REPORT ON OPENSTA

By: Prateek Kumar 2021181 Btech - ECE

1. INTRODUCTION

OpenSTA is an open-source tool used for static timing analysis of digital designs. It analyzes the timing of the design by considering the delays of various components and interconnections. OpenSTA supports industry-standard formats like Liberty (.lib) for library characterization and Synopsys Design Constraint (SDC) for input constraints.

The main theme of this report is to analyze the delay calculation of OpenSTA and to find out methods to use a custom delay calculation model in OpenSTA instead of using their own delay calculation model.

NOTE: This report is based on STA version 2.4.0

2. INITIATING THE SOFTWARE

The invoke command for OpenSTA is in the ‘app’ folder inside OpenSTA. To initialize the software we must have the corresponding netlist using the .v file, the liberty file, and the sdc constraints file .The commands can be written directly into the OpenSTa software or can be given to the software using a tcl file.

3. MAIN FUNCTION

The main function for the software is located in the ‘app’ folder by the name of Main.cc. This function is run when the invoke command is first called. Let's analyze this function.
The main function has various conditions:

a. If Argc=2 and the second argument is “-help” the code goes to the function showUsage which takes in the first argument as the invoke command or Argv[0] and the second argument as init_filename which is predefined.
b. Else If Argc=2 and the second argument is “-version” the code prints the value of STA_VERSION ie the current sta version being used (2.4.0).
c. Else the function sets Argc=1 and passes 1, Argv, and tclAppInit as parameters to the Tcl_Main

If no errors are found during tcl initialization then the function initStaApp runs

- This function calls another function initSta() which is located at OpenSTA/search/Sta.cc:223

- There it initializes a func initElapsedTime() which is stored in MachineLinux.cc and stores the current time in the variable &elapsed_begin_time_
- Next the initSta() function initializes the TimingRole class located at OpenSTA/liberty/TimingRole.cc:53
- Various new objects are created while initializing Timingrole:

wire _,combinational_,tristate_enable_,tristate_disable_,reg_clk_q_,reg_set_clr_,latch_en_q_,latch_d_q_,sdf_iopath_,setup_,hold_,recovery_,removal_,width_,period_,skew_,nochange_,output_setup_,output_hold_,gated_clk_hold_,latch_setup_,latch_hold_,data_check_setup_,data_check_hold_,non_seq_setup_,non_seq_hold_

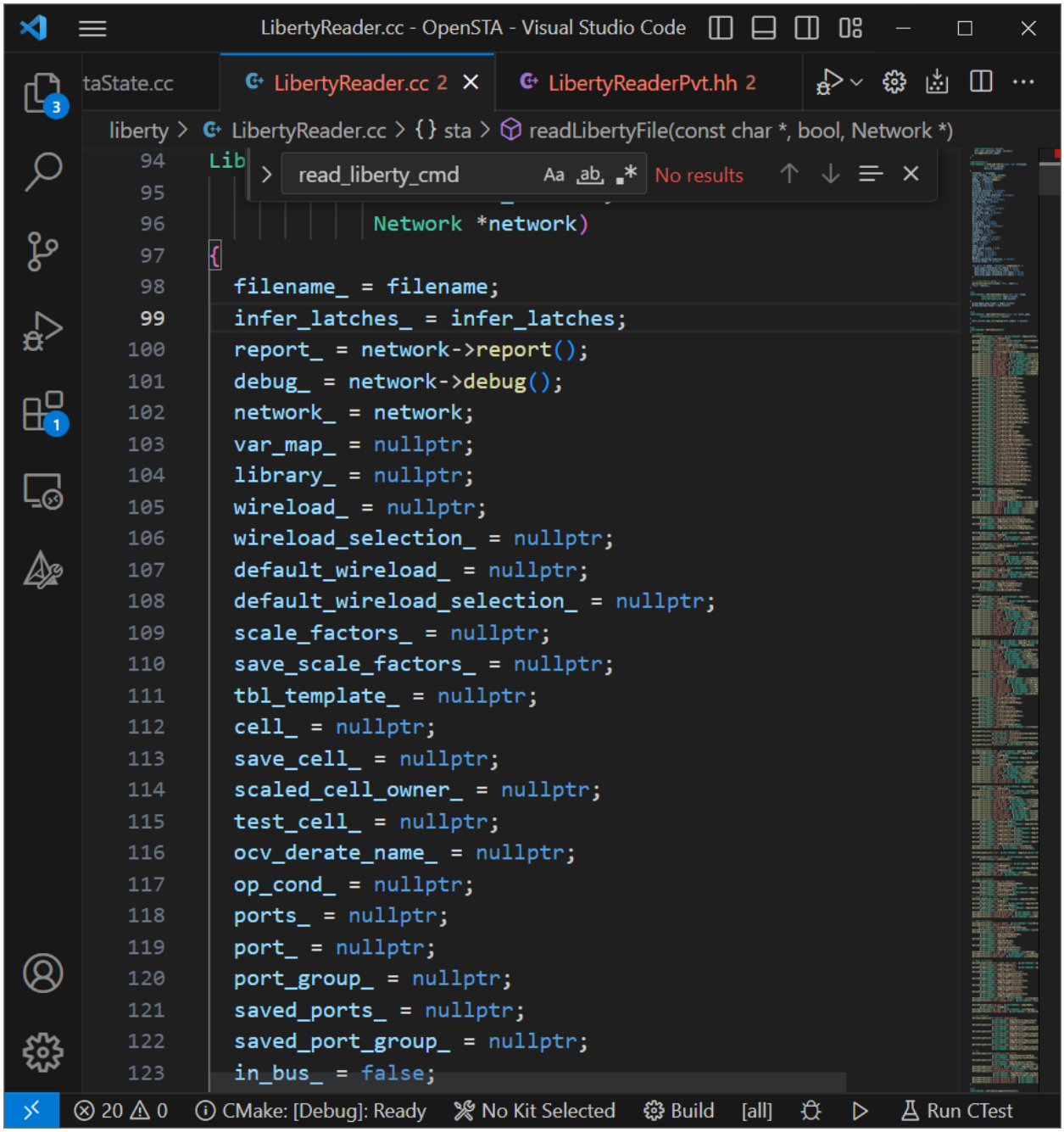
***We have basically initiated the constraints which are there in sta.TimingRoles in OpenSTA define the roles or responsibilities of specific objects or elements in the design regarding timing analysis, and they play a crucial role in determining and enforcing timing constraints during the analysis process.

