Digital Design 2

Spring 2015

Project 1

Netlist Beautifier

Yehia Hesham - Ahmad Abdalla

The following is a description of the design of our application, the gate-level netlist beautifier and the testing strategy used.

Phase 1 :  Parsing and Extracting Data

The parsing of the verilog files was done manually using regular expressions on C++. Initially, we thought of using Perl or Python. In Perl, the APIs and libraries we found online were only available on Linux, not Windows. As for python, we found no libraries or APIs for verilog parsing at all. The parser goes through the text, ignoring comments, and extracting the declarations of inputs, outputs and wires, saving them in a struct, then going through each cell in the netlist, saving its type, name, inputs and output in the same struct. Finally, it does the assign instructions. The parser is designed to parse netlist cells with an undefined number of inputs, assuming that there is only one output for each gate/netlist cell.

Phase 2 : Creating a Directed Acyclic Graph

After the parser had retrieved all information (such as name, type, inputs, and output) for each wire, bus, cell, input, and output into a vector of structs ,we declare a variable of Graph type, which will have the nodes and edges stored in another vector named adj of type struct also (as for adjacency list ). Next, we add edges via the addedge function in the Graph class. Inside the class Graph, the addedge tries to capture the nodes and its edges and transfer the corresponding related information such as type, inputs pins, name, and output pin. By the end of adding these edges, the Toplogical sort is called also via a function inside the Graph class named Toplogicalsort.

 Phase 3 : Topological Sorting

As for the topological sorting, it is divided into two parts. The first is to ensure we iterate on all nodes, so that we don't leave one that for example have only 1 shared son with the rest of the graph. Which happens inside the Toplogicalsort function.

The second part implements the Depth First Search Algorithm (DFS). For each node in the first part we go through all its sons and in the way give to each node it corresponding level value by following these rules:

1) If it is an input, its level is set to one

2) If it is a wire, don't increment the level. Otherwise its level is the level of its father +1

3) If it happens that a node was given a value (in a certain iteration, that is we visited it before and gave it a level value) less than the new calculated value we assign the higher value. Because a node can have two inputs connected to it one from level 1 and the other is x (greater than 1). Thus we need to assign that node level x+1 instead of 2.

Note in order to implement this without error we had to put the wires at the bottom of the vector of adj, so that the initialized (wrong value) given to it at the beginning of the program is ignored.

Phase 4 : JSON file generation

A manual JSON file generator was written, since libraries designed for this purpose were not really available on windows. It creates a text file with .json extension, and writes to it using the fstream library. It was made sure that the format is that of JSON files.

Phase 5: Testing

The testing phase involved using the usage of the verilog files provided to us on blackboard. We would read the file, draw the gates by hand, assigning them, as well as the inputs and outputs, their corresponding levels, then check the output of our program (in the JSON file) and compare for errors. This was only possible on small modules, such as booth.g.v file, not on much larger modules. An additional verilog file (uploaded on Github) was created to test if the parsing of netlist cells taking more than 2 inputs, which was not covered in the provided samples, was correct. Here’s an example:

JSON format:

{

 "inputs": {

  "md[0]",

  "md[1]",

  "md[2]",

  "md[3]",

  "mr[0]",

  "mr[1]",

  "mr[2]",

  "mr[3]"

 },

 "outputs": [

  {

   "name":   "x[0]",

   "level":   2

  },

  {

   "name":   "x[1]",

   "level":   4

  },

  {

   "name":   "x[2]",

   "level":   4

  },

  {

   "name":   "x[3]",

   "level":   4

  },

  {

   "name":   "z[0]",

   "level":   2

  },

  {

   "name":   "z[1]",

   "level":   3

  },

  {

   "name":   "z[2]",

   "level":   3

  },

  {

   "name":   "z[3]",

   "level":   3

  },

 ]

 "modules": [

  {

   "type":   "INVX1",

   "name":   "\_2\_",

   "level":   2,

  },

  {

   "type":   "NOR2X1",

   "name":   "\_3\_",

   "level":   3,

  },

  {

   "type":   "XOR2X1",

   "name":   "\_4\_",

   "level":   2,

  },

  {

   "type":   "AND2X2",

   "name":   "\_5\_",

   "level":   3,

  },

  {

   "type":   "XOR2X1",

   "name":   "\_6\_",

   "level":   2,

  },

  {

   "type":   "INVX1",

   "name":   "\_7\_",

   "level":   2,

  },

  {

   "type":   "NOR2X1",

   "name":   "\_8\_",

   "level":   3,

  },

  {

   "type":   "XOR2X1",

   "name":   "\_9\_",

   "level":   2

  },

 ]

}

Hand-Drawn Diagram:

