EE598/CEN598 VLSI Design Automation Mini Project #1

This mini-project consists of two parts, with an intermediate and a final deadline. You will be required to submit an intermediate submission for **Phase-1** by **February 18**. Your Phase-2 will reuse what you have done in Phase-1 however you can modify it to work better with Phase-2. The **final** submission **deadline is March 7**.

Ideally, you should write these programs using **standard** Python. You are NOT allowed to use any in-built graph libraries or in-built graph traversal/sorting algorithms. The goal of this project is for you to understand native data structures such as queues, dictionaries, np arrays, and lists and to understand underlying algorithms for graph traversal and delay computation. These are industry standard practices in EDA where these algorithms are usually designed from scratch for optimized implementation and not used from an existing library.
You are also NOT allowed to use ChatGPT generated code (detectors are more advanced than the generators). Please be mindful of this.

You may use simple Python libraries such as numpy, pandas, matplotlib, datastructure libraries, visualization libraries, etc. but NOT libraries or parts of the library that implement underlying algorithms such as sorting and traversal. One way to know what library you may use is to see if whatever function you are calling you can write it on your own.

You may develop your code on the platform of your choice **BUT** your submission must run using Python3.7 and must contain a requirements.txt which contains all python packages that can be installed using pip in a virtual environment. Please see instructions here:
https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/

I will be evaluating your submission on the EECAD servers. Please ensure your scripts run on that server. Python3.7.
python3.7 -m venv <name of venv>
source <name of venv>/bin/activate
pip3 install -r requirements.txt
python3.7 <your_main.py> <arguments>

The above commands should allow me to create a venv that has all packages required to run your code.

Your solution should be capable of handling a netlist with 100,000 gates. I would strongly encourage you to make sure it works for a range of circuits: small (e.g., c17), medium (e.g., c7552), and large (e.g., b15_C).

Submission Process

Please submit your code via Canvas as one .zip/.rar/.gz/.tgz file containing all the Python scripts source code. It should also contain the requirements.txt, and a README file. The README must list a brief (one-sentence), logically-organized description of each files within the archive and must list the commands to run your code. See below for format of the commands.

Description of the mini-project

In this project you will calculate the delay of a circuit by performing static timing analysis (STA) on it, similar to the topological traversal of a graph as discussed in class. Each node of the graph corresponds to each gate of the circuit, while each edge denotes a wire connection.

The circuit is described in *.bench format, an example of which is given below:

** Circuit c17.bench from ISCAS85 benchmark suite**
INPUT(n1)
INPUT(n2)
INPUT(n3)
INPUT(n6)
INPUT(n7)
OUTPUT(n22)
OUTPUT(n23)

n10 = NAND(n1, n3)
n11 = NAND(n3, n6)
n16 = NAND(n2, n11)
n19 = NAND(n11, n7)
n22 = NAND(n10, n16)
n23 = NAND(n16, n19)

The delay of each gate is a nonlinear function of its load capacitance and the input slew, and its calculation is summarized in the following figure.

