CSCE 337 – Digital Design II
Project 2 – Static Timing Analysis
Project Report

Mennatallah Yehia	900100570
Sara Kandil	900114861
Shaden Raouf	900111674
Sohaida Reyad	900120398

Introduction

Our Static Timing Analyzer is an application that accepts a gate-level netlist (GLN), a library file (liberty), net capacitance file, and timing constraints file, then it parses those files into internal data structures that will be integrated to perform analysis. The gate-level netlist is parsed into a directed acyclic graph (DAG) from which the different timing paths of the given logic will be deduced. Each path is then analyzed to obtained the path's arrival time, required time, and slack, which all will be saved in a report file.

The netlist will be parsed using Perl resulting in a .txt file that contains the inputs, outputs, wires, and gates. This parsed file is the picked up by the C++ application to continue the other operations in order to perform the STA.

Design & Implementation

The project is divided into the following modules:

1- Parsing all files (**parser** class):

  A. gate level netlist
  B. liberty file
  C. net capacitance file
  D. timing constraints file

2- Constructing DAG (**DAG** class)
3- Identify timing paths (**STA** class)
4- Perform STA (**STA** class)
5- Write report (**STA** class)

Parsing the Gate-Level Netlist:

The parsing was done with Perl library "Perl Verilog". The library includes classes for to model modules, cells, pints, ports, etc….

The library was installed on linux and the sample code was used as a starting point. Then, using the documentation of the library available, the code was tailored to meet our specific needs.

The code parses the gate level netlist file then writes the data in a format, that we have agreed upon, in a text file.

The format we agreed on as follows:

- [number of input ports n] followed by n lines. Each line contains the name of one input port and its size (bus or wire).
- [number of output ports m] followed by m lines. Each line contains the name of one output port and its size.

- [number of inout ports k] followed by k lines. Each line contains the name of one inout port and its size.
- [number of wires w] followed by w lines, each line contains the name of one wire.
- [number of assign statements a] followed by a lines. Each line contains the left hand side and the right hand side of one assignment statement, separated by a tab.
- Then a list of all gates, each gate in one line with the following data:
    - [type of gate]   [name of gate]   [number of inputs]   then list of inputs of the gate   then the output wire of the gate.

All fields are separated by tabs.

Parsring Gatelevel Netlist:

One of the "**parser**" class functions is readGLNFile, which opens and parses a .txt file that contains the output of the Perl Parser developed as outlined above. This function reads the file fields and fills the vector of **nodes**. Each node stores the following gate details:

- cell type (AND/NOR/etc..)
- instance (_1_, _2_, etc...)
- name (cell + instance)
- number of inputs
- output
- inputs
- gate delay
- output transition

Parsing Liberty:

A class called "Liberty" is used to internally parse the Liberty file, it uses two other classes to model the gates and their attributes: **Cell** and **Pin**.
Liberty file is parsed into an internal data structure that is composed of a map of cells, to map cell name to the contents of the cell.
Each cell has a name, and a map of pins.
Each pin has a capacitance, direction and 4 2D-tables to store the rise delay, fall delay, rise transition and fall transition.
Except in case of sequential cells: Latch and FFs, where the CLK pin has four extra 2D-tables: setup time rise, setup time fall, hold time rise and hold time fall.

The liberty class has the following public methods:
void parse(string filename);   to parse the liberty file passed as an argument

float getCapacitance(string gatename, int pinNumber); to retrieve the pin capacitance of particular pin in a particular cell, to be used in calculating total load capacitance of a cell.

float getOutputTransition(string cellname, float loadCapacitance, float inputTransition, int pinNumber) To retrieve the output Transition from the 2D table of the required pin, and use interpolation or extrapolation if necessary.
The function retrieves two values, one from transition rise, and one from transition fall, and returns the maximum.

float getDelay(string cellname, float loadCapacitance, float inputTransition, int pinNumber) To retrieve the gate delay from the 2D table of the required pin in the required cell, use interpolation or extrapolation if necessary. The function retrieves two values, one for rise delay, one for fall delay, and returns the maximum of the two values.

