EE6332: Modelling and Optimization in VLSI

Arunachalam Ramesh (EP22B004)
Department of Electrical Engineering
Indian Institute of technology, Madras

May 25, 2025

1 Phase I: Continuous Gate sizing

1.1 Question 1: Compute and Analyze T_{wall}

1.1.1 Objective

Implement the GP-based gate sizing algorithm and determine the minimum possible delay (T_{wall}) for each ISCAS-85 circuit.

1.1.2 Approach

Constraints Used in the GP Formulation

The following constraints were imposed to optimize the critical path delay:

(a) **Gate Delay Model:**

$$d_i = \frac{g_i \cdot C_{\text{load},i}}{C_{\text{in},i} \cdot x_i} + p_i \quad \forall i \in \text{gates}$$

where g_i is the logical effort, p_i the parasitic delay, $C_{\text{in},i}$ the input capacitance, $C_{\text{load},i}$ the fanout load, and x_i the gate size.

(b) **Timing Constraints:**

$$T_i \geq T_j + d_i \quad \forall (j \to i) \in \text{fanin edges}$$

$$T_i \geq d_i \quad \text{if } i \text{ has no fanins}$$

where T_i is the arrival time of the gate handling the output at the node i.

(c) **Primary Input Load Constraint:**

$$\sum_{g \in \text{fanouts}(pi)} C_{\text{in},g} \cdot x_g \leq 150 \quad \forall pi \in \text{primary inputs}$$

Since the inputs given to these circuits can handle the size of only 50x inverter, we formulate the following taking the sum of the default gate capacitance multiplied with the size of that gate to be less than the gate capacitance of 50x load.

(d) **Gate Size Bounds:**

$$1 \leq x_i \leq 64 \quad \forall i$$

These are the given gate size bounds

(e) **Final Output Delay Bound:**

$$T_{wall} \geq T_i \quad \forall i \in \text{primary outputs}$$

The T_{wall} should be less than or equal to the arrival times of all the primary outputs.

The optimization minimizes $T_{\max}$, which represents the worst-case delay across all output paths. All delays are later scaled by a technology-dependent time constant τ.

1.1.3 Results

Table 1: T_{wall} for ISCAS-85 Circuits (in ps)

Circuit	T_{wall} (1 unit here is 5 ps)
c17	28.83
c432	153.47
c880	118.93
c1908	177.58
c2670	178.69
c3540	220.74
c5315	203.07
c6288	564.64
c7552	167.86

1.2 Question 2: Area Analysis for Multiple T_{spec} Values

1.2.1 Objective

Evaluate how the total area varies for $T_{\text{spec}}/T_{\text{wall}} \in \{1.05, 1.1, 1.2, 1.3, 1.4, 1.5\}$.

1.2.2 Approach

Modified Constraints for Area Optimization

The geometric program is modified to minimize the total gate area while ensuring that the delay does not exceed a target specification time T_{spec}.

The following changes are made to the previous formulation:

(a) **New Objective:** Instead of minimizing $T_{\max}$, the new objective is to minimize the total gate area:

$$\min \sum_i x_i$$

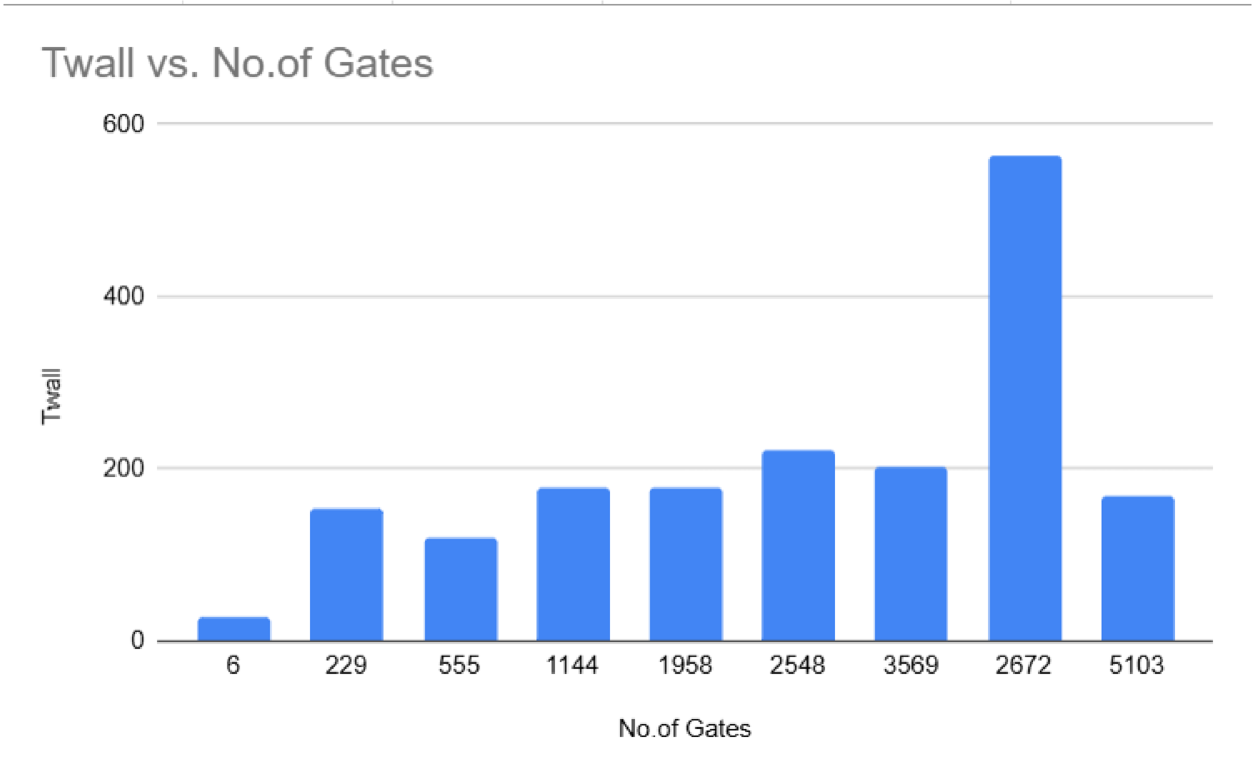

Figure 1: Twall vs No.of Gates

(b) **Arrival Time Constraint for Final Outputs:** For all gates with no fanouts (i.e., primary outputs), enforce:

$$T_i \leq T_{\text{spec}} \quad \forall i \in \text{primary outputs}$$

(c) **Retained Constraints:** All other constraints from the delay minimization formulation are retained:

- Gate delay expression
- Gate-to-gate timing propagation
- Primary input fanout load constraint:

$$\sum_{\text{fanouts}(pi)} C_{\text{in}} \cdot x \leq 150$$

- Gate size bounds:

$$1 \leq x_i \leq 64$$

This optimization is solved iteratively for multiple values of $T_{\text{spec}} = \alpha \cdot T_{\text{wall}}$ with $\alpha \in \{1.05, 1.1, 1.2, 1.3, 1.4, 1.5\}$.