- Next, we initiate the PortDirection by coming back to the initsta() func in sta.cc Takes me to the address OpenSTA/network/PortDirection.cc:32
- Here new objects are initialized of type PortDirection like input_,output_,tristate_,bidirect_,internal_,ground_,power_ and unknown_ These are for checking the type of Port in the circuit.
- Now the next function is run in initsta() : initTmpStrings(); Location: OpenSTA/util/StringUtil.cc:163
- Used to initialize 100 temporary strings of size 100 each. while initializing sta
- Next function in initsta(): initLiberty(). Location:OpenSTA/liberty/Liberty.cc:48
- Initialising liberty takes me the initiation of TimingArcSet which is located in OpenSTA/liberty/TimingArc.cc:511
- Initial configuration of timing arc is being set here
- Now the initSta initiates the delay constants located at OpenSTA/graph/DelayFloat.cc:33
- Initialises min and max delay constant values to 0
- Next function : registerDelayCalcs ()
- Location : OpenSTA/dcalc/DelayCalc.cc:35

4. PROCESSING A LIBERTY FILE INTO OPENSTA

The data from the liberty file is taken and stored in openSTA in its own network format. The steps of reading this network from a .lib file are:

1. TCL MAIN initiated from the main.cc file. This initiates the TCL interpreter
2. The tcl interpreter takes the source tcl file as input from the sourceTclFile from StaMain.cc which takes the file from Main.cc
3. The tcl interpreter starts evaluating the file and uses the file StaAppTCL_wrap.cxx which is a mapping between tcl functions and opensta functions.

4. In the wrapper file the _wrap_read_liberty_cmd is initiated
5. The read_liberty wrapper takes us to the function read_liberty_cmd
6. Now we move to the read_liberty function in sta.cc which takes me to the libertyreader.cc file
7. Here firstly a builder is initialised which then defines the visitor attributes are defined in Opensta. The visitors are the columns in the liberty file which is to be read by opensta.
8. The reader initialized in libertyreader then initiates a function readLibertyFile(const char *filename, bool infer_latches, Network *network)
9. 

initially declared as a null pointer, and then their values are written into them using the

parseLibertyFile(filename, this, report_) function in the libertyparser.cc file this finally connects the opensta network to the liberty file. Each cell is defined by using the liberty files

10.
```
139
140   for (auto rf_index : RiseFall::rangeIndex()) {
141     have_input_threshold_[rf_index] = false;
142     have_output_threshold_[rf_index] = false;
143     have_slew_lower_threshold_[rf_index] = false;
144     have_slew_upper_threshold_[rf_index] = false;
145   }
146
147   //::LibertyParse_debug = 1;
148   parseLibertyFile(filename, this, report_);
149   return library_;
150 }
151
```

5. PROCESSING A VERILOG FILE INTO OPENSTA

1. TCL MAIN initiated from the main.cc file. This initiates the tcl interpreter
2. The tcl interpreter takes the source tcl file as input from the sourcetclfile from stamain.cc which takes the file from main.cc
3. The tcl interpreter starts evaluating the file and uses the file StaAppTCL_wrap.cxx which is a mapping between tcl functions and opensta functions.
4. In the wrapper file the _wrap_read_verilog_cmd is initiated
5. Initially it checks if the file provided exists and no error is generated.
6. Finally the wrapper maps the verilog read function to read_verilog_cmd () function which is also in the same wrapper file. It is given the name of the verilog file as input
7. the read_verilog_cmd function provides an interface for reading a Verilog file and constructing the corresponding network in OpenSTA. It ensures that OpenSTA is properly initialized and performs the necessary setup before calling the appropriate function to read the Verilog file.
8. If the network reader object is valid ie opensta has been properly initialised then the network is read onto the opensta app using the readVerilogFile(filename,network) function
9. If a netlist is already there then clear the previous data.
10. Now the function readverilogfile is initialised which takes me to the verilogreader.cc. New verilogreader is initialised which takes in the input as network.
11. This verilog reader sets up the network for the open sta
12. Next the read command is used to read the verilog file provided and read it to the network details in opensta

13. Here an init function is invoked which reads the file by taking in input as the filename.
14. The function then looks for the "Verilog" library in the network. If the library is not found, it creates a new library with the name "Verilog".init function sets up the necessary state and data structures before parsing and reading the Verilog file.

6. LINKING A VERILOG FILE TO THE LIBERTY NETWORK

1. The command given in the tcl interpreter is analyzed by the inbuilt functions of the interpreter and then it comes to the StaAppTCL_wrap.cxx . Here it is mapped to the function _wrap_link_design_cmd
2. This function initially checks for any errors and then initiates the link_design_cmd function. With the input as the name of the top_cell which is in this case TOP
3. This function in turn calls the linkdesign function which is in the sta.cc file.
4. This function firstly clears all the previous data if present and then the linknetwork function is initialized in the concretenetwork.cc function. We move to the link_verilog_network functionin verilogreader.cc and then linknetwork function is initialized
5. Finds the module for the top cell in the library file provided. Once it is found then it finds the corresponding Verilog module. It then makes an instance of the top cell in the network. The code then iterates over itself to cover each port of the module and bind the ports of the module to the ports of the library.

