Assignment: Implementing the STA AT/RAT Algorithm

In this assignment, you will implement the STA AT/RAT algorithm that was learnt in class using TCL. The algorithm calculates the arrival times and required times of the signals in the circuit and compares them against the constraints specified by the designer. This allows the designer to identify potential timing violations and optimize the circuit's performance.

The algorithm considers the delays associated with each element in the circuit, and computes the timing characteristics of the circuit, such as setup time and hold time. The original version of the STA AT/RAT algorithm is more complicated and involves many steps that are difficult to implement in practice, thus we will cover the algorithm with a few assumptions to ease the implementation.

Warmup Questions:

1. Explain what STA is and why it is important in digital circuit design.
2. Discuss the limitations of STA and situations where it may not be applicable.

Theoretical Warmup:
During the lecture you were taught the AT/RAT algorithm to calculate the slack and the critical paths of a circuit. In the following exercise, we would like to implement the mentioned algorithm and dive into its essence.

3. What is setup and hold violations, provide the equations and explain your answer.
4. Explain the concept of slack and how it is used in the STA algorithm.
5. What are the inputs and outputs of the algorithm.
6. Refer to the lecture, and to the homework assignment about STA (the AT/RAT algorithm) and make sure you understand how the algorithm works.

Implementation

To implement the STA AT/RAT algorithm, you will need to perform the following steps:

1. **Parsing the input and creating the timing graph:** this will be done manually like shown in class and recitation.

   In this exercise we will represent the graph using three matrices:

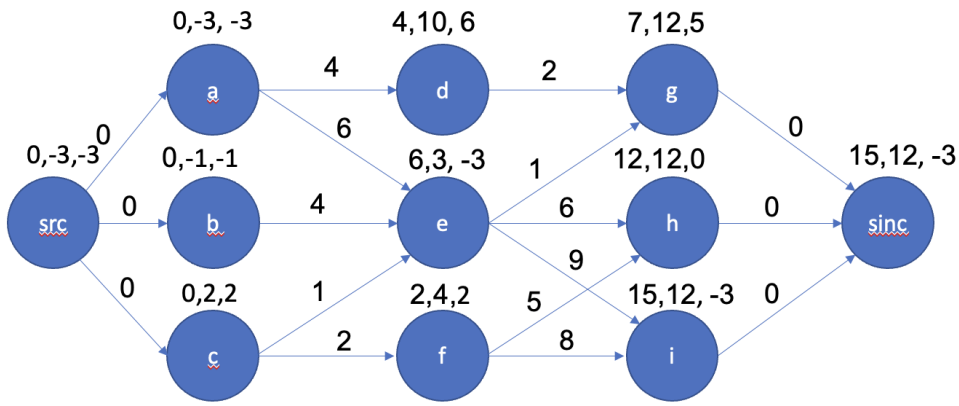
   - **The adjacency matrix** represents the connections between nodes in the graph. It's represented as a 2D array of 0s and 1s. A value of 1 in the matrix means that the corresponding nodes are connected, and a value of 0 means they're not.

   - **The weights matrix** represents the weight of each edge in the graph. It's also represented as a 2D array, with each element corresponding to the weight of the edge between the nodes represented by the row and column indices. It basically illustrates the timing delay going through a cell from a specific input pin to a specific output pin.

   - **The nodes list** is a list of nodes in the graph, represented as a 2D array with four columns: index, AT, RAT, and SLACK. Index is the node's index in the graph, AT is its arrival time, RAT is its required arrival time, and SLACK is the slack time of the node.

   Together, these matrices and nodes list represent a graph with weighted edges.

Here is an example:
Given this timing graph:



Given that src is in index 0, sinc is in index 10 and the rest of the nodes are sorted alphabetically between them, this will be the graph representation:

```
set adjacencyMatrix    {  {0 1 1 1 0 0 0 0 0 0 0}
                          {0 0 0 0 1 1 0 0 0 0 0}
                          {0 0 0 0 0 1 0 0 0 0 0}
                          {0 0 0 0 0 1 1 0 0 0 0}
                          {0 0 0 0 0 0 0 1 0 0 0}
                          {0 0 0 0 0 0 0 1 1 1 0}
                          {0 0 0 0 0 0 0 0 1 1 0}
                          {0 0 0 0 0 0 0 0 0 0 1}
                          {0 0 0 0 0 0 0 0 0 0 1}
                          {0 0 0 0 0 0 0 0 0 0 1}
                          {0 0 0 0 0 0 0 0 0 0 0} }
```