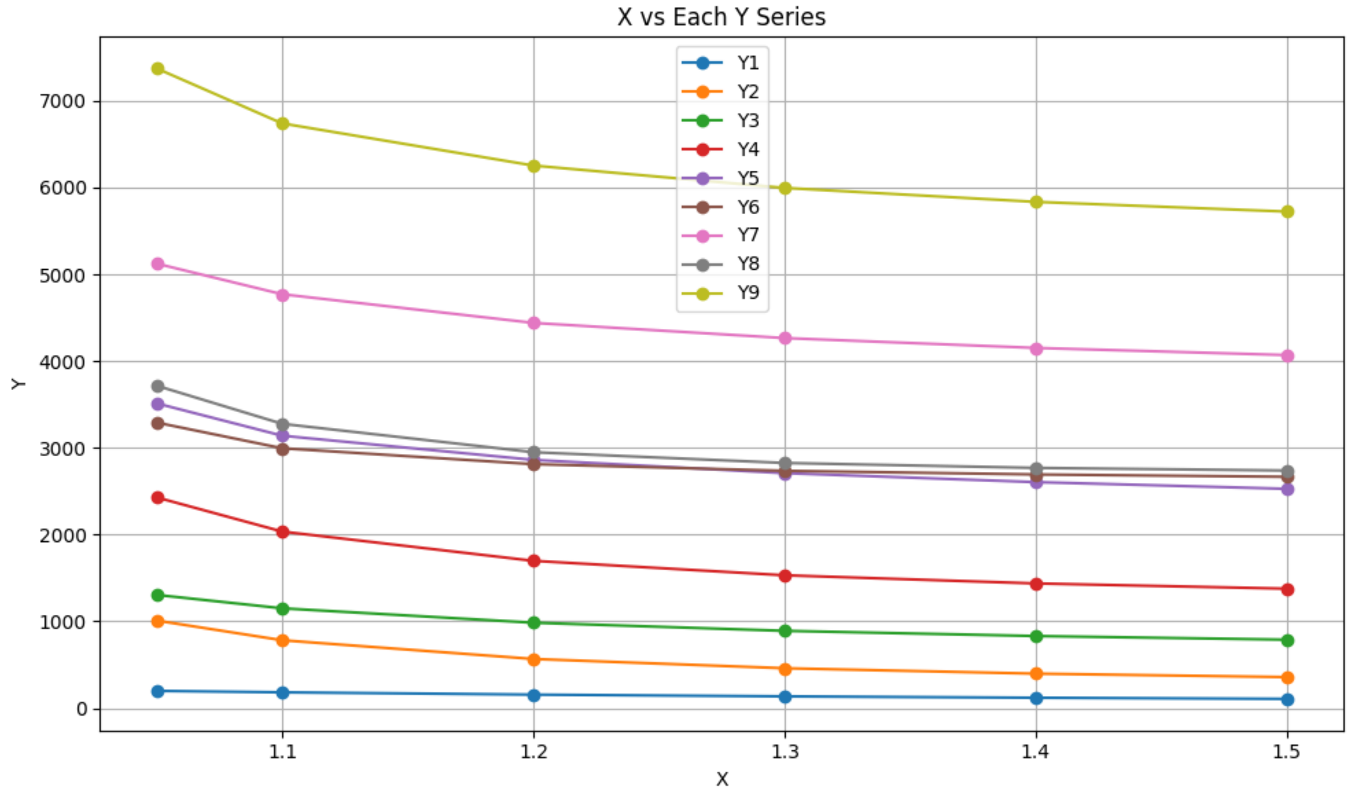

Figure 2: Various circuits Amin for Tspecs. Y1 is c17, Y8 is c7552

1.2.3 Results

Table 2: Minimum Area vs. T_{spec} for Various ISCAS-85 Circuits

Circuit	$A_{\min}$ **(1.05** T_{wall}**)**	$A_{\min}$ **(1.1** T_{wall}**)**	$A_{\min}$ **(1.2** T_{wall}**)**	$A_{\min}$ **(1.3** T_{wall}**)**	$A_{\min}$ **(1.4** T_{wall}**)**	$A_{\min}$ **(1.5** T_{wall}**)**
c17	199.17	182.75	156.35	136.13	120.22	107.41
c432	1009.93	782.82	567.14	460.32	397.89	358.31
c880	1305.52	1150.84	984.93	891.26	831.44	789.36
c1908	2429.47	2035.38	1698.05	1531.52	1437.48	1377.11
c2670	3513.12	3140.14	2861.34	2708.80	2605.20	2527.74
c3540	3291.86	2995.35	2812.11	2735.50	2693.50	2666.15
c5315	5123.67	4770.46	4440.02	4265.34	4151.61	4069.35
c6288	3716.93	3277.14	2949.73	2825.95	2769.09	2738.74
c7552	7373.47	6739.31	6253.37	5995.43	5834.97	5722.60

1.3 Question 3: Critical vs Non-Critical Path Analysis

1.3.1 Objective

Analyze gate sizes on the critical path and compare them with those on non-critical paths.