7. REPORT GENERATION

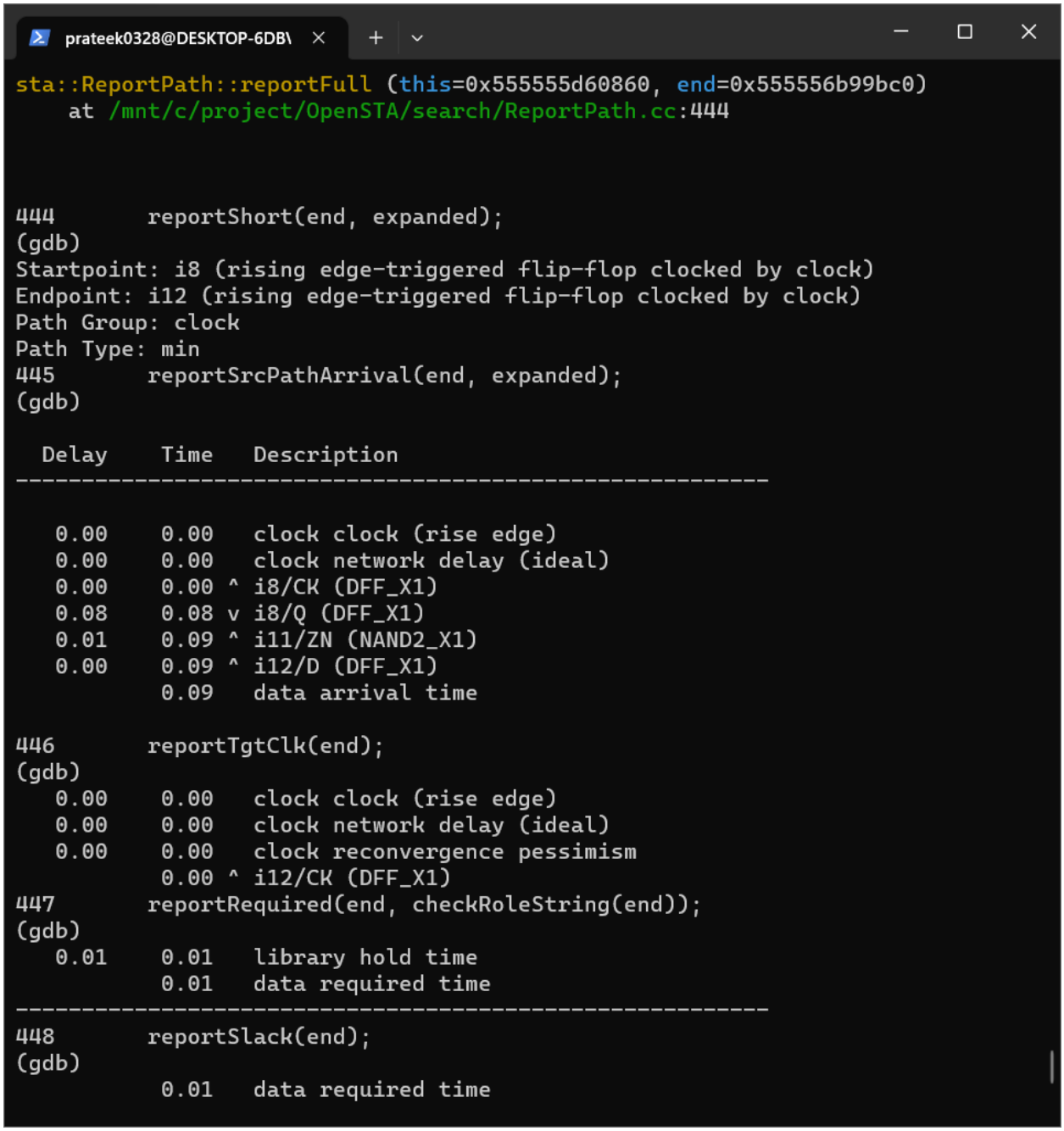
The command for the generation of the report comes from the Search.tcl file and the cpp function that is called by it using the StaAppTCL_wrap.cxx is report_path_end2

```
5404    report_path_end2(PathEnd *end,
5405                     PathEnd *prev_end)
5406    {
5407      Sta::sta()->reportPathEnd(end, prev_end);
5408    }
```

- Takes me to sta.cc which has the function reportPathEnd .this function takes me to reportpath.cc where it is defined

- For a full report the reportpath.cc file calls the following functions

```
425  ReportPath::reportShort(const PathEndCheck *end)
426  {
427    PathExpanded expanded(end->path(), this);
428    reportShort(end, expanded);
429  }
430
431  void
432  ReportPath::reportShort(const PathEndCheck *end,
433              PathExpanded &expanded)
434  {
435    reportStartpoint(end, expanded);
436    reportEndpoint(end);
437    reportGroup(end);
438  }
439
440  void
441  ReportPath::reportFull(const PathEndCheck *end)
442  {
443    PathExpanded expanded(end->path(), this);
444    reportShort(end, expanded);
445    reportSrcPathArrival(end, expanded);
446    reportTgtClk(end);
447    reportRequired(end, checkRoleString(end));
448    reportSlack(end);
449  }
450
```



- The report full function is used to get the full report for a particular instruction given in the tcl file.

8. ARRIVAL INFORMATION FINDING PROCEDURE

OVERVIEW OF THE PROCESS THAT TAKES PLACE:

1. Make graph: This step involves creating a graph data structure that represents the design under consideration. The graph likely represents the logical and physical elements of the design, such as cells, pins, nets, and their interconnections.

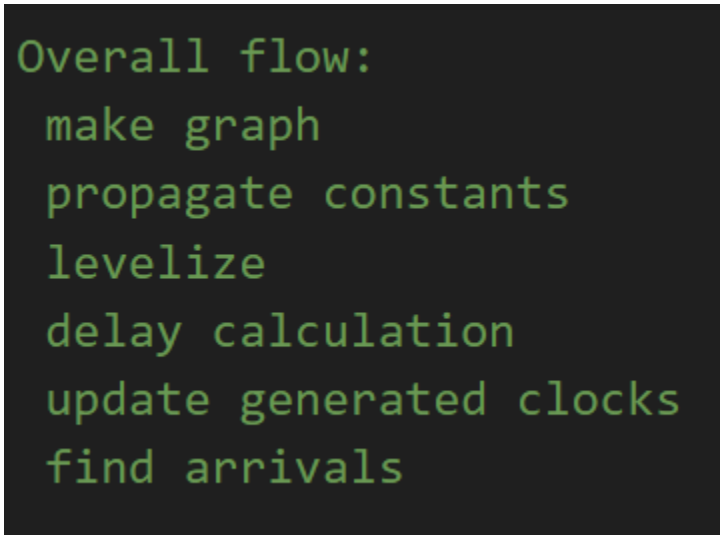
2. Propagate constants: This step involves propagating constant values throughout the graph. Constants are fixed values that do not change during circuit operation, and propagating them can help simplify subsequent analysis and optimizations.

3. Levelize: Levelization is a process in which the graph is partitioned into levels based on the logical and physical dependencies of its elements. This step organizes the graph in a way that allows for efficient analysis and optimization algorithms to be applied.

4. Delay calculation: This step involves calculating the delay of paths in the design. The delay represents the time it takes for a signal to propagate from a source to a destination through various components in the circuit. Delay calculation is crucial for determining the timing behavior of the design and identifying critical paths.

5. Update generated clocks: This step involves updating information related to generated clocks in the design. Generated clocks are derived clocks that are generated from primary clocks or other clock sources. Updating their information may involve adjusting their timing characteristics or accounting for clock gating and other clock-related optimizations.

6. Find arrivals: This step involves finding arrival times for paths in the design. Arrival times represent the times at which signals reach their destinations. By determining the arrival times, it becomes possible to analyze the timing behavior of the design and identify violations or meet specific timing requirements.



