CSCE 3304 DIGITAL DESIGN II

PROJECT 2: STATIC TIMING ANALYSIS

Yasmin ElDokany

Mazhar Ibrahim

Shahd Sherif

SECTIONS

ABSTRACT

The project presented in the following pages is a static timing analysis tool for Verilog HL. The tool, which is implemented in JavaScript reads the gate level netlist of a Verilog design in JSON format, and identifies the paths from each input/Qpin of a DFF to every output or Dpin of a DFF. The tool supports sequential and combinational designs and the methods for which we went about this are presented in the following pages.

COMPONENTS

The code is composed of a few different parts. There are three main data structures that govern the tool, and within each one we have all of the parsed and necessary information about the circuit. We believe in sophistication to the level of the source code itself, and have opted to use object literals instead of basic data structures. These object literals represent and contain all of the necessary information about every (potential) node in our dag. The three main structures are:

1. Inputs: This is an array of objects of size *InputCount.* The components of each member of the array are:
    I. *name:* this is the name of the input in English that the designer assigned in the beginning. It can be referenced by Inputs[i].name
    II. *bits:* This is the numerical identifier that JSON gives all of the inputs themselves. This is what we used in our algorithm to identify the paths. Referenced by: Inputs[i].bits

III. *type:* This field is ‘input’ or ‘output’ depending on the direction of the particular port. This may seem redundant as it is contained in the Inputs array, but it shows its purpose in more complex sequential scenarios.

2. Outputs: This is an array of objects of size *OutputCount.* The components of each member of the array are:
   IV. *name:* this is the name of the output in English that the designer assigned in the beginning. It can be referenced by Outputs[i].name
   V. *bits:* This is the numerical identifier that JSON gives all of the outputs themselves. This is what we used in our algorithm to identify the paths. Referenced by: Outputs[i].bits
   VI. *type:* This field is ‘input’ or ‘output’ depending on the direction of the particular port. This may seem redundant as it is contained in the Outputs array, but it shows its purpose in more complex sequential scenarios.

3. Celliez: This is an array of object literals that hold information about every gate in the circuit. The contents of one entry in *Celliez* are:
   I. *name:* This is the name of the cell in English that is specified by the synthesizer (NOR2X1, MUX, etc). Referenced by Celliez[i].name
   II. *connections:* This is a one dimensional array of all the connections to other ports stored in the numerical identifiers that

JSON assigns. This is referenced by Celliez[i].connections[i]. Note that a connection exists between Input/Cell[i] iff Input[i].bits == Celliez[i].connections[i = 0 → amount-1]

III. *amount:* This is how many of these connections there are.

IV. *output:* This is the outputs coming out of a cell. This information is necessary when knowing what cells are connected to each other.

METHODOLOGY

In this section, we discuss the main observations that we made that allowed for us to implement this tool, and the methodology. It is based on these, that all of our work depends. First, we will begin by providing the methodology of the parsing.

```
function main_func(){
    var bytes = document.getElementById("ourFile");
    var reader = new FileReader();

    reader.readAsText(bytes.files[0]);
    reader.onload = function()
    {

        var text = reader.result; //the letters of the file are
        in this variable
        var json = JSON.parse(text); //here we have the json
        objects we just need to index them correctly

        var ModuleNames = []; //the names of all our modules
        var HowManyModules = 0;//how many modules we have (might
        be useful, so why not lol)
        var meow = 0;
        for(property in json.modules) //here we're gonna know
        how many modules are in our top module, and their names
        will be stored in property
        {
            var traverse_ports = json.modules[property]["ports"];
            var Inputs = [];
            var Outputs = [];
            var InputCount = 0;
            var OutputCount = 0;
            for(portt in traverse_ports) //so here we are in the
```

Here, in our main function, we first extract the plain text from our JSON file, and send it to JSON.parse to empty the objects in our var json. We then loop on the modules, and ports and extract and fill our three main data structures. After we have acquired all of this information, we begin filling in our directed graph structure. Once the DAG is filled in, we use some of the .alg functions provided by the library to output it in an understandable manner to the user.

INTEGRATION

In this section, we talk about the front-end process of this project; namely the one regarding the user interaction. We have placed our main_func within an HTML file, and have attempted to provide a user friendly visual environment. We allow the user to upload their file using a button, and the results of our tool appear in a new window.

TESTING STRATEGIES

In this section, we talk about how we tested our design, and the inputs we send to it. We have provided a series of comprehensive Verilog modules. We synthesized these using *Yosys 0.6* as JSON files. We then would use mechanisms to identify the actual paths between inputs and outputs, and would test those against the output of our tool. Our modules range from very simple to a little bit complex. We have full adders, half adders, multiplexors, simple sequential circuits, a register similar to the one discussed in the lectures, and other simple combinational and sequential circuits.

Group Members

Mazhar Ibrahim (ID no. 900130306)

Shahd Sherif (ID no. 900140569)

Yasmin ElDokany (ID no. 900131538)

Input File Formats

File Name	Function	Format

Net Capacitances File	Lists capacitances for every pin in a gate. The name for the loc is stated between curly braces, and many miscellaneous capacitances are provided.
	
	The format is something like this:
	
	Pin (IN1):
	{
	Capacitance: x;
	Rise_capacitance: y;
	Fall_capacitance: z;
	Rise_capacitance_range: (y0, y1);
	
	}
		Parasitics Extraction File (.spef)

	(source: Bhasker J, Rakesh Chadha)	
Clocks Skew File	Contains information on Clock skew, buffers, and delay	Based on .sdc file
Timing Constraints File	Contains information about the clock period, the input delays and output delays	.sdc

Technical Specifications

Programming Language of choice: Javascript

Middleware/Libraries to be used:

Graphlib: A JavaScript library that provides data structures for undirected and directed multi-graphs along with algorithms that can be used with them. (reference: https://github.com/cpettitt/graphlib)