1.3.2 Discussion

Critical path gate sizes: Gate sizes are such that the final fan-out arrival time is minimized. This is done by gradually increasing the size of gates in the critical path from a minimum value within a maximum value of 64 depending on the number of gates in the path. We can observe in our outputs that, the gate sizes on the noncritical paths are sized(much lesser than that on the critical paths) so that the final arrival time of all fan-outs is the same as the final arrival time of the critical path.

Your observations:

1.4 Question 4: Discretization Effects

1.4.1 Objective

So for the given set of constraints, the solution we get as in Question 1 is the most optimal to find minimum T_{wall}. However, since they may not be available in real life, you might have to discretize them into available sizes. Here we assume that all integer sizes are available.

1.5 Discussion of Method I used in project

Once the continuous gate sizes obtained from geometric programming are computed, a discretization step is applied to map them to implementable sizes. The discretization rule is as follows:

- For gates in the set of primary outputs $\mathcal{P}_{\text{out}}$, the sizes are **rounded up** using the ceiling function:

$$s_g^{\text{discrete}} = \lceil s_g \rceil \quad \text{if } g \in \mathcal{P}_{\text{out}}$$

- For all other gates, the sizes are **rounded down** using the floor function:

$$s_g^{\text{discrete}} = \lfloor s_g \rfloor \quad \text{if } g \notin \mathcal{P}_{\text{out}}$$

After discretization, we evaluate the impact of rounding on timing. For each gate, the delay is computed using the logical effort model:

$$D_g = \tau \left(\frac{g_g \cdot C_{\text{load},g}}{C_{\text{in},g} \cdot s_g^{\text{discrete}}} + p_g \right)$$

where g_g is the logical effort, p_g is the parasitic delay, $C_{\text{in},g}$ is the input capacitance, s_g^{discrete} is the discretized size, and τ is the technology constant.

Arrival times are computed in topological order. The critical path is traced back from the gate with the maximum arrival time. Violations are checked for:

- **Timing violations**: gates where arrival time exceeds $1.09 \cdot T_{\max}$.
- **Size violations**: gates where $s_g^{\text{discrete}} > 64$.

- **Primary input load violations**: primary inputs driving gates whose total load exceeds 150 units.

Your observations:

This is a very basic way. This gives solutions which can be better by huge bounds. For example, the least slack I can get for all 8 circuits to obey timing is 9 percent. But the total area used is just slightly off from the once we get for Twall case, and much higher than what we get for Area Minimization problem for T_{spec} being $1.1 * T_{wall}$.

1.5.1 Better Way to Approach the problem

Using slack and free kind of algorithm is the best way to solve this problem. So say u need a minimum area solution to get for 10 percent from the minimum time, then you can solve the problem of GP with continuous values with slack of 5 percent, and then use the remaining 5 percent slack we have extra to minimize to take it to a space of having 10 percent slack. That is one of the better algorithms we can use, there by minimizing the area as well as going off 10 percent from minimum time which anyway we get by downsizing all gates except the primary outputs(which we upsize to get 9 percent, downsizing even them pushed the slack required upto 15 percent)

Your rationale:

To optimize gate sizing, we solve a geometric program that minimizes total gate sizes subject to timing constraints. A relaxed timing specification is used to allow feasible optimization:

$$T_{\text{spec}} = 1.05 \times T_{\max}$$

This slack (5%) ensures that the circuit meets timing with some margin, accommodating estimation or modeling inaccuracies.

