# Physical Design Automation - Lab 3

# Legalizer Report

110511210 李博安

## Brief Introduction

My program includes four parts, and each part is briefly introduced below.

1. **Reading Input**:

   - Reading the given inputs (.lg, .opt)

2. **Initial Placement**:

   - Place the initial placement in the .lg file into my “Row” data structure

3. **Legalization**: (will be elaborated in Pseudo code section)

   - The tetrisLegalization() function is called, ensuring cells are legally placed:
     - Adjusting the position of a newly added cell (new_FF) while ensuring no overlaps or violations with existing placements.
     - Using a movable "window" to isolate and adjust a subset of cells.
     - Repeatedly tries different positions for unplaced cells within a window until a legal solution is found.
   - Results are written to the output file.

4. **Optional Visualization and Metrics Output**:

   - Generate output for debug usage (visualizing)

## Pseudo Code

```
FUNCTION tetrisLegalization(input, output, row_vec, FF_map):
    WHILE input HAS data:
        1. Parse the new cell (`new_FF`) and remove cells from
           `FF_map` (merged cells)

        2. Initialize a placement window and add `new_FF` to it.
```

3. WHILE placement is not successful:
    a. Clear and set up the window with local cells and rows.
    b. Attempt to place all cells in the window using `tetrisPlace`.
    c. IF placement fails:
        - Adjust `new_FF`'s position randomly.
        - Retry with updated window.

4. Commit the placement:
    - Update `FF_map` and rows (`row_vec`) with new positions.
    - Store placement details for output.

5. Write placement results to the output.

END FUNCTION

FUNCTION tetrisPlace(output, row_vec, cell_unplaced, FF_map_temp):
1. Initialize a list for cells that remain unplaced (`still_unplaced`).

2. WHILE `cell_unplaced` is not empty:
    a. Take the last cell from `cell_unplaced`.

    b. IF the cell's upper-right corner is within the row boundaries:
        - Move the cell upward by one row height.

    c. ELSE:
        - Add the cell to `still_unplaced` and continue.

    d. Attempt to place the cell using `simplePlace`.

3. Replace `cell_unplaced` with `still_unplaced` for any remaining cells.

END FUNCTION

FUNCTION simplePlace(cell_ptr, row_vec, cell_unplaced):
1. Adjust `cell_ptr`'s position if it is out of bounds:
    a. Move it upward or downward until it is within the die boundaries.

2. Identify rows (`y_rows`) intersecting with `cell_ptr`:

a. Calculate the initial row index (`row_y_idx`) based on `cell_ptr`'s position.
b. Check if the cell intersects multiple rows and add those rows to `y_rows`.

3. Check placement legality:
   a. IF any row in `y_rows` does not allow placement, mark the placement as illegal.
   b. Look for a valid segment across all intersecting rows:
      - Find an empty segment in the first intersecting row.
      - Ensure this segment overlaps with empty segments in other rows.

4. IF a valid segment is found:
   a. Place the cell in the identified segment.
   b. Insert the cell into all intersecting rows (`y_rows`).
   c. Mark `cell_ptr` as placed.

5. ELSE:
   a. Add `cell_ptr` to `cell_unplaced` for retrying later.

END FUNCTION

## Time Complexity Analysis

- **simplePlace()**
  The bottleneck is the loop of finding an legal placement from the rows with the cells initial coordinate. Time complexity = $O(R' * S^2)$, where R' is the length of intersected rows, s is the segments in each row
- **TetrisPlace()**
  The bottleneck is calling simplePlace(), so time complexity is $O(N' * R' * S^2)$, where N' is the number of cells to place (a window usually have 10~20 cells)
- **tetrisLegalization()**
  The bottleneck is tetrisPlace(), so time complexity is $O(N * N' * R' * S^2)$, where N is the total nums of cell to optimize
- **Total**
  The total time complexity will be the complexity of tetrisLegalization(),

which is O(N * N' * R' * $S^2$), N is the cell number in .opt, N' is 10~20, R' is usually 4~16(cell height in .opt), and S will be at around (total cells)$^{1/2}$, so the time complexity will be **O(C * N * (total cells)$^{1/2}$),** where C is a constant.