THE PROCEDURE :

- A procedure is defined in search.tcl file named find_timing_paths_cmd in line 107. This procedure takes in the cmd and the name of the variable storing the command line arguments. The procedure is used to find the timing paths and check for errors and call the functions to compute the desired timing paths

- This tcl procedure handles the report command passed to it and tells the Opensta app which timing path to analyze. It then calls the appropriate function for calculating the information related to that timing path.
- This tcl command, in the end, calls the _wrap_find_path_ends function present in the sta_tcl swig wrapper file. In turn, this function calls the find_path_ends function in the same file after selecting the appropriate inputs for the function.
- The find_path_ends function takes in a bunch of inputs

```
5349    PathEndSeq
5350    find_path_ends(ExceptionFrom *from,
5351                   ExceptionThruSeq *thrus,
5352                   ExceptionTo *to,
5353                   bool unconstrained,
5354                   Corner *corner,
5355                   const MinMaxAll *delay_min_max,
5356                   int group_count,
5357                   int endpoint_count,
5358                   bool unique_pins,
5359                   float slack_min,
5360                   float slack_max,
5361                   bool sort_by_slack,
5362                   PathGroupNameSet *groups,
5363                   bool setup,
5364                   bool hold,
5365                   bool recovery,
5366                   bool removal,
5367                   bool clk_gating_setup,
5368                   bool clk_gating_hold)
```

- This function first calls the cmdLinkedNetwork() function to check whether the network has been linked or not. If the Network is not linked then further execution is not necessary.
- If the network is linked and ready the function then calls the findPathEnds() function in the sta.cc file with all the necessary parameters.
- This function initially calls the searchPreamble() function. This function has 4 major functions:
    - findDelays(): Initially calls the delayCalcPreamble() function which calls the ensureClkNetwork() function. This function is ensuring the Network is already present and linked before finding the Delays.Going into the ensureClkNetwork() function :
        - Has two functions: ensureLevelized() and ensureClkNetwork()
            - ensureLevelized(): before the graph is levelized (graph partitioned into levels based on the logical and physical dependencies of its elements also identifies the critical path)we need to first make a graph, ensure that the graph is Sdc annotated and propagate the constants. ensureGraph() function ensures that the design graph is constructed and available for subsequent operations. The design graph represents the logical and physical connectivity of the design, including cells, pins, nets, and their relationships. If the

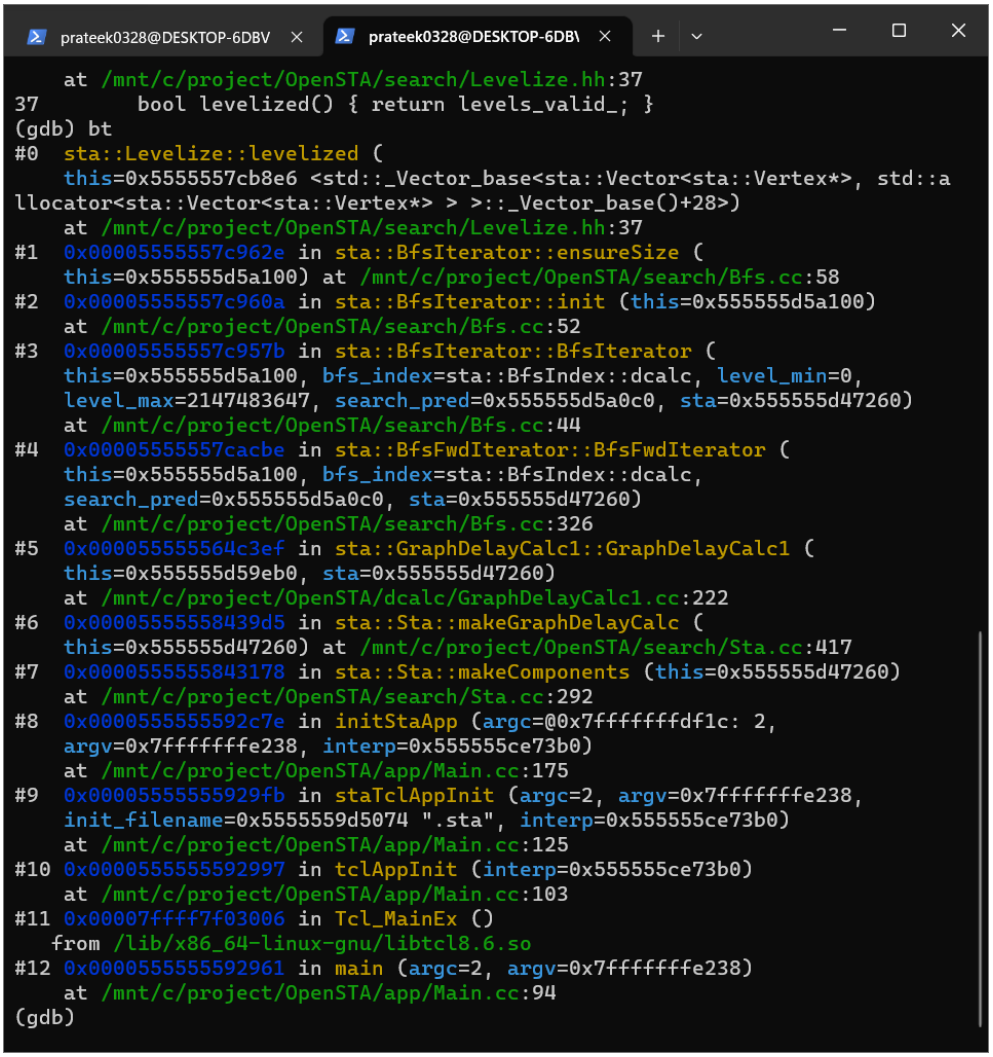
graph has not been constructed, this step ensures its creation. Then the ensureGraphSdcAnnotated() This function ensures that the design graph is annotated with information from the SDC (Synopsys Design Constraints) file. The SDC file contains timing and constraint information for the design, such as input delays, clock frequencies, and setup/hold constraints. Annotating the graph with this information is necessary for accurate delay calculation.

Note: All output pins are considered constrained because they may be downstream from a set_min/max_delay -from that does not have a set_output_delay.

Then the ensureConstantsPropogated() function is called and if the constants are not already propagated then this step ensures that constant propagation is performed on the design graph. Constant propagation analyzes the logic paths in the design to identify and propagate constant values (such as 0 or 1) through the logic gates. This information helps determine the values of signals and enables subsequent optimizations and accurate delay calculation.

ensureLevelized function ensures that the design graph is levelized, meaning that the graph's cells and pins are organized into levels or stages based on their logical dependencies. Levelization is essential for performing timing analysis, as it establishes the order in which the cells and pins are evaluated to calculate timing delays. The levelization process identifies the critical paths and helps determine the signal arrival times and required times at each endpoint. This levelization is done in

levelize.cc function which uses DFS to do this.



levelized() is called earlier using this stack trace

- The ensureClkNetwork() function is called from clknetwork.cc
- findClkPins(): If the clock pins are not valid, this function is called to identify and validate the clock pins in the design. The findClkPins function is responsible for traversing the design hierarchy and identifying the primary and generated clock pins.
-

- Now we are done with delayCalcPreamble(). Next in findDelays() we have graph_delay_calc_->findDelays(levelize_->maxLevel()) By passing the maximum level of levelization as the input parameter, the findDelays function calculates the delays in the design up to that level. This ensures that the delay calculations are performed only on the relevant portions of the design, improving efficiency.
- Checks first that arc_delay_calculation is valid. Now if the delays are not seeded ie the initial value for the delays is not written yet then the