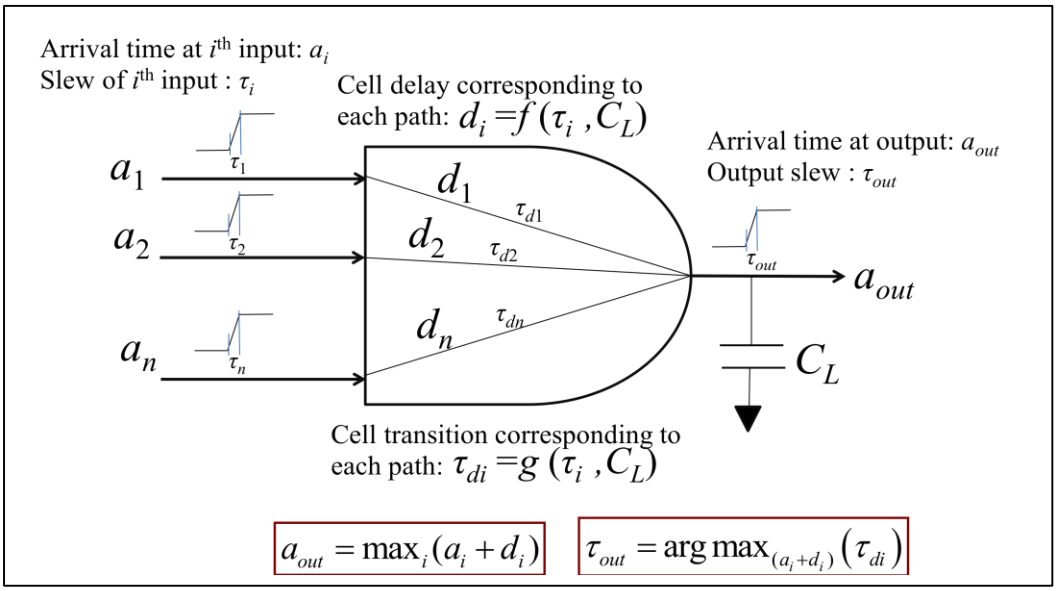

Figure 1. Delay (and output slew) of an *n*-input gate with a load capacitance of C_L

The functions, *f* and *g*, in Fig. 1, called the non-linear delay models (NLDMs), are typically provided by the foundry as look-up tables (LUTs). Each index of the LUT corresponds to the load capacitance (C_L) and input slew (τ_i) of a gate. The entries of the LUT represent either the delay (d) or the output slew values (τ_d) of the cell corresponding to (τ_i , C_L).

The NLDMs are stored in a *.lib file (called the "liberty" file). You are provided with one such file, "sample_NLDM.lib". This file is an overly simplified version of a real liberty file, and currently only includes delay/slew values corresponding to 2-input gates.

A snippet from sample_NLDM.lib with explanation of each part is provided below:

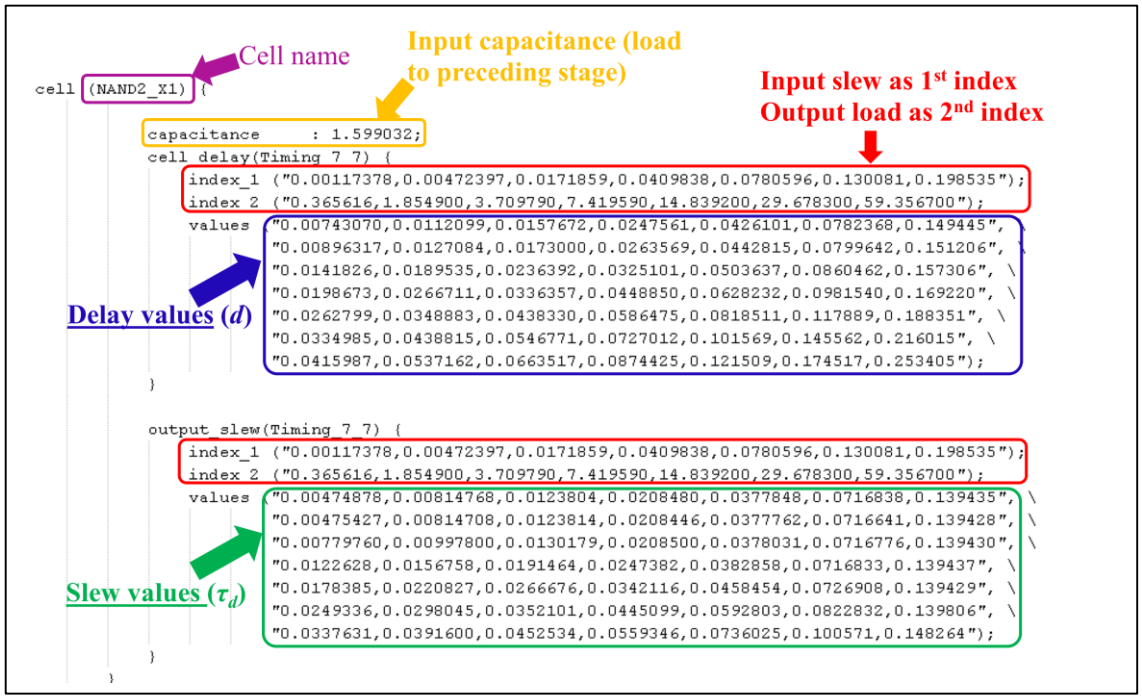

Figure 2. A snippet from the sample liberty file for a 2-input NAND gate.

To make your life easier, we are asking you to make the following simplifying assumptions:

- Since the liberty file only contains 1 or 2-input gates, for an *n*-input gate (with the same logic function), in a circuit under test, multiply the entry from the LUT with (*n*/2) to obtain the corresponding delay/slew values[1] if *n*>2.
- The input capacitance of the *n*-input gate may be assumed to remain unchanged from that of the corresponding 1or 2-input gate.
- No need to differentiate between rise and fall transitions and you can assume the same delay/slew LUTs for both. You can also assume the same delay/slew LUTs for all the input-output paths within the gate[2].