However My Algorithm or Rather the Ceiling and Flooring gave me:

1.5.2 Results

Table 3: Changes due to downsizing

Circuit	$\boldsymbol{A_{\min}}$ ($\mathbf{T_{arrival}}$ **as factor of** $\mathbf{T_{wall}}$)	$\boldsymbol{A_{\min}}$ **(Area)**
c17	1.010	220
c432	1.039	2729
c880	1.052	4614
c1908	1.021	15926
c6288	1.079	33834
c7552	1.086	45957

1.6 Question 5: Drive Strength Design Decisions

1.6.1 Objective

As a standard cell designer, decide on a good set of drive strengths (for INV, NAND2-4, NOR2-4).

Histogram-Based Decision Making

After solving for the optimal gate sizes, we generate histograms grouped by gate type (e.g., NAND, NOR, INV). Each histogram shows the distribution of gate sizes (drive strengths) chosen by the optimizer.

- **X-axis:** Gate size (rounded to nearest integer)
- **Y-axis:** Number of gates with that size

These plots help in:

- Identifying frequently used drive strengths
- Guiding standard cell library design (e.g., which sizes to implement)
- Detecting over-sizing or under-sizing trends in specific gate types

For example, if most inverters fall between sizes 2 and 6, then only these drive strengths need to be included in the library, reducing layout effort and area overhead.

Now By making these for all our circuits, we are able to observe that we have most of our gates lying mostly around 3 and 4, and few being dispersed from 11-64. So mostly, we can design or have many sizes available in our library within 10, and we can have lesser sizes post 10, since there isn't to many gates required with size greater than 10, and we can afford to use sizes close by (whatever we can fit in our standard cell library). So my typical library size will be (1x, 2x, 3x, 4x, 5x, 10x, 16x, 32x, 48x, 60x for NOT gates) (1.75x, 2x, 2.25x, 2.5x, 3x, 3.5x, 4x - NOR) (1.5x, 2x, 2.5x, 3x, 5x, 9x, 21x, 34x, 48x, 64x for NAND) (The given sizes are based on outputs of c17, c432, c880).

1.7 Bonus Question: Area Minimization Using Slack

1.7.1 Objective

Can the area be minimized by utilizing available slack, under fixed load/input capacitance constraints?

The optimizer designed by me in this assignment is already equipped with such that there is no positive slack present in the circuit. The slacks have already made zero in all paths by making the gate sizes lesser at required places so that there is no slack present. So, the area that is mentioned in the above part is the minimum area that is optimised by removing the positive slacks that are present in the circuit.

2 Phase II - Continuous Gate sizing

In this work, we implement a discrete gate sizing methodology based on the Multi-Moves heuristic, as proposed in literature for timing-aware optimization of digital circuits. The algorithm starts from an initial gate sizing solution obtained from a geometric programming (GP) solver and iteratively improves it while respecting delay constraints.

2.1 Core Components

The following are the primary components of the optimization pipeline:

- **Arrival and Required Arrival Time (RAT):** Arrival times are computed using a DFS over the circuit graph, and required arrival times are back-propagated from primary outputs using a global timing constraint T_{wall}.
- **Slack Computation:** Slack for each gate is evaluated as:

  $$\text{Slack} = \text{RAT} - \text{Arrival Time}$$

  which serves as the basis for evaluating timing safety of local perturbations.
- **Multi-Moves Optimization:** This optimization technique is inspired by the gate sizing approach presented in *Gate Sizing: A General Purpose Optimization Approach* by Coudert. The core idea is to iteratively improve gate sizes by exploring all legal sizing options for each gate and evaluating their impact on both delay and area in a localized subnetwork.