Multidriver nets are found and the rootslews are seeded(GraphDelayCalc1::seedRootSlews() 551). Else:

- Firstly it ensures that the BFS queue is deep enough for the max logic level. Now if incremental mode is on then firstly seedInvalidDelay() function is called.:
  - This function iterates over all the vertices which are marked as invalid and seeds their slew time if they are root vertex or if they are not root vertex then enqueues non-root vertices with non-latch predecessors for further processing during delay calculation
- Now, a visitor object is initialized and visitparallel function is invoked to perform parallel visitation of vertices up to a certain level.
  -
- Now timing check edge delay is done by iterating over all the elements in the invalid_check_edge container and using the function findCheckEdgeDelays(). Similarly latch edge delay is calculated for all the latch element connections by iterating over them and using the function findLatchEdgeDelays(). **This ends the findDelays function**

- updateGeneratedClks(): The basic function of the updateGeneratedClks() module in OpenSTA is to update the generated clocks in the design. Generated clocks are clocks that are derived from other clocks in the design hierarchy. This module ensures that the waveform of each generated clock is valid by generating it based on the waveform of its master clock.
- Sdc -> searchPreamble() : it has two functions ensureClkHpinDisables() and ensureClkGroupExclusions(). In conclusion, this module prepares clock configurations by disabling clock sink pins and applying exclusions to the clock group.
  - ensureClkHpinDisables(): This function enables all clock sink pins (hierarchical pins) to be disabled. Disabling the clock sink pins means that the sinks are not considered as sequential elements for clock path analysis. This step is necessary to accurately model the clock tree structure and avoid incorrect timing analysis results.
  - ensureClkGroupExclusions(): This function ensures that the necessary exclusions or exceptions are applied to clock groups. Clock groups are used to define relationships between related clocks in the design, such as synchronous or asynchronous clock domains. By specifying exclusions, certain paths between clocks within the same group can be ignored during timing analysis, improving efficiency and reducing false violations.
- search_->deleteFilteredArrivals(): This function call deletes any previously filtered or pruned arrival times in the search_ object. It is possible that during previous path searches or filtering operations, certain arrival times were excluded or marked as invalid. By deleting the filtered arrivals, the path search starts with a

clean slate, considering all valid arrival times for subsequent analysis.

```
From/thrus/to are used to make a filter exception.  If the last
search used a filter arrival/required times were only found for a
subset of the paths.  Delete the paths that have a filter
exception state.
```

- Now we are done with setting up the environment for searching for the path. Now we begin actually searching the path using the findPathEnds() function in the search.cc file.
- We first get the findFilteredArrival() function in findPathEnds().:
    - This function initially deletes the results from the previous time the findPathEnds() was called. Uses the function deletePathGroups()
    - Next it checks all the from thrus and tos to be valid… no value should be empty or invalid.
    - Filtering or finding all arrivals: Depending on the presence of valid from and thrus objects, the function determines whether to perform filtered arrivals or find all arrivals. If either the from the object has associated pins or instances, or the thrus object is not null, the function proceeds with filtered arrivals.
        - findAllArrivals(): Takes in the boolean input thru_latches which determines if thru_latches are present or not
        - Firstly it initializes the visitor object which is used to track and update the arrival time of all the signals
        - Then we move into a loop to calculate the arrival time until arrival time at every latch stop changing
        - Inside the loop firstly we enqueue all the pending latches whose arrival time is yet to be calculated
        - Invokes the findArrivals1 function to calculate the arrival times of signals at the specified maximum level. This function performs the actual calculation of arrival times using the topological levelization of the design.
            - findArrivalsSeed(): performs initial setup and seeding of the arrival time calculation
                - genclks_->ensureInsertionDelays(): Ensures that the insertion delays of generated clocks are properly accounted for.
                - Then it clears any information about arrival time or required arrival time that was present earlier.
                - Calls the seedArrivals(): handles the seeding of arrival times for different types of vertices based on their properties and connectivity within the design. It ensures that the necessary arrival time information is collected and processed during the arrival time calculation.
                - Next all the invalid arrivals are seeded.
            - visitParallel(): function in OpenSTA's BfsIterator class is responsible for performing a parallel visit of vertices in a

        - breadth-first search (BFS) manner up to a specified level returns the number of arrivals to be counted
- After the delay is calculated
- Check if the recovery/removal checks are enabled in the SDC (Synopsys Design Constraints) and update the recovery and removal variables accordingly. If the checks are not enabled, both recovery and removal are set to false.
- checks if the gated clock checks are enabled in the SDC and update the clk_gating_setup and clk_gating_hold variables accordingly. If the checks are not enabled, both clk_gating_setup and clk_gating_hold are set to false.
- It calls the makePathGroups() function which returns a path group created according to the given constraints
- Then it calls the ensureDownstreamClockPins() function which uses backward BFS and marks all the pins in the level above to have downstream pins and the found downstream pins flag to be true if it founds a downstream pin.
- Next the makePathEnds function generates the path ends for different groups based on the provided constraints and options. It handles sorting, resizing, and the generation of the path ends for unconstrained paths, providing a sequence of path ends ready for further analysis and reporting. Finally these path ends are returned
- Next these path end values are sent to the tcl interpreter and there it calls and does the reporting part

9. DELAY CALCULATION IN OPENSTA

In OpenSTA, the calculation of delays involves several steps and functions working together. Here is an overview of how the delay calculation process works using the provided functions:

1. **Path Generation**: The process begins with the `findPathEnds` function. It takes various parameters such as `from`, `thrus`, `to`, and other constraints to generate the path ends using the `makePathGroups` and `makePathEnds` functions. These path ends represent the starting and ending points of the paths for which delays need to be calculated.

2. **Arrival Time Calculation**: The `findFilteredArrivals` function is called within `findPathEnds` to calculate the arrival times for the generated paths. This function filters the arrivals based on the given exceptions and determines the arrival times using the `findAllArrivals` and `findArrivals1` functions.

3. **Required Time Calculation**: Once the arrival times are determined, the required times need to be calculated. The `required_iter_->visitParallel` function is called within the `findArrivals1` function to calculate the required times for the paths up to the specified level. This process involves visiting the vertices in a breadth-first manner using multiple threads to efficiently traverse the design graph.

4. **Delay Calculation**: With both the arrival times and required times calculated, the delays can be determined. The delay calculation typically involves subtracting the arrival time from the required time to obtain the path delay. This calculation is performed within the `visit` function of the `VertexVisitor` class, which is passed as an argument to the `visitParallel` function.