This mini-project is split into two phases:

Phase-1 (due February 19)

You have to parse the circuit as well as the liberty file, populate the necessary data structures, and produce the details of both the circuit and the NLDMs as necessary. You may develop a script called as parser that uses the keywords "read_ckt" and "read_nldm", respectively. For example, if the python script is called parser.py, then,

[1] In reality, the *n*-input gate would have its own LUT, but to make the assignment easier, we are asking you to make the simplifying assumption.
[2] In reality separate LUTs are provided for rise and fall transitions for each input-output path within the gate.

1) The command "`python3.7 parser_sta.py --read_ckt c17.bench`" should produce the details of the circuit as:

- Number of primary inputs and outputs
- Number and type of each gate,
- For each gate, print the number and type of its fanin and fanout gates

2) The commands for reading liberty file, sample_NLDM.lib, have two parts differentiated by two arguments (delay and slews) in the command:

- The command "`python3.7 parser_sta.py --delays --read_nldm sample_NLDM.lib`" should produce the cell delays of the NLDM, i.e., print the cells and their corresponding LUTs of delay in a tabular form in a separate file.
- The command "`python3.7 parser_sta.py --slews --read_nldm sample_NLDM.lib`" should produce the output slews of the NLDM, i.e., print the cells and their corresponding LUTs of output slews in a tabular form in a separate file.

Hint: Use python argparse and pathlib libraries for pointers to paths of the NLDM and circuit file

The specific details of the expected output of this phase are provided in **Appendix-1**.

Please refer **Appendix-4** for suggested data structure that your parsing can populate.

Your code will be tested on numerous other benchmark circuits some or all of which can be found in http://www.pld.ttu.ee/~maksim/benchmarks/iscas85/bench/.

Phase-2 (due March 10)

You have to perform STA on the circuit under test, find the slack at each gate output, and eventually, print the critical path. You can make the following assumptions:

- The arrival times at each primary input is 0, and input slew is 2 picoseconds (ps).
- The load capacitance of each gate is equal to the sum of capacitance of each of its fanout gates (i.e., ignore any wire capacitance).
- The load capacitance of the final stage of gates (which are simply connected to the primary outputs) is equal to four times the capacitance of an inverter from the liberty file.
- The required arrival time is 1.1 times the total circuit delay, and is the same at each primary output of the circuit.

For this phase, your expected output is outlined in **Appendix-2**.

For finding circuit delay, first perform the forward traversal of the circuit. At each gate, using the LUT, find the appropriate delays (and slews) of each path from its inputs to its output. The details of obtaining values corresponding to particular input slew and load cap for a path is provided in **Appendix-3**.
Next find the arrival time at the output of this gate by the "max" function as shown in Fig. 1. Repeat this until you reach the primary outputs. The maximum among the arrival times at all the primary outputs is the circuit delay.

For finding the slack at each gate, you need to perform a backward traversal of the circuit. Find the required arrival time at the output of each gate. The difference of the required arrival time and the actual arrival time (obtained from the forward traversal) is the slack at this gate.

Finally, find the critical path of the circuit based on the slack values. Start with the primary output with the minimum slack, and traverse backwards selecting the gates connected to this output with minimum slack, till you reach the primary input. If more than one primary output have the same value of the slack that is minimum, select any one randomly.

Example data structures which you can use for parsing the circuit netlist, the liberty file, and for performing STA, are provided in **Appendix-4.**

Appendices

Appendix-1: Sample expected output of Phase-1

The expected outputs are provided with respect to the following circuit (c17_dummy.bench), which has 5 primary inputs, 2 primary outputs, 4 two-input NAND gates, and 2 two-input NOR gates.

Note that you should index (or name) each gate by the name of its output wire.