  The optimization follows these steps:

  1. For each gate, a localized subnetwork (typically its depth-1 fanin and fanout neighborhood) is extracted to evaluate the impact of possible sizing moves. This is based on the insight that the influence of a sizing change diminishes rapidly as one moves away from the gate.
  2. All valid sizing options for the gate are tested, and for each, a cost is computed based on area and slack. Only moves that improve the cost are retained as candidates.
  3. The optimal sizing move is chosen for each gate out of all the candidates. A *multi-move* update to the circuit is created by applying these moves together.
  4. The upgraded gates' local neighborhoods are flagged for reassessment in the upcoming iteration. This process continues until a stopping criterion (such as the maximum number of iterations or runtime) is met, or no further improvements are possible.

  This approach strikes a balance between convergence speed and solution quality, avoiding the drawbacks of purely greedy or purely global techniques. By evaluating costs in small subnetworks and limiting updates to affected regions, it avoids costly full-circuit recomputation, making the approach scalable to large circuits. Moreover, by allowing

```python
def extract_subnetwork(self, gate_name, depth=1):   #Making the local chang
    visited, frontier = set(), {gate_name}
    for _ in range(depth):
        next_frontier = set()
        for g in frontier:
            gate = self.gates[g]
            for inp in gate.inputs:
                if inp in self.parser.node_to_gate:
                    next_frontier.add(self.parser.node_to_gate[inp])
            for fout in self.parser.reverse_graph.get(gate.output, []):
                if fout in self.parser.node_to_gate:
                    next_frontier.add(self.parser.node_to_gate[fout])
        visited.update(frontier)
        frontier = next_frontier - visited
    return visited | frontier
```

Figure 3: Sub-network

```python
def evaluate_local_cost(self, subnetwork, rat_times): #Local cost
    delays = self.compute_delays()
    arrival = self.compute_arrival_times(delays)
    slacks = self.compute_slacks(arrival, rat_times)
    S = min(slacks.values())

    if S < 0:
        return (float('inf'), float('inf'))

    area = sum(self.gates[g].current_size for g in subnetwork)
    return (-S, area)
```

Figure 4: Calculation of Cost

```python
def go_to_local_min(self, rat_times): #Picking a gate, locating its closeby sub-ne
    import time                        #across the available sizes for the given ga
    self.rat_times = rat_times
    update = set(self.gates.keys())
    converged = False
    iteration = 0
    start_time = time.time()
    while not converged and iteration < 16 and time.time()-start_time<1800:
        moves = {}
        for gate_name in update:
            gate = self.gates[gate_name]
            original_size = int(gate.current_size)
            subnetwork = self.extract_subnetwork(gate_name, depth=1)

            best_move = original_size
            best_cost = self.evaluate_local_cost(subnetwork, rat_times)

            for size in sorted(gate.size_options):
                size = int(size)
                if size == original_size:
                    continue

                gate.current_size = size
                cost = self.evaluate_local_cost(subnetwork, rat_times)
                gate.current_size = original_size

                if cost is not None and cost < best_cost:
                    best_cost = cost
                    best_move = size
```

Figure 5: Finding minima

multiple coordinated gate size changes in each iteration, the optimizer is able to escape poor local optima and move toward higher-quality solutions.

This strategy avoids the pitfalls of purely greedy or global methods by balancing convergence speed and solution quality. By evaluating cost in small subnetworks and propagating only necessary updates, the method remains scalable to large circuits while avoiding expensive full-circuit recomputation. Additionally, it enables the optimizer to escape poor local optima by allowing multiple coordinated gate size changes per iteration.

- **Simulated Annealing Refinement:** If the local multi-moves optimization converges prematurely or fails to yield an acceptable solution within timing constraints, a simulated annealing (SA) based refinement step is invoked. In this phase, the sizing parameters (i.e., gate sizes) are restricted to integer values from the cell library, and random perturbations are applied to these discrete sizes. The cost function used during annealing is:

$$\text{Cost} = \text{Area} + \lambda \cdot \text{Penalty}(T)$$

where `Area` is the total gate area, λ is a tunable penalty coefficient, and `Penalty`(T) quantifies the timing violation. Specifically, timing slack violations are penalized quadratically based on how far the critical path delay deviates from the timing target T_{wall}.