float getSetupTime(string gatename, float clkTransition, float pinTransition)
float getHoldTime(string gatename, float clkTransition, float pinTransition)
The two methods are used by class STA to get the setup time and hold time respectively of any sequential cell

Parsing Net Capacitance File:

Net Capacitance File Format:
<net_name>   <net_capacitance>

In the "**parser**" class, the function parseNet opens and parses a .txt file that contains the net capacitances. This function reads the file fields and fills the vector of **nets**. Each net stores the following net details:

- net name
- net capacitance

Parsing Timing Constraints File:
Timing Constraints File Format:
<num_inputs>

<input1_name>	<input1_delay>
<input2_name>	<input2_delay>

<num_outputs>

<output1_name>	<output1_delay>
<output2_name>	<output2_delay>

<clock_period>

In the “parser” class, the function parseTimeConst opens and parses a .txt file that contains the timing constraints. This function reads the file fields and fills the vector of constraints. Each constraint stores the following details:

- type (0=input; 1=output)
- name
- capacitance;
- delay

The parser also reads the clock period as the last field in the file and saves it internally in the parser.

Constructing DAG:

A class called “**DAG**” has been constructed in order to translate the vector of nodes into an adjacency matrix, which shows connections from input to gate, gate to gate, and gate to output in a 2D array (vector of vector). If one node is the input of another, a ‘1’ is saved in the cell that corresponds those two nodes. For example, if the output of an AND gate is the input of an OR gate, then the cell at row OR and column AND will have a ‘1’.

The adjacency matrix is then filled by iterating over each node and checking if its inputs are the outputs of other nodes, which indicates a connection between them.

Identify Timing Paths:

Depth First Search (DFS) has been applied on the DAG to identify the different timing paths:

- input -> output
- input -> FF
- FF -> FF
- FF -> output

Each path is saved in a vector, such that each element in this vector is another vector that has the list of indices of the nodes in the path identified. However, the result of the DFS is reversed order of nodes; hence, before analysis is done, each path is reversed.

Perform STA:

For each path in the vector of paths:

For each node in the path, we start analyzing it based on its type (primary i/p, gate, primary o/p).

1) If it is a primary input, we retrieve is delay from the vector of constraints and save it in its nodes.
2) If the node is a gate, we look at what came before it and what comes after it in the path.

a) if the next node is gate, we get its input pin capacitance and add it to the interconnect capacitance, which we retrieve from the nets vector.
b) If the next node is a primary output, we get the o/p capacitance (obtained from the timing constraints file) and continue as what we did in case a.

After we get the o/p capacitance and input transition, we retrieve the gate's output transition from the liberty file and save the result in the node's "outputTrans" (which stands for output transition). We also get the gate's delay and save it in "gateDelay" field of the node at hand.

3) If is it a primary output, we retrieve the previous node's output transition.

At each node

- if it is a gate, when we get gate delay we add it to the path delay
- if it is an input, we add its input transition (obtained from the timing constraints file) to the path delay.
- if it is an output, we add its previous node's output transition to the path delay.

All details of each path is saved in a "report" vector which saves each path's nodes and delays.

Arrival time = path delay.

Required time = clock period (from timing constraints) + skew - setup time (from liberty)

Slack = Arrival time + Required time

Write Report:

For each path there is a report line in the report vector. This line is then displayed node by node in a .txt file to show the node name, increment, and delay. Finally, for that path, the arrival, required, and slack times are recorded.

For the test file with the name "booth.v", the following report file was obtained:

```
                    REPORT # 0
--------------------------------------------------------------------------------------
Pin             Type            Incr            Path Delay
--------------------------------------------------------------------------------------
md[1]               INPUT           0       0       r
INVX1_2_/0          INVX1           0.0938915       0.0938915
NOR2X1_3_/1         NOR2X1          1.79622         1.89011
x[1]            OUTPUT      2.6496          4.53971         f
```

```
-----------------------------------------------------------------------------------------
Data Arrival Time                                          4.53971
Data Required Time                                         5.5
-----------------------------------------------------------------------------------------
Slack                                                      0.960293

*********************************************************************************************
*********************************************************************************************
```

This the analysis of only one path. The generated report for this test file shows 11 paths each analyzed individually but all written in the same .txt file.