** Circuit c17_dummy.bench adapted from c17.bench in ISCAS85 benchmark suite** # c17
# 5 inputs
# 2 outputs
# 0 inverter
# 6 gates (6 NANDs)

INPUT(1)
INPUT(2)
INPUT(3)
INPUT(6)
INPUT(7)

OUTPUT(22)
OUTPUT(23)

10 = NAND(1, 3)
11 = NAND(3, 6)
16 = NAND(2, 11)
19 = NAND(11, 7)
22 = NAND(10, 16)
23 = NAND(16, 19)

1) The command "`python3.7 parser.py --read_ckt c17.bench`" should produce a file called "ckt_details.txt", with each line as follows (contents within the dashed lines):

---

5 primary inputs
2 primary outputs
6 NAND gates
Fanout...
NAND-10: NAND-22
NAND-11: NAND-16, NAND-19
NAND-16: NAND-22, NAND-23
NAND-19: NAND-23
NAND-22: OUTPUT-22
NAND-23: OUTPUT-23
Fanin...
NAND-10: INPUT-1, INPUT-3
NAND-11: INPUT-3, INPUT-6
NAND-16: INPUT-2, NAND-11
NAND-19: INPUT-7, NAND-11
NAND-22: NAND-10, NAND-16
NAND-23: NAND-16, NAND-19

---

2) The command "`python3.7 parser.py --delays --read_nldm sample_NLDM.lib`" should produce a file, "delay_LUT.txt" with each line as follows (contents within the dashed lines):

---

cell: NAND2_X1
input slews: 0.00117378,0.00472397,0.0171859,0.0409838,0.0780596,0.130081,0.198535
load cap: 0.365616,1.854900,3.709790,7.419590,14.839200,29.678300,59.356700

delays:
0.00743070,0.0112099,0.0157672,0.0247561,0.0426101,0.0782368,0.149445;

0.00896317,0.0127084,0.0173000,0.0263569,0.0442815,0.0799642,0.151206;

0.0141826,0.0189535,0.0236392,0.0325101,0.0503637,0.0860462,0.157306;

0.0198673,0.0266711,0.0336357,0.0448850,0.0628232,0.0981540,0.169220;

0.0262799,0.0348883,0.0438330,0.0586475,0.0818511,0.117889,0.188351;

0.0334985,0.0438815,0.0546771,0.0727012,0.101569,0.145562,0.216015;

0.0415987,0.0537162,0.0663517,0.0874425,0.121509,0.174517,0.253405;

cell: NOR2_X1
input slews: 0.00117378,0.00472397,0.0171859,0.0409838,0.0780596,0.130081,0.198535
load cap: 0.365616,1.854900,3.709790,7.419590,14.839200,29.678300,59.356700

delays:
0.0136008,0.0161505,0.0205760,0.0292340,0.0462805,0.0801242,0.147594;

0.0142633,0.0167540,0.0211519,0.0298650,0.0470595,0.0810783,0.148696;

0.0202027,0.0226399,0.0267281,0.0350583,0.0519006,0.0857406,0.153365;

0.0282210,0.0316593,0.0372260,0.0469299,0.0634650,0.0965651,0.163573;

0.0381015,0.0422278,0.0490020,0.0610840,0.0815179,0.115113,0.181020;

0.0503320,0.0550648,0.0628653,0.0768972,0.101146,0.141009,0.207020;

0.0651817,0.0704989,0.0792584,0.0950330,0.122530,0.168656,0.242490;

… and so on for the other cells …

---

3) The command "`python3.7 parser.py --slews --read_nldm sample_NLDM.lib`" should produce a file, "slew_LUT.txt" with each line as follows (contents within the dashed lines):

---

cell: NAND2_X1
input slews: 0.00117378,0.00472397,0.0171859,0.0409838,0.0780596,0.130081,0.198535
load cap: 0.365616,1.854900,3.709790,7.419590,14.839200,29.678300,59.356700

slews:
0.00474878,0.00814768,0.0123804,0.0208480,0.0377848,0.0716838,0.139435;

0.00475427,0.00814708,0.0123814,0.0208446,0.0377762,0.0716641,0.139428;

0.00779760,0.00997800,0.0130179,0.0208500,0.0378031,0.0716776,0.139430;

0.0122628,0.0156758,0.0191464,0.0247382,0.0382858,0.0716833,0.139437;

0.0178385,0.0220827,0.0266676,0.0342116,0.0458454,0.0726908,0.139429;

0.0249336,0.0298045,0.0352101,0.0445099,0.0592803,0.0822832,0.139806;

0.0337631,0.0391600,0.0452534,0.0559346,0.0736025,0.100571,0.148264;

… and so on for the other cells …

Appendix-2: Sample expected output of Phase-2

The command "`python3.7 main_sta.py --read_ckt c17.bench --read_nldm sample_NLDM.lib`" should produce a file called "ckt_traversal.txt", with each line as follows:

Circuit delay: 62.5014ps

Gate slacks:
INPUT-1: 31.585 ps
INPUT-2: 25.5707 ps
INPUT-3: 6.25014 ps
INPUT-6: 6.25014 ps
INPUT-7: 31.058 ps
OUTPUT-22: 6.25014 ps
OUTPUT-23: 6.25014 ps
NAND-10: 31.585 ps
NAND-11: 6.25014 ps
NAND-16: 6.25014 ps
NAND-19: 11.8792 ps
NAND-22: 6.25014 ps
NAND-23: 6.25014 ps

Critical path:

INPUT-3, NAND-11, NAND-16, NAND-22, OUTPUT-22

Appendix-3: Reading from an LUT

The LUTs in the sample liberty file are all 7x7 tables with 1st index corresponding to the input slew and the 2nd index to the load capacitance. The units are provided within this liberty file and they are nanosecond for time and femtoFarad for capacitance.

The input slews for which each LUT is provided are: ("0.001,0.004,0.017,0.041,0.078,0.130,0.198") ns
The load caps for which each LUT is provided are: ("0.365,1.855,3.709,7.419,14.839,29.678,59.357") fF

However, suppose you encounter a path in the gate for which input slew (τ) and load cap (C) do not exactly match any of the values for which the LUT is characterized. In that case you will use a 2D interpolation. First you have to find the values of $C_{"}, C_{\$}, \tau_{"}, \tau_{\$}$ for which the values (delay/slew) are actually defined in the LUT as shown in Fig. 3, such that $C_{"} \leq C < C_{\$}$ and $\tau_{"} \leq \tau < \tau_{\$}$

	C_1	C_2
τ_1	v_{11}	v_{12}
τ_2	v_{21}	v_{22}

Figure 3. A part of the LUT of delay/slew, generalized by v_{ij} used to find the value at (τ , C).

Then the value, v, corresponding to (τ , C) for $C_1 \leq C < C_2$ and $\tau_1 \leq \tau < \tau_2$ is given by:

$$v = \frac{v_{11}\,(C_2 - C)(\tau_2 - \tau) + v_{12}(C - C_1)(\tau_2 - \tau) + v_{21}(C_2 - C)(\tau - \tau_1) + v_{22}(C - C_1)(\tau - \tau_1)}{(C_2 - C_1)(\tau_2 - \tau_1)}$$

Appendix-4: Example data structures for the project

The following is a suggested data structure you can use for your project. You can use class node and create objects of the class Node for every gate in the netlist and populate it. Alternatively, you can create a dictionary of dictionaries. I leave this to you. Below is only a suggestion.

Possible data structure to parse the netlist:

```
class Node:
    def __init__(self):
        self.name = ""
        self.outname = ""
        self.Cload = 0.0
        self.inputs = []  #list of handles to the fanin nodes of this node
        self.outputs =[]  #list of handles to the fanout nodes of this node
        self.Tau_in = []  # array/list of input slews (for all inputs to
                            the gate), to be used for STA
        self.inp_arrival = []  # array/list of input arrival times for
                                 input transitions (ignore rise or fall)
        self.outp_arrival = []  # array/list of output arrival times,
                                  outp_arrival = inp_arrival + cell_delay
        self.max_out_arrival = 0.0  # arrival time at the output of this
                                      gate using max on (inp_arrival +
                                      cell_delay)
        self.Tau_out = 0.0  # Resulting output slew

        # ... More functions/variables as required ...

# Example usage:
# node_instance = Node()
# node_instance.name = "example_gate"
# node_instance.Cload = 1.5
# node_instance.inputs = [node1, node2]  # assuming node1 and node2 are
instances of Node class
# node_instance.outputs = [node3, node4]  # assuming node3 and node4 are
instances of Node class
# ... Set other attributes as needed ...
```

Possible data structure to parse the liberty file:

```
class LUT:
    def __init__(self):
        self.Allgate_name = #all cells defined in the LUT
        self.All_delays = #2D numpy array delay LUTs for each cell
        self.All_slews =  #2D numpy array to store output slew LUTs for
                           each cell
        self.Cload_vals = #1D numpy array corresponds to the 2nd index in
                            the LUT
        self.Tau_in_vals = #1D numpy array corresponds to the 1st index in
                            the LUT

    def assign_arrays(self, NLDM_file):
        # define the arrays to be used during STA call later
        # also helps to simply assign the arrays so that a call to this
          function will fetch the arrays,
        # and you can easily print out the details of this NLDM

        # ...

# Example usage:
lut_instance = LUT()
lut_instance.assign_arrays("your_NLDM_file.txt")
```