5. **Reporting and Analysis**: Once the delays are calculated, OpenSTA provides various reporting and analysis functions to present the delay information. These functions may include generating delay reports, identifying critical paths, analyzing setup/hold violations, and reporting clock-to-clock maximum cycle warnings, as seen in the `sdc_->reportClkToClkMaxCycleWarnings` function call within `findPathEnds`.

Overall, the delay calculation process in OpenSTA involves generating path ends, calculating arrival times, calculating required times, performing delay calculations, and providing analysis and reporting functionalities. By utilizing these functions and the underlying design data, OpenSTA enables accurate and comprehensive delay analysis of digital designs.

NOTE: The delay calculation procedure happens inside the GraphDelayCalc.cc and GraphDelayCalc1.cc class. The function SearchPreamble calls the findDelays function in these classes and the delay calculation procedure is done here.

10. CUSTOM DELAY MODEL

I have found 2 ways through which we can implement our own delay calculation model inside OpenSTA.

10.1 Inbuilt API

It is possible to use a custom delay calculation function in OpenSTA to calculate the delay. OpenSTA provides a flexible architecture allowing users to define their delay calculation algorithms by implementing a new class derived from the ArcDelayCalc class.
**We get to know about this api from the StaApi.text file which is in the doc directory.
Let's see how we can use a custom delay calculation function in OpenSTA:

1. **Create a New Delay Calculator Class:** Define a new class that is derived from the ArcDelayCalc class. This can be created by either modifying the current class or creating a new class by using this interface .This new class should implement the virtual functions defined in ArcDelayCalc to calculate the gate delay, driver slew, load delays, and load slews for a given timing arc. The specific logic for your custom delay calculation should be written in these functions.
2. **Register the New Delay Calculator:** Register your custom delay calculator with OpenSTA using the registerDelayCalc function. This function allows OpenSTA to recognize your new delay calculator and make it available for use.

3. **Set the Custom Delay Calculator:** Use the setArcDelayCalc function in OpenSTA to set your custom delay calculator as the active delay calculator. This function ensures that OpenSTA uses your custom delay calculation logic during the timing analysis.
4. **Compile and Link:** After implementing the custom delay calculator, compile your code and link it with OpenSTA to include the new delay calculator in the STA executable.

The interface for the new model class is stored in the include\sta folder inside the ArcDelayCalc.h file

```
46  class ArcDelayCalc : public StaState
47  {
48  public:
49    explicit ArcDelayCalc(StaState *sta);
50    virtual ~ArcDelayCalc() {}
51    virtual ArcDelayCalc *copy() = 0;
52
53    // Find the parasitic for drvr_pin that is acceptable to the delay
54    // calculator by probing parasitics_.
55    virtual Parasitic *findParasitic(const Pin *drvr_pin,
56                                     const RiseFall *rf,
57                                     const DcalcAnalysisPt *dcalc_ap) = 0;
58    virtual ReducedParasiticType reducedParasiticType() const = 0;
59    // Find the wire delays and slews for an input port without a driving cell.
60    // This call primarily initializes the load delay/slew iterator.
61    virtual void inputPortDelay(const Pin *port_pin,
62                                float in_slew,
63                                const RiseFall *rf,
64                                const Parasitic *parasitic,
65                                const DcalcAnalysisPt *dcalc_ap) = 0;
66
67    // Find the delay and slew for arc driving drvr_pin.
68    virtual void gateDelay(const LibertyCell *drvr_cell,
```

```
66
67    // Find the delay and slew for arc driving drvr_pin.
68    virtual void gateDelay(const LibertyCell *drvr_cell,
69        const TimingArc *arc,
70        const Slew &in_slew,
71        // Pass in load_cap or drvr_parasitic.
72        float load_cap,
73        const Parasitic *drvr_parasitic,
74        float related_out_cap,
75        const Pvt *pvt,
76        const DcalcAnalysisPt *dcalc_ap,
77        // Return values.
78        ArcDelay &gate_delay,
79        Slew &drvr_slew) = 0;
80    // Find the wire delay and load slew of a load pin.
81    // Called after inputPortDelay or gateDelay.
82    virtual void loadDelay(const Pin *load_pin,
83        // Return values.
84        ArcDelay &wire_delay,
85        Slew &load_slew) = 0;
86    virtual void setMultiDrvrSlewFactor(float factor) = 0;
87    // Ceff for parasitics with pi models.
88    virtual float ceff(const LibertyCell *drvr_cell,
89        const TimingArc *arc,
```

```
90              const Slew &in_slew,
91              float load_cap,
92              const Parasitic *drvr_parasitic,
93              float related_out_cap,
94              const Pvt *pvt,
95              const DcalcAnalysisPt *dcalc_ap) = 0;
96
97      // Find the delay for a timing check arc given the arc's
98      // from/clock, to/data slews and related output pin parasitic.
99      virtual void checkDelay(const LibertyCell *drvr_cell,
100             const TimingArc *arc,
101             const Slew &from_slew,
102             const Slew &to_slew,
103             float related_out_cap,
104             const Pvt *pvt,
105             const DcalcAnalysisPt *dcalc_ap,
106             // Return values.
107             ArcDelay &margin) = 0;
108     // Report delay and slew calculation.
109     virtual string reportGateDelay(const LibertyCell *drvr_cell,
110                                    const TimingArc *arc,
111                                    const Slew &in_slew,
112                                    // Pass in load_cap or drvr_parasitic.
113                                    float load_cap,
```

```
                                        float load_cap,
                                        const Parasitic *drvr_parasitic,
                                        float related_out_cap,
                                        const Pvt *pvt,
                                        const DcalcAnalysisPt *dcalc_ap,
                                        int digits) = 0;
    // Report timing check delay calculation.
    virtual string reportCheckDelay(const LibertyCell *cell,
                                    const TimingArc *arc,
                                    const Slew &from_slew,
                                    const char *from_slew_annotation,
                                    const Slew &to_slew,
                                    float related_out_cap,
                                    const Pvt *pvt,
                                    const DcalcAnalysisPt *dcalc_ap,
                                    int digits) = 0;
    virtual void finishDrvrPin() = 0;

  protected:
    GateTimingModel *gateModel(const TimingArc *arc,
                               const DcalcAnalysisPt *dcalc_ap) const;
    CheckTimingModel *checkModel(const TimingArc *arc,
                                 const DcalcAnalysisPt *dcalc_ap) const;
    TimingModel *model(const TimingArc *arc,
```

```
    TimingModel *model(const TimingArc *arc,
                       const DcalcAnalysisPt *dcalc_ap) const;
  };

  } // namespace