Unlike traditional annealing that perturbs both design parameters and constraints, this method fixes the sizing options and instead gradually *relaxes the timing constraint* by tolerating slight violations in early stages. This is achieved using a penalty term that allows some flexibility in timing in exchange for improved area, with the hope that better configurations (with respect to both timing and area) can be uncovered beyond the current local minimum.

The annealing process explores the solution space probabilistically, accepting worse solutions with a certain probability to escape local minima. Over time, the temperature parameter cools down, reducing the likelihood of accepting worse solutions and thereby encouraging convergence to a refined minimum. The timing constraint is still enforced through the penalty mechanism, but its enforcement is smooth and adaptive rather than binary, enabling the exploration of near-feasible regions.

- **Greedy Slack Adjustment:** Following the simulated annealing phase, a final greedy pass is executed to clean up the solution. Any gates that exhibit significantly positive slack are evaluated for potential downsizing. If reducing their size does not violate the timing constraint (i.e., slack remains non-negative), the smaller size is adopted. This step ensures that area is minimized wherever timing allows, tightening the solution to an efficient final configuration.

```python
def repeater(parser, sizes_ceil, Twall, gate_nodes, a, max_repeats=11):
    best_overall_sol, best_overall_cost = simulated_annealing(parser, sizes_ceil, Twall, gate_nodes, a)
    for repeat in range(max_repeats):
        VarA, constraints_area, gate_nodes,_, pog = constraints_former(1, parser, Twall, 1.2 +  a)
        sizes, area = optimize_area(VarA, constraints_area, gate_nodes, pog)
        sizes_ceil = sizes_ceiler(sizes,pi_to_gates)

        new_sol, new_cost = simulated_annealing(parser, sizes_ceil, Twall, gate_nodes, a)

        if new_cost < best_overall_cost:
            best_overall_cost = new_cost
            best_overall_sol = new_sol
        else:
            print(f"No improvement in iteration {repeat + 1}, stopping.")
            break  # Stop since there's no improvement
        a+=0.01
    return best_overall_sol, best_overall_cost
```

Figure 6: Repeater block for SA+GP

2.2 Convergence Criteria

The optimization process terminates when:

1. No further improvement in cost is observed in local moves.
2. A maximum number of iterations or runtime limit is exceeded.
3. Simulated annealing steps do not produce a better solution across multiple rounds.

2.3 Final Output

The final result is a discrete sizing assignment for each gate that minimizes total area while satisfying the timing constraints, verified using slack analysis on the final netlist. However, outputs for the circuits which does not involve heuristic calculation does have slight violation of slack, say it goes up 1.305 times the minimum time, instead of 1.3 because strict enforcement of negative slack hinders the process of solution obtained in my code. Especially in the case of c2670, post which all the circuits started being resorted by heuristic or the SA+GP code I made.

Circuit	$\mathbf{A_{min}}$ **($\mathbf{T_{spec} = 1.3 \times T_{wall}}$)**	**Multi-Moves + Slack Adjustment + (SA+GP)**	**SA+GP**
c17	136.13	142	142
c432	460.32	493	520
c880	891.26	937	967
c1908	1531.52	1658	1666
c2670	2708.80	3138	2980
c3540	2735.50	2935	2846
c5315	4265.34	4625	4590
c6288	2825.95	3136	2996
c7552	5995.43	6346	6446

Table 4: Comparison of Area Results for Different Gate Sizing Strategies

Here Only SA+GP is something which I thought of, mixing the ideas of multimoves and heuristic rounding. Though it is not purely greedy, it is sequentially greedy, and extremely useful when we need to obey timing constraint more than anything else, atleast for the parameter set I used.