For example node a (index 1) is connected to node e (index 5) so
adjacencyMatrix[1][5] == 1

```
set weightsMatrix      {  {0 0 0 0 0 0 0 0 0 0 0}
                          {0 0 0 0 4 6 0 0 0 0 0}
                          {0 0 0 0 0 4 0 0 0 0 0}
                          {0 0 0 0 0 1 2 0 0 0 0}
                          {0 0 0 0 0 0 0 2 0 0 0}
                          {0 0 0 0 0 0 0 1 6 9 0}
                          {0 0 0 0 0 0 0 0 5 8 0}
                          {0 0 0 0 0 0 0 0 0 0 0}
                          {0 0 0 0 0 0 0 0 0 0 0}
                          {0 0 0 0 0 0 0 0 0 0 0}
                          {0 0 0 0 0 0 0 0 0 0 0} }
```

For example node a (index 1) is connected to node e (index 5) with weight of 6
adjacencyMatrix[1][5] == 6

```
set nodesList          { {0 0 0 0} {1 0 0 0} {2 0 0 0}
                         {3 0 0 0} {4 0 0 0} {5 0 0 0}
                         {6 0 0 0} {7 0 0 0} {8 0 0 0}
                         {9 0 0 0} {10 0 0 0} {11 0 0 0}
                         {12 0 0 0} }
```

node[0] = index, node[1] = AT, node[2] = RAT, node[3] = SLACK
All the nodes are initialized to 0.

2. **Calculating arrival times (AT):** After creating the timing graph, you need to calculate the arrival times of each node in the graph. Complete the function ***calcAT*** as you like, it should output an updated nodesList with the AT values in the right place.
3. **Calculating required times (RAT):** Once the arrival times have been calculated, the next step is to calculate the required times of each node in the graph. Complete the function ***calcRAT*** as you like, it should output an updated nodesList with the RAT values in the right place.
4. **Calculating slack:** After calculating the arrival and required times, you need to calculate the slack of each node in the graph. Complete the function ***calcSLACK*** as you like, you can use the ***calcAT/calcRAT*** functions you wrote. it should output an updated nodesList with the slack values in the right place, and report on any node with a negative slack in the following format:

   Node 0 has negative slack: -3
   Node 1 has negative slack: -3
   Node 2 has negative slack: -1
   Node 4 has negative slack: -3
   Node 6 has negative slack: -3
   Node 8 has negative slack: -3
   Node 11 has negative slack: -3
   Node 12 has negative slack: -3

- Given to you is a function called ***topologicalSort,*** it returns nodesList with the node's indices sorted topologically (you can read online on topological sort).
  you are advised to use this function (however you want), if you do, please explain.
- Attached 3 inputs and outputs examples, these examples do not cover all edge cases so you should build more on you own.
- You may use online TCL compilers, for example: https://www.jdoodle.com/execute-tcl-online/
- You may use any tool to write the code, including chatGPT, but be sure to understand the logic deeply.

Requirements

Your implementation should satisfy the following requirements:

- Your implementation should be able to calculate the arrival times, required times, and slack for each node in the graph.
- Your implementation should be able to detect timing violations and report them.
- Your implementation should be well-documented and easy to understand. Each function you create should include an explanation on what it does.
- Your implementation should be written in TCL.

Evaluation

Your implementation will be evaluated based on the following criteria:

- Correctness: Does your implementation correctly implement the STA algorithm?
- Completeness: Does your implementation satisfy all the requirements listed above?
- Efficiency: Is your implementation efficient in terms of time and memory usage?
- Documentation: Is your implementation well-documented and easy to understand?

Submission

Submit a zip file containing a pdf with you answers and a TCL file with you implementation:

STA_{YOUR GROUP NAME}.zip

STA_{YOUR GROUP NAME}.pdf

STA_{YOUR GROUP NAME}.tcl

For example:

If you are group 15

STA_advvlsi_15.zip

STA_ advvlsi_15.pdf

STA_ advvlsi_15.tcl