
**EC440 VLSI Design Automation
Project (181EC135) Documentation
Release 2.0**

Rahul M Hanchate

25 November 2020

Contents

Packages Imported.....	3
Parse Program.....	4
Partitioning.....	10
Floorplanning.....	14
Placement.....	19
Routing	22
Other Netlists.....	26
References	26

Packages Imported

The following packages have been imported to implement and visualize VLSI Design Automation for partitioning, floorplanning, placement and routing.

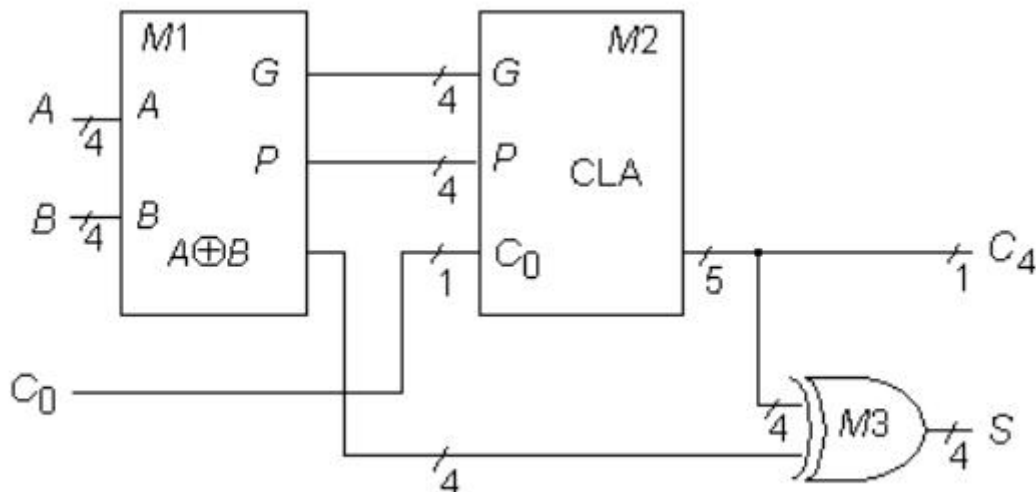
1. numpy
2. random
3. networkx
4. matplotlib
5. math
6. os
7. time
8. timeit

Parse Program

A parsing program (*parse.py*) has been created in order to parse and extract the circuit out of the netlist. This program is compatible with both ISCAS-85 and ISCAS-89 netlists.

Hence, I'll be able to do VLSI System Design Automation on **any of the netlists** given on the website. I'll be using [Fast Adder Circuit \(74283.isc\)](#) as an example throughout this project.

The circuit for Fast Adder Circuit is:

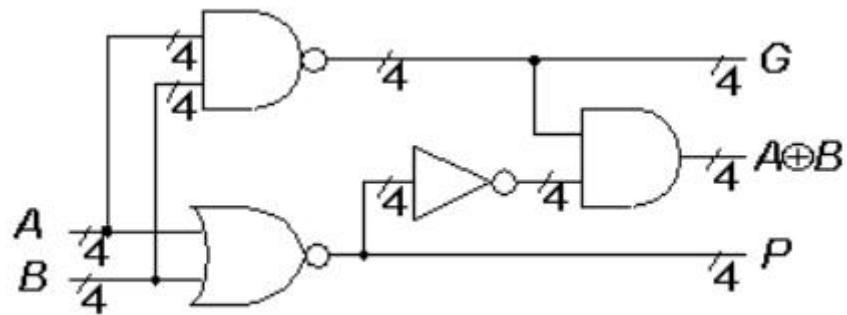


This circuit has:

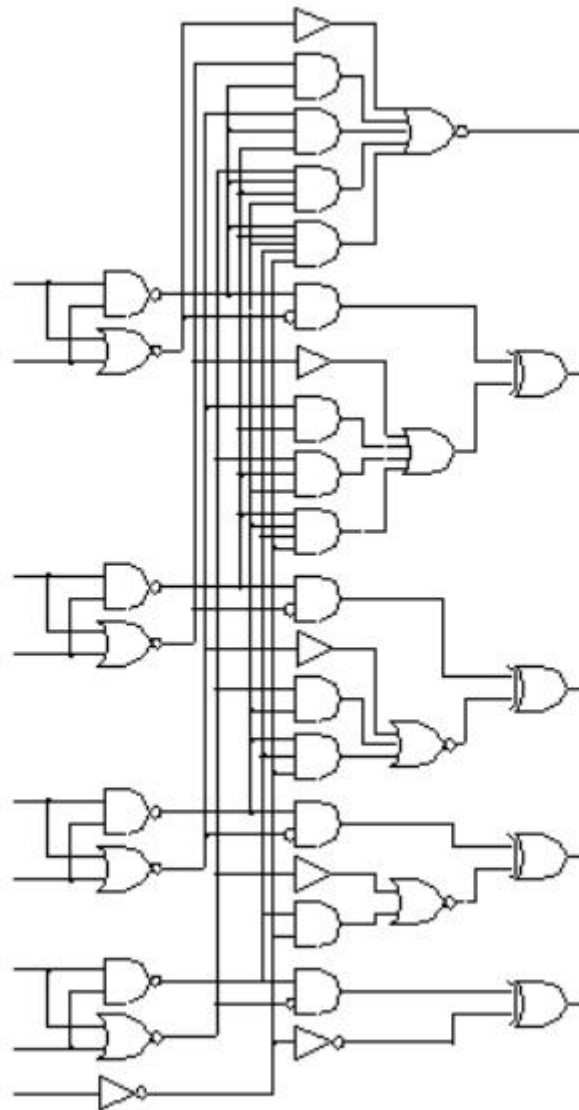
- 9 inputs
- 5 outputs
- 36 gates

The module M1 produces the generate, propagate, and XOR functions. The module M2 is similar to the 74182 (Carry Look-Ahead Circuit). The XOR word gate M3 produces the sum function.

The gate level schematic for module M1 is given below:



The gate level schematic of the entire circuit is given below:



Other netlists can be downloaded from [here](#). There are various files with *.isc* and *.bench* extensions which are ISCAS-85 and ISCAS-89 netlists respectively.

The parse.py program uses this file to generate required input for floor planning, partition, placement and routing programs.

Functions:

- **netadj85(filename)**: Parse function for ISCAS-85 netlist.
- **netadj89(filename)**: Parse function for ISCAS-89 netlist.

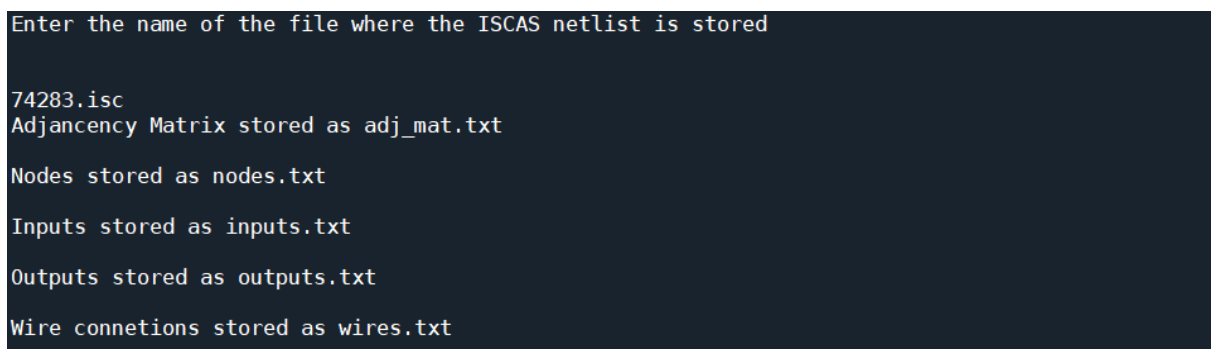
Input:

- **filename**: xyz.isc or xyz.bench

Output:

- **adj_mat.txt**: Contains the adjacency matrix denoting the connections between different nodes. The nodes are usually gates and sometimes a part of the logic (DFFs, etc)
- **nodes.txt**: Contains the list of all nodes of graph which will be created later. It contains nodes, type of nodes and number of inputs. This will help while creating the blocks in floorplanning.
- **inputs.txt**: Contains the list of all input nodes of the graph.
- **outputs.txt**: Contains the list of all output nodes of the graph.
- **wires.txt**: Contains the dictionary of all connected nodes

