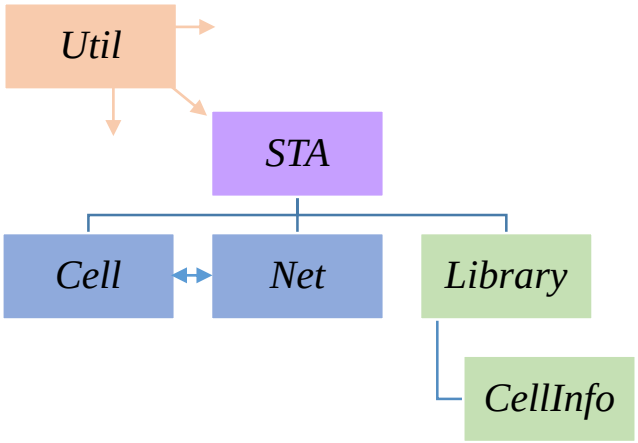
Lab 2 Static Timing Analysis

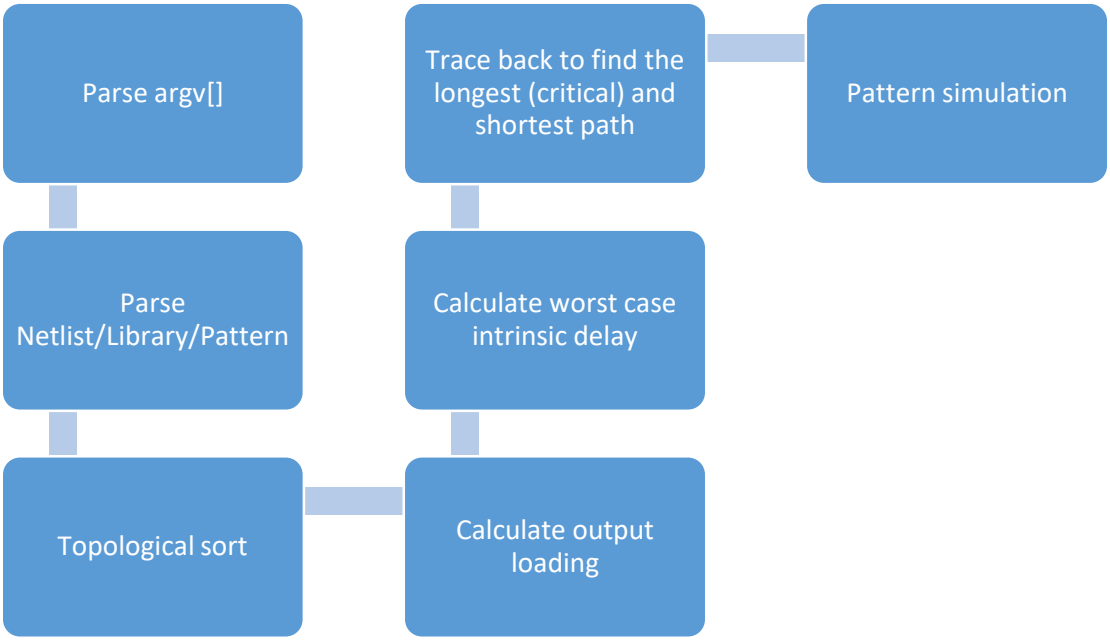
312510224 林煜睿

1. Data Structure



- ***Util.h:***
  Declaration and definition of some constant that repeatedly used, such as regular expression ***regex***, enumerate type ***enum***. There are also define some ***inline*** function to change ***string*** into corresponding ***enum*** types. The ***truthTable*** inside it is the input and output relation of corresponding gate.
- ***Cell.h***
  Information of each cell (gate), including its type, input/output transition type, output loading, (intrinsic) delay. In order to execute STA, it also records the arrival time of input signals, input/output nets, and logic values. Using ***indegree*** to do topological sort, ***prevCell*** to trace critical path.
- ***Net.h***
  Information of the input/output cells of current net, also records the type ***NetType.***
- ***Library.h***
  Information inside *test_lib.lib.*
- ***CellInfo.h***
  Lookup-tables of corresponding ***CellType***.