```

Register Dealy calculator function is in the dcalc directory in the Delaycalc.cc file

```
27  namespace sta {
28
29  typedef Map<const char*, MakeArcDelayCalc, CharPtrLess> DelayCalcMap;
30
31  static DelayCalcMap *delay_calcs = nullptr;
32
33  void
34  registerDelayCalcs()
35  {
36    registerDelayCalc("unit", makeUnitDelayCalc);
37    registerDelayCalc("lumped_cap", makeLumpedCapDelayCalc);
38    registerDelayCalc("slew_degrade", makeSlewDegradeDelayCalc);
39    registerDelayCalc("dmp_ceff_elmore", makeDmpCeffElmoreDelayCalc);
40    registerDelayCalc("dmp_ceff_two_pole", makeDmpCeffTwoPoleDelayCalc);
41    registerDelayCalc("arnoldi", makeArnoldiDelayCalc);
42  }
43
44  void
45  registerDelayCalc(const char *name,
46                    MakeArcDelayCalc maker)
47  {
48    if (delay_calcs == nullptr)
49      delay_calcs = new DelayCalcMap;
```

```
44  void
45  registerDelayCalc(const char *name,
46                    MakeArcDelayCalc maker)
47  {
48    if (delay_calcs == nullptr)
49      delay_calcs = new DelayCalcMap;
50    (*delay_calcs)[name] = maker;
51  }
```

The setArcDelayCalc function is defined in sta.cc file in the search directory.

```
3325  void
3326  Sta::setArcDelayCalc(const char *delay_calc_name)
3327  {
3328    delete arc_delay_calc_;
3329    arc_delay_calc_ = makeDelayCalc(delay_calc_name, sta_);
3330    // Update pointers to arc_delay_calc.
3331    updateComponentsState();
3332    graph_delay_calc_->delaysInvalid();
3333    search_->arrivalsInvalid();
3334  }
```

10.2 FUNCTION IN PLACE OF DELAY CALCULATION FUNCTION

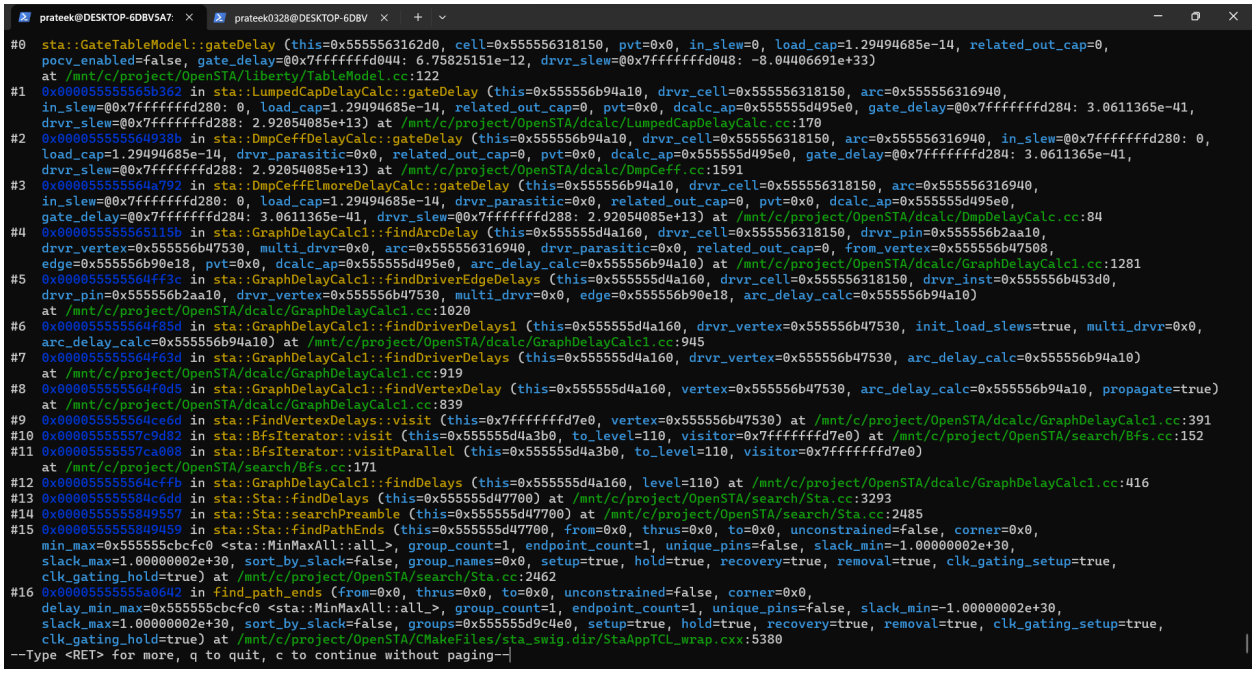
The delay calculation takes place in openSTA by using a delay calculation model. All the delay calculation models are derived from the main parent class ArcDelayCalc where the abstract class is defined to decide how each of the models will be designed.
This is the hierarchy of the net delay models that are used in OpenSTA

```
34  // Delay calculator class hierarchy.
35  //  ArcDelayCalc
36  //   UnitDelayCalc
37  //   LumpedCapDelayCalc
38  //    RCDelayCalc
39  //     SlewDegradeDelayCalc
40  //     DmpCeffDelayCalc
41  //      DmpCeffElmoreDelayCalc
42  //      DmpCeffTwoPoleDelayCalc
43  //     ArnoldiDelayCalc
44
```

But we need to find the cell delay. For that, I found the TimingDelay model in OpenSTA.
This model is defined in The file TableModel.cc in the liberty folder.

- This file has the definition for the GateTable Model. This model is derived from the ArcDelay class and hence it defines its virtual functions.
- The function which is used to calculate the gate delay is gateDelay() defined from line 97.

- The hierarchy before the function call is shown below:



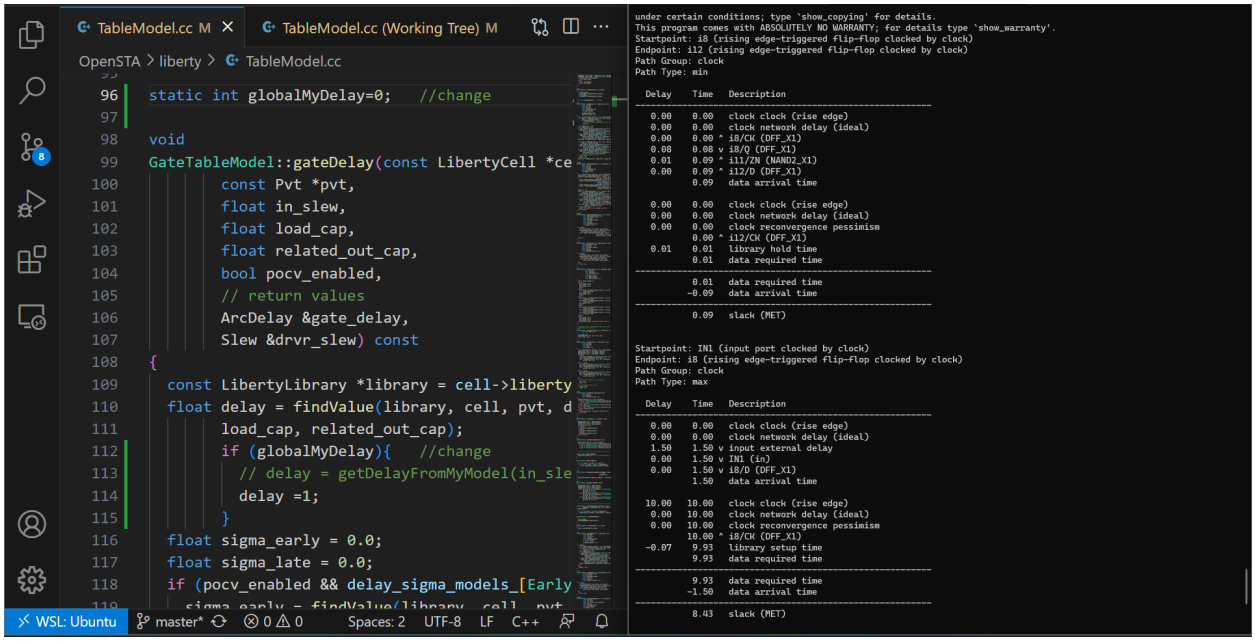
- Now inside this function the findValue function is used to pick up the delay from the liberty files using the slew and the load capacitance

```
 96  void
 97  GateTableModel::gateDelay(const LibertyCell *cell,
 98                            const Pvt *pvt,
 99                            float in_slew,
100                            float load_cap,
101                            float related_out_cap,
102                            bool pocv_enabled,
103                            // return values
104                            ArcDelay &gate_delay,
105                            Slew &drvr_slew) const
106  {
107    const LibertyLibrary *library = cell->libertyLibrary();
108    float delay = findValue(library, cell, pvt, delay_model_, in_slew,
109                            load_cap, related_out_cap);
110    float sigma_early = 0.0;
111    float sigma_late = 0.0;
112    if (pocv_enabled && delay_sigma_models_[EarlyLate::earlyIndex()])
113      sigma_early = findValue(library, cell, pvt,
114                              delay_sigma_models_[EarlyLate::earlyIndex()],
115                              in_slew, load_cap, related_out_cap);
116    if (pocv_enabled && delay_sigma_models_[EarlyLate::lateIndex()])
117      sigma_late = findValue(library, cell, pvt,
118                             delay_sigma_models_[EarlyLate::lateIndex()],
```

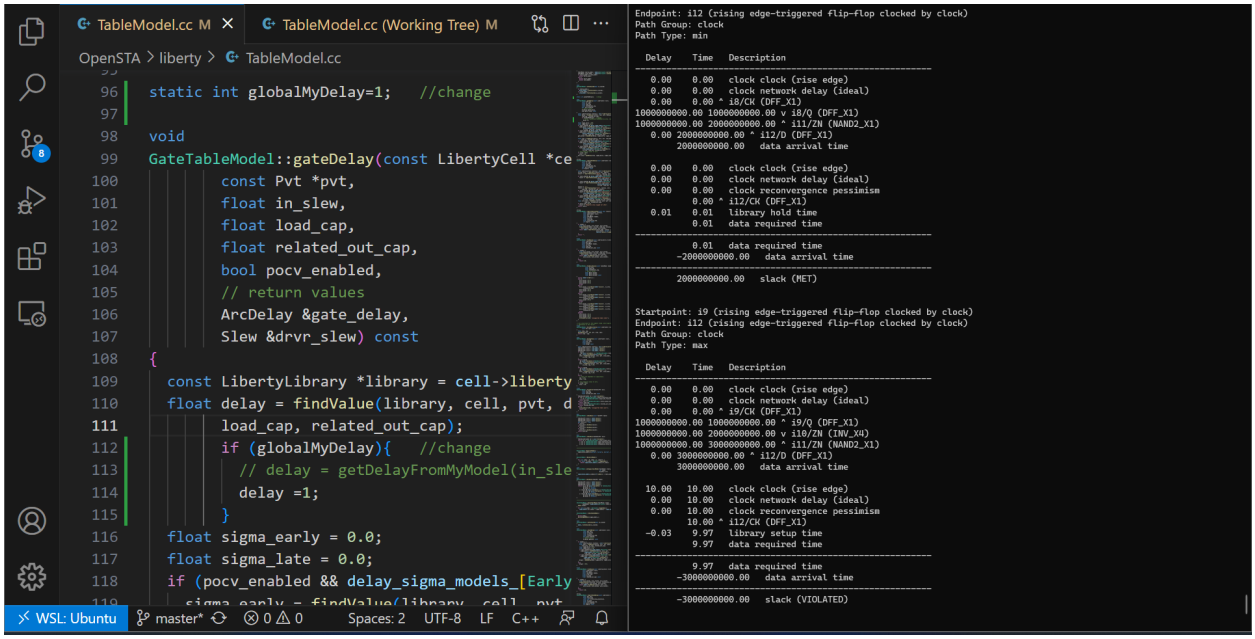
```
113     sigma_early = findValue(library, cell, pvt,
114                             delay_sigma_models_[EarlyLate::earlyIndex()],
115                             in_slew, load_cap, related_out_cap);
116   if (pocv_enabled && delay_sigma_models_[EarlyLate::lateIndex()])
117     sigma_late = findValue(library, cell, pvt,
118                            delay_sigma_models_[EarlyLate::lateIndex()],
119                            in_slew, load_cap, related_out_cap);
120   gate_delay = makeDelay(delay, sigma_early, sigma_late);
121
122   float slew = findValue(library, cell, pvt, slew_model_, in_slew,
123                          load_cap, related_out_cap);
124   if (pocv_enabled && slew_sigma_models_[EarlyLate::earlyIndex()])
125     sigma_early = findValue(library, cell, pvt,
126                             slew_sigma_models_[EarlyLate::earlyIndex()],
127                             in_slew, load_cap, related_out_cap);
128   if (pocv_enabled && slew_sigma_models_[EarlyLate::lateIndex()])
129     sigma_late = findValue(library, cell, pvt,
130                            slew_sigma_models_[EarlyLate::lateIndex()],
131                            in_slew, load_cap, related_out_cap);
132   // Clip negative slews to zero.
133   if (slew < 0.0)
134     slew = 0.0;
135   drvr_slew = makeDelay(slew, sigma_early, sigma_late);
136 }
```

- We make changes at this location to set our own model.We can have a flag variable inside this class and use that flag variable to call the CustomDelayFunction when the flag is high. The custom model will take slew and loadCap as input and can show the output when that custom model is used.

- We test the code first with the flag as zero



- Next when the flag is high



Here the commented line represents calling the model we design.