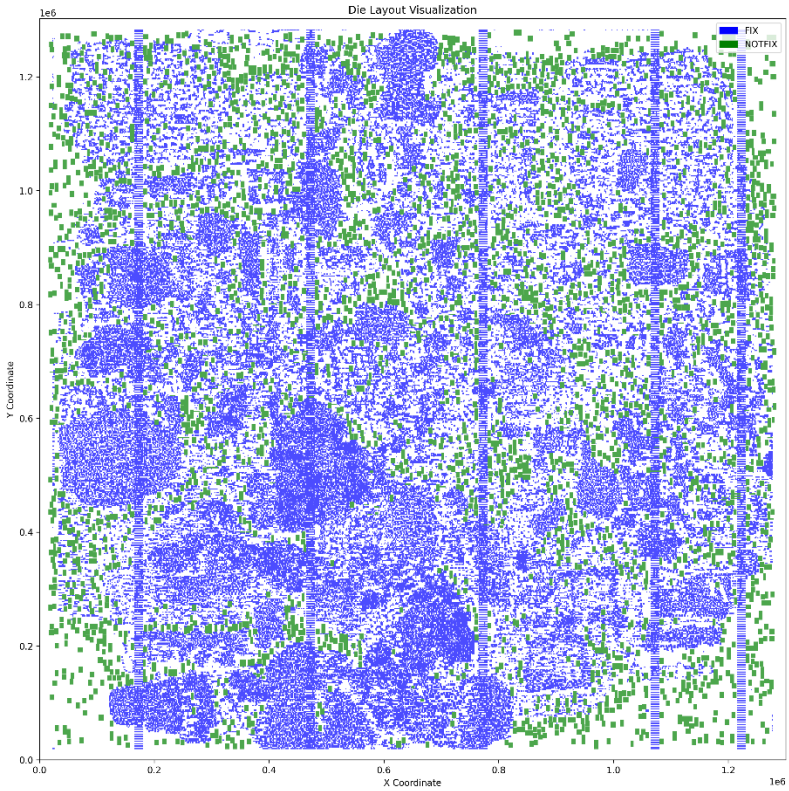
## Special Features

- **Window-Based Placement:**
  The "window" strategy keeps the computation localized, making the algorithm scalable for large layouts.
- **Random placement:**
  By iteratively trying new positions for new_FF randomly, the algorithm avoids deadlocks in placement where cell is too intense in some area, providing a legal placement in a reasonable time with the tradeoff of moved distance.
- **Tetris-Inspired Strategy:**
  The tetrisPlace function uses strategies similar to stacking blocks in the game Tetris, prioritizing compact and aligned placements.

## Feedback

I think it's a little hard for us, because it is more like a problem that no research has done before, unlike the first two labs. There may be some similar experiments, but there isn't an exact one that we can refer to. We need to find many papers and combine some of them to finish this lab, which is a bit difficult for me now, since I haven't get many training like this. But it's still accomplishing to finish the lab and see the blocks we moved be printed out on the screen (the importance of visualization!!).

By the way, I kind of finished the hellish case (with bad performance I think..), and it's interesting because the result is very sparse. The program went pretty slow around step 6300 of optimization, and became faster again after that. I think this is because the cells were removed by the optimization moves, and the area of merged FFs is way smaller than the original sizes of FFs, therefore the result placement is not so dense as I expected.




```
0:53 gahxi0769@vda04 [~/pda/lab3] >$ ./Legalizer testcase/testcase1_MBFF_LIB_7000.lg testcase/testcase1_MBFF_LIB_7000.opt r
esult/testcase1_MBFF_LIB_7000_post.lg
0:54 gahxi0769@vda04 [~/pda/lab3] >$ ./Evaluator testcase/testcase1_MBFF_LIB_7000.lg testcase/testcase1_MBFF_LIB_7000.opt r
esult/testcase1_MBFF_LIB_7000_post.lg
+---------------+--------------+--------+-----------------+---------------+
|      Cost     |       -      | Weight |      Value      | Percentage(%) |
+---------------+--------------+--------+-----------------+---------------+
|   Move Times  |    79200.00  | 100.00 |    7920000.00   |    0.00(%)    |
| Total Distance | 4195787877.00 | 200.00 | 839157575400.00 |    99.99(%)   |
|     Total     |       -      |   -    | 839165495400.00 |   100.00(%)   |
+---------------+--------------+--------+-----------------+---------------+
```

< testcase1_MBFF_LIB_7000 legalization result>

## Conclusion

I created a solution of legalizing FF placements after banking by combining the concept of window and Tetris legalization, acquiring a solution with reasonable run time and cost. My next step would be improving the placement algorithm of my program, from randomly choosing the window position to some more regular way, or find the segment to place the cell by a more efficient method, in order to achieve better cost within the same scale of runtime.