(Net *n);
  vector<Net *> *getInputs();
  Net* getOutput() const;
  int getNumInputs() const;
 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(string n); // contains the name of the design
  ~Design();
  string name(); // returns the name of the design

  /* See the provided skeleton code in the header file we provide */
  /*   for descriptions of the functions you need to write        */

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 the specified functionality in the header file. You must use a map to associate gate and net names with a pointer to the corresponding object.
2) Implement a Net class to implement the specified functionality of the provided header file.
   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 Gate base class to implement the specified functionality of the provided header file. 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). So dump should not show the delay. On the other hand, getDelay() should return the delay value and should return the default value of 1 if no delay was provided.
4) Implement derived And, Or, Nand, Nor, Xor, and Not classes from Gate class.
5) **Extra Credit Parser**: For extra credit, you can implement the parse function of the VlgParserMan class. You can consider appropriate helper functions like the one we prototyped. You can add your own or remove helper functions. This parse(…) 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 add them to the Design object
   c) Create gate objects (And, Or, Nand, Nor, Xor, Not) and add them to the Design object
   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
6) 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.
7) 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 the specified output file. 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.
8) Complete the main program in gsim.cpp that calls your parse function and if parsing is successful dumps the design to the output file specified on the command line.

5 Procedure

Perform the following.

1. Create a directory on your VM
   ```
   $ mkdir gsim1
   $ cd gsim1
   ```
2. Copy the sample images down to your local directory:
   ```
   $ wget http://ee.usc.edu/~redekopp/ee355/code/gatesim.tar
   ```
   This will download the following files:
   - `Makefile` – Builds your program
   - `verilog.lex & verilog.y` – Verilog tokenizer & grammar for auto parsing
   - `gates.h` – Gate base class. Define all the derived classes in this file too.
   - `net.h` – Net class
   - `design.h` – Design class
   - `vlg_parser.h` – Abstract header declaration for manual parsing (extra credit)…If you want to do the extra credit, derive a class named `vlg_parser_man.h & .cpp` to override the parse() function
   - `vlg_parser_auto.h & .cpp` – Complete parser for auto-parsing (no extra credit)
   - `gsim.cpp` – Where main() is defined and your program control starts
   - `mux21.v & mux21.sim` – Sample Verilog file of a 2-to-1 mux
   - `adder4.v & adder4.sim` – 4-bit Ripple carry adder Verilog file and sim
   - `test.v & test.sim` – Sample Verilog file 1
   - `test2.v & test2.sim`– Sample Verilog file 2
3. We have provided a "dumb" Makefile that will always compile all your .cpps (no smart compilation). However, it has two target rules:
   - `$ make selfparse` – Uses the vlg_parser_man.cpp where you define the VlgParserMan::parse() function [This is the extra credit option]
   - `$ make autoparse` – Assumes you will use the auto parser (no extra credit)
4. Complete all the classes and complete `main()` in **`gsim.cpp`**
5. Be sure you deallocate all memory that you are responsible for. Note: the parser doesn't know how you will use what it creates so it doesn't deallocate any memory. Thus, don't worry about solving errors that result from yyparse() or other function in the parser. But if any memory is "new'd" in your classes, it must be deallocated.
6. Complete the parsing and design creation functionality specified in the Requirements section. Be sure if you are implementing your own parser that it 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)
   - Missing commas in a gate or module declaration

You may assume all other syntax errors will not be present. You may also 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).

7. **Indicate both team members name as a comment at the top of gsim.cpp and also indicate if you implemented the SELF PARSER for Extra Credit**
8. Be sure you have added all .cpp, .h, Makefile, verilog.lex, and verilog.y files to your repository (using svn add and svn commit). It's up to you if you want to add .v files. Failure not to include these in your svn repository will result in point deductions.
9. Submit your completed code by checking in a tagged copy of your working code to your SVN archive using the command below…BE SURE TO REPLACE "01" with your group number.

   Note each of the command below should be on one single command line.

```
$ svn cp –m "gsim1 submission" svn://bits.usc.edu/ee355g01/trunk
                svn://bits.usc.edu/ee355g01/tags/gsim1submit
```

   Ensure your submission is correct by going up a directory (cd ..) and checking it back out to a new directory, compiling and running/testing your code again. **Failure to do this step may mean you did not submit correctly and may lose points for not doing so!**

```
$ svn checkout --username stu01 --password g01_XXX
               svn://bits.usc.edu/ee355g01/tags/gsim1submit gsim1test
$ cd gsim1test
$ make autoparse (or make selfparse)
$ ./gsim mux21.v mux21.out
```

   If you make a submission and then realize you need to update it, you will need to remove the old submission by running:

```
$ svn rm –m "remove gsim1 submission"
                svn://bits.usc.edu/ee355g01/tags/gsim1submit
```

Then re-perform the submission instructions above (i.e. svn cp, etc.)

Name: _____

Item	Outcome	Score	Max
Code submitted via SVN tag appropriately	Yes / No		1
Code Design • Appropriately derives specific gate types from the Gate base class • Each derived gate class implementing only the functions that provided different behavior for that gate type	 Yes / No Yes / No		 2 1
Design Class • Uses maps to associate names with Nets and Gates • Dynamically allocates Net and Gate objects and deallocates them at the appropriate time	 Yes / No Yes / No		 1 1
Net Class • Dynamically allocates vectors to store pointers to the driver and load gates and deallocates them at the appropriate time • Adds driver and load gates appropriately	 Yes / No Yes / No		 1 1
Gate Class(es) • Dynamically allocates vectors to store pointers to input nets and deallocates them at the appropriate time • Appropriate dump() function is implemented for each derived gate type • Adds input and output nets appropriately • Handles delays and default delays appropriately	 Yes / No Yes / No Yes / No Yes / No		 1 1 1 1
Dumping • Successfully dumps test.v • Successfully dumps test2.v • Successfully dumps mux21.v • Successfully dumps other instructor tests	 Yes / No Yes / No Yes / No Yes / No		 1 1 1 5
Subtotal			20
Extra credit parsing • Successfully parses test.v • Successfully parses test2.v • Correctly indicates line number of the first parse error, if any • Appropriately detects parse errors (input files that do not meet the specified grammar) [Comments]:	 Yes / No Yes / No Yes / No _____		 2 1 2 5
Late Deductions (-8 for 1 day / -16 for 2 days)			
Total			