The following image shows a demonstration of how the parse.py works.



```
Enter the name of the file where the ISCAS netlist is stored
74283.isc
Adjacency Matrix stored as adj_mat.txt
Nodes stored as nodes.txt
Inputs stored as inputs.txt
Outputs stored as outputs.txt
Wire connetions stored as wires.txt
```


outputs.txt:

```
outputs.txt x
1 97gat
2 101gat
3 102gat
4 103gat
5 104gat
6
```

nodes.txt:

```
nodes.txt x
1 ['26gat', 'not', 1]
2 ['32gat', 'nand', 2]
3 ['38gat', 'nor', 2]
4 ['41gat', 'nand', 2]
5 ['49gat', 'nor', 2]
6 ['53gat', 'nand', 2]
7 ['61gat', 'nor', 2]
8 ['66gat', 'nand', 2]
9 ['72gat', 'nor', 2]
10 ['78gat', 'not', 1]
11 ['79gat', 'not', 1]
12 ['80gat', 'not', 1]
13 ['81gat', 'not', 1]
14 ['82gat', 'and', 2]
15 ['83gat', 'and', 3]
16 ['84gat', 'and', 4]
17 ['85gat', 'and', 5]
18 ['86gat', 'and', 2]
19 ['87gat', 'and', 2]
20 ['88gat', 'and', 3]
21 ['89gat', 'and', 4]
22 ['90gat', 'and', 2]
23 ['91gat', 'and', 2]
24 ['92gat', 'and', 3]
25 ['93gat', 'and', 2]
26 ['94gat', 'and', 2]
27 ['95gat', 'and', 2]
28 ['96gat', 'not', 1]
29 ['97gat', 'nor', 5]
30 ['98gat', 'nor', 4]
31 ['99gat', 'nor', 3]
32 ['100gat', 'nor', 2]
33 ['101gat', 'xor', 2]
34 ['102gat', 'xor', 2]
35 ['103gat', 'xor', 2]
36 ['104gat', 'xor', 2]
```


wires.txt

```
wires.txt x
1  {'25': ['26'],
2  '1': ['32', '38'],
3  '4': ['32', '38'],
4  '7': ['41', '49'],
5  '10': ['41', '49'],
6  '13': ['53', '61'],
7  '16': ['53', '61'],
8  '19': ['66', '72'],
9  '22': ['66', '72'],
10 '38': ['78', '97'],
11 '49': ['79', '82', '98'],
12 '61': ['80', '83', '87', '99'],
13 '72': ['81', '84', '88', '91', '100'],
14 '32': ['82', '83', '84', '85', '86'],
15 '41': ['83', '84', '85', '87', '88', '89', '90'],
16 '53': ['84', '85', '88', '89', '91', '92', '93'],
17 '66': ['85', '89', '92', '94', '95'],
18 '26': ['85', '89', '92', '94', '96'],
19 '78': ['86'],
20 '79': ['90'],
21 '80': ['93'],
22 '81': ['95'],
23 '82': ['97'],
24 '83': ['97'],
25 '84': ['97'],
26 '85': ['97'],
27 '87': ['98'],
28 '88': ['98'],
29 '89': ['98'],
30 '91': ['99'],
31 '92': ['99'],
32 '94': ['100'],
33 '86': ['101'],
34 '98': ['101'],
35 '90': ['102'],
36 '99': ['102'],
37 '93': ['103'],
38 '100': ['103'],
39 '95': ['104'],
40 '96': ['104']}
```

Partitioning

Kernighan Lin Algorithm

Kernighan Lin is probably the best and one of the easiest way to implement partitioning. It proposes a graph which is a bi-sectioning algorithm which starts with a random initial partition and then uses pairwise swapping of vertices between partitions, until no improvement is possible. One of the problems with the K-L algorithm is the requirement of prespecified sizes of partitions. This algorithm based on group migration are used extensively in partitioning VLSI circuits.

It starts by initially partitioning the graph $G = (V, E)$ into two subsets of equal sizes. Vertex pairs are exchanged across the bisection if the exchange improves the cutsize. The cost of partition is called the cutsize, which is the number of hyperedges crossing the cut. The above procedure is carried out iteratively until no further improvement can be achieved.

