# CAD Contest Problem B:

# Power and Timing Optimization Using Multibit Flip-Flop

Introduction to Electronic Design and Automation - Final Project

B11902164 羅瑋宸

June 20, 2025

# Introduction

## The Challenge:

In modern semiconductor design, minimizing power consumption and silicon area while preserving timing is a critical challenge.

## Core Technique:

A key optimization strategy is the intelligent use of Multi-bit Flip-Flops (MBFFs).

## Key Operations:

### Banking

Substituting multiple single-bit flip-flops (FFs) with a single MBFF.

**Benefits:** This saves area and reduces power and clock routing complexity.

### Debanking

Splitting a large MBFF into several smaller FFs.

**Benefit:** This is necessary to better optimize timing-critical nets.

# Problem Formulation & Scoring

## Objective:

To develop a banking and debanking algorithm that produces a placement result optimized for timing, power, and area.

## Inputs:

- Cost function weights: Alpha (timing), Beta (power), Gamma (area).
- Initial design metrics: Total Negative Slack (TNS), Total Power (TPO), and Cell Area.
- Design database: LEF/DEF files for placement and an SDC file for timing constraints.
- A list of all available flip-flop library cells.

## Outputs:

- A pin mapping file detailing all changes from the original to the final design.
- A new design with the optimized FF placement in Verilog and DEF formats.

## Scoring Function (Lower is Better):

The quality of a solution is determined by a cost function. The initial score is calculated as:

$$Initial\ score = \alpha \cdot TNS + \sum_{\forall i \in FF} (\beta \cdot Power(i) + \gamma \cdot Area(i))$$

## Workflow

- Input Files (LEF/DEF, SDC, .v, .tf, .def, _weight)
1. Parsing (Read all design data into internal structures)
2. Logical Transformation (Clustering)
3. Physical Legalization
4. Output Generation
- Output Files (.txt, .v, .def)

# Architectural Framework

To solve persistent overlap and pin mapping errors, I adopted a robust **two-phase architecture** for all of the algorithms. This was the key to ensuring correctness.

### Phase 1: Logical Transformation

- The chosen algorithm (Greedy, K-Means, or Mean-Shift) analyzes the flip-flops.
- It decides *which* FFs to group together.
- The output is just a *list of decisions* (candidate cells), not a physical placement. This ensures every original FF is accounted for.

### Phase 2: Physical Legalization

- A separate placement engine takes the candidate list from Phase 1.
- It places each cell one-by-one onto a grid representing the die.
- Because each placement immediately updates the grid's occupancy, this process is guaranteed to be **overlap-free**.

# Algorithm 1: Greedy Clustering

**Concept:** A simple and fast bottom-up approach that builds clusters by making locally optimal choices at each step.

## Methodology:

1. Select an unprocessed flip-flop as a "seed" for a new cluster.
2. Calculate the center (centroid) of the current cluster.
3. Find the unprocessed neighbor that is physically closest to the cluster's centroid and whose bit-width fits within the `max_bit` limit.
4. Add this best neighbor to the cluster and update the centroid.
5. Repeat steps 3-4 until no more neighbors can be added.

## Pseudocode for Greedy Clustering:

```
PROCEDURE Greedy_Clustering(FlipFlops):
  FOR each clock_group:
    WHILE unprocessed_FFs exist in group:
      NewCluster = create_with_seed_FF()
      LOOP:
        find neighbor_FF closest to Centroid(NewCluster)
        IF neighbor_FF exists AND can_be_added(neighbor_FF):
          Add neighbor_FF to NewCluster
        ELSE:
          BREAK
      Add NewCluster to candidate_list
```

# Algorithm 2: K-Means Clustering

**Concept:** A classic top-down clustering algorithm that partitions data points into K groups by iteratively minimizing within-cluster distances. It is highly effective at creating spatially compact, spherical clusters. This is a standard approach referenced in prior work.

## Methodology:

1. **Initialization:** Heuristically determine the number of clusters, K, and randomly select K initial locations for the cluster centers (centroids).
2. **Assignment Step:** Assign each flip-flop to its nearest centroid.
3. **Update Step:** Recalculate the position of each centroid to be the mean (average position) of all flip-flops assigned to it.
4. **Repeat:** Repeat steps 2 and 3 until cluster assignments no longer change.

## Pseudocode for K-Means Clustering:

```
PROCEDURE K_Means(FlipFlops, K):
  1. Initialize K random centroids
  REPEAT
    2. FOR each FlipFlop:
         Assign FlipFlop to the nearest_centroid
    3. FOR each Centroid:
         Update Centroid_position = Mean(all FFs in its cluster)
  UNTIL no change in assignments
  RETURN clusters
```

## Algorithm 3: Mean-Shift Clustering

**Concept:** A density-based, non-parametric clustering algorithm that does not require specifying the number of clusters (K) beforehand. Its goal is to find the densest regions in the data distribution. This method is also cited in prior academic work.

### Methodology:

1. **Initialization:** Treat each flip-flop's location as a data point.Calculate the center (centroid) of the current cluster.
2. **Shift Step:** For each point, calculate the mean of all neighboring points within a defined bandwidth radius.
3. **Update Step:** Move the point to the calculated mean.
4. **Repeat:** Repeat steps 2 and 3. This process causes all points to "climb the hill" towards the densest areas.
5. **Group:** Points that converge to the same final location are grouped into a cluster.

## Pseudocode for Mean-Shift Clustering:

```
PROCEDURE Mean_Shift(FlipFlops, bandwidth):
  1. shifting_points = copy(FlipFlop_locations)
  REPEAT for a fixed number of iterations:
    2. FOR each point p in shifting_points:
         Neighbors = find_points_within_radius(p, bandwidth)
         p = calculate_mean_of(Neighbors)
  3. Group points that have converged to the same location
  RETURN clusters
```

# Experimental Results & Analysis

| Test Case | Greedy | K-Means | Mean-Shift |
|---|---|---|---|
| testcase1_0614 | **1.83e12** | 2.16e12 | 2.31e12 |
| testcase1_0812 | 1.05e9 | **8.72e8** | 9.26e8 |
| testcase2_0812 | 8.23e7 | 2.57e6 | **1.68e6** |
| testcase3 | **8.04e8** | 8.75e8 | 9.11e8 |

## Overall Performance

Across these four diverse test cases, there is no single algorithm that wins every time. However, the **K-Means algorithm demonstrates the most balanced and consistently strong performance**. While the Greedy algorithm won two cases, its scores show extreme variance. The K-Means algorithm was either the winner or a very close second in the cases it didn't win, making it the most reliable and robust strategy overall.

## Analysis Summary

- **Greedy:** A high-risk, high-reward strategy. It excels at reducing area and power but is highly unstable and can severely damage the timing score (TNS). It only wins when the cost function heavily favors area/power over timing.
- **Mean-Shift:** A conservative and stable algorithm. Its main strength is its ability to avoid making bad decisions, as seen in `testcase2_0812`. However, this also means it often misses opportunities for significant improvement.
- **K-Means:** The most robust and balanced performer. Its strategy of creating geometrically compact clusters is a very effective heuristic for this problem, consistently delivering good, reliable results across a variety of test cases.

## Conclusion

- Implementing three clustering algorithms within a robust two-phase framework that guarantees correct, overlap-free output.
- Through experimentation, the **K-Means clustering algorithm** proved to be the most effective and stable strategy for minimizing the contest's cost function.

Thank

you!