2. Algorithm Flow



- ***Parse argv[]***

  $./312510224\ \ <netlistName>.v\ -l\ test\_lib.lib\ -i\ \ <patternName>.pat$

  According to $-l$ and $-i$, we can get the position of $test\_lib.lib$ and $<patternName>.pat$, and store them into ***string*** *libraryPath* and ***string*** *patternPath*, respectively. Since $<netlistName>.v$ does not have command-line flags like previous case, we then check its file extension and store it into ***string*** *netlistPath*.

- ***Parse Netlist/Library/Pattern***

  In *void verilogParser(),* I first use *string removeComment(const string&)* inside list, then parse it into data structure *Cell.h* and *Net.h*.

  In *void libraryParser()*, I use regular expression to shorten my code and parse it into data structure defined in *Library.h* and *CellInfo.h*.

  In *void patternParser(),* I store different patterns inside $vector<vector<string>>$.

- ***Topological sort***

  In order to do STA, we need an appropriate order to traverse whole netlist without conflicts. Therefore, topological sort is an excellent ordering that can fulfill our desire. Sort the cells by their *indegree,* which is defined as number of *wire* type net connect to the current cell.

- ***Calculate output loading***

  Traverse netlist (not necessarily in topological order), sum up the input pin capacitance of cells that are inside the output net of current cell. The input pin capacitance can be decided by *vector<double> CellInfo.pinCap.*

- ***Calculate worst case intrinsic delay***

  Traverse netlist in topological order, according to the output load of current cell, we can decide its *riseDelay* and *fallDelay* through table-lookup. We then compare the value of *riseDelay* and *fallDelay,* the larger one is the worst case, set *cell.delay* and *cell.worstCaseValue* to corresponding value and use table-lookup to decide its worst case transition time. Besides, we store the information of the previous cell that results in worst case value of current cell.
- ***Trace back***

  Trough each output net, select the largest/smallest propagation delay path and trace back by *net.inputCell.prevCell* we get from previous step, then get the critical (longest) path and shortest path or current netlist.
- ***Pattern simulation***
- Set pattern value to each input net, and traverse netlist in topological order. Similar as step ***Calculate worst case intrinsic delay,*** but this time we need to consider the influence of controlling value of each gate, which is stored inside *cell.controllingValue.* The core logic can be descripted in following 3 condition:

  i. **One** input of current cell is its controlling value

  ii. **Both** inputs of current cell are its controlling value

  iii. **Neither** two inputs of current cell are its controlling value

  In condition *i*, directly choose the input signal with controlling value as critical path. In condition *ii,* we need to choose the input signal with smaller arrival time since it can decide the output of current cell when it arrive. In condition *iii,* we need to choose the input signal with larger arrival time since neither of them can decide output of current cell by themselves.

3. Optimization

- *const regex*

I declare all regular expression into constant type rather than create it repeatedly in loops. Since in the very beginning I used quite large amount of *regex_replace* and *regex_search,* it fastened my code by around 400% in large netlist and pattern.

- *string removeComment(const string&)*

At first, I use regular expression to remove odd and diverse comments in 3 steps:

i. *Remove //... without */ after it*

ii. *Remove all /* ... */ comments*

iii. *Remove all // ... comments*

After confirm the comment only has following 2 types:

i. *// ..., and there's no /* or */ after it*

ii. */**/ ，and there's no // between /* and */*

Besides, comment will not tear apart a single command like this:

Ex: *output /*fdsg*//*htrghr*/ N22, N/*;*/23; ...*

I re-design the *removeComment* function into line-by-line processing, since each *regex_replace* needs to traverse whole code and search complex expression takes too many time, especially when netlist file is quite large. Such optimization fastened my code by around 35% in large netlist and pattern.

- *void assignPattern()* and *void dumpGateInfo(ostringstream &, const vector<Cell*> &))*

At first, I directly create *ofstream* inside function *void assignPattern()* and call *dumpGateInfo* by passing current *ofstream&* to it each time I finish an iteration (pattern). I found that its quite slow when I repeatedly execute file I/O actions. Therefore, I first store results inside an *ostringstream* each time I complete a pattern, and write whole results into *ofstream* in one action. Such optimization fastened my code by around 25% in large netlist and pattern.