The algorithm is given below:

```
Algorithm KL
begin
  INITIALIZE();
  while( IMPROVE(table) = TRUE ) do
    (* if an improvement has been made during last iteration,
    the process is carried out again. *)
    while ( UNLOCK(A) = TRUE ) do
      (* if there exists any unlocked vertex in A,
      more tentative exchanges are carried out. *)
      for ( each  $a \in A$  ) do
        if (  $a = \text{unlocked}$  ) then
          for( each  $b \in B$  ) do
            if (  $b = \text{unlocked}$  ) then
              if (  $D_{\max} < D(a) + D(b)$  ) then
                 $D_{\max} = D(a) + D(b)$ ;
                 $a_{\max} = a$ ;
                 $b_{\max} = b$ ;
              TENT-EXCHGE( $a_{\max}, b_{\max}$ );
              LOCK( $a_{\max}, b_{\max}$ );
              LOG(table);
               $D_{\max} = -\infty$ ;
            ACTUAL-EXCHGE(table);
          end.
    end.
```

The algorithm is implemented in Python and produces partitions with least amount of cutsize possible. The time complexity of Kernighan-Lin algorithm is $O(n^3)$. The complexity of this algorithm is considered too high even for moderate size problems.

It starts with a randomly assigning the nodes into two groups obtained from the parsing program. It is then passed to *kernighanlin()* function which continuously iterates the *bestswap()* function. If the cutsize is converging, it breaks out of the *kernighanlin()* function. The *bestswap()* function iterates for all the number of pairs available in which nodes are exchanged and looked for the lowest cutsize by subtracting the previous cutsize with the current iteration. If the difference is seen to be the best amongst all iterations, the partitions for that iteration is taken into consideration. But during this practise, we might also take some amount of worse iterations during the search for the best iteration. All the pairs are swapped exactly once during *bestswap()* iterations. Thus, we can say that this is an example of a heuristic algorithm.

The characteristics of the Kernighan Lin implementation of partitioning in Python is given below:

Functions:

- **cutsize():**
 - **Inputs:** partition1, partition2, adjacency matrix
 - **Output:** cutsize
- **bestswap():**
 - **Inputs:** partition1, partition2, adjacency matrix, number of pairs
 - **Outputs:** Lowest cutsize among the iterations, the partitions corresponding to that cutsize.
- **kernighanlin():**
 - **Inputs:** partition1, partition2, adjacency matrix, number of pairs
 - **Outputs:** Final cutsize and partitions

Inputs:

- Adjacency Matrix file generated from the parsing program.
- Nodes from the parsing program. (For visualization only)

Outputs:

- Initial partitions with cutsize.
- Final partitions with cutsize.
- Graph visualization of both the partitions.

The following image shows the output of the *partitioning.py* for the Fast Adder Circuit (74283.isc).

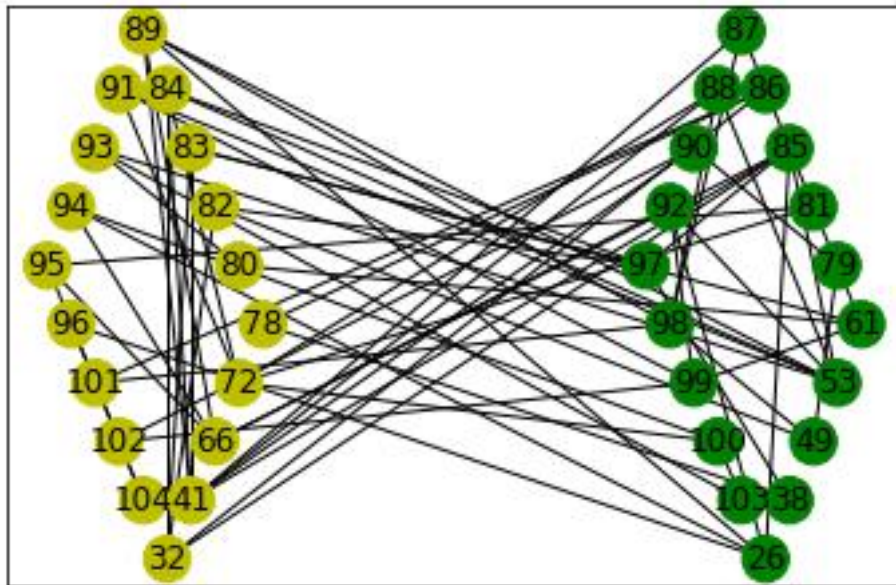
```
Enter the name of the file containing the required netlist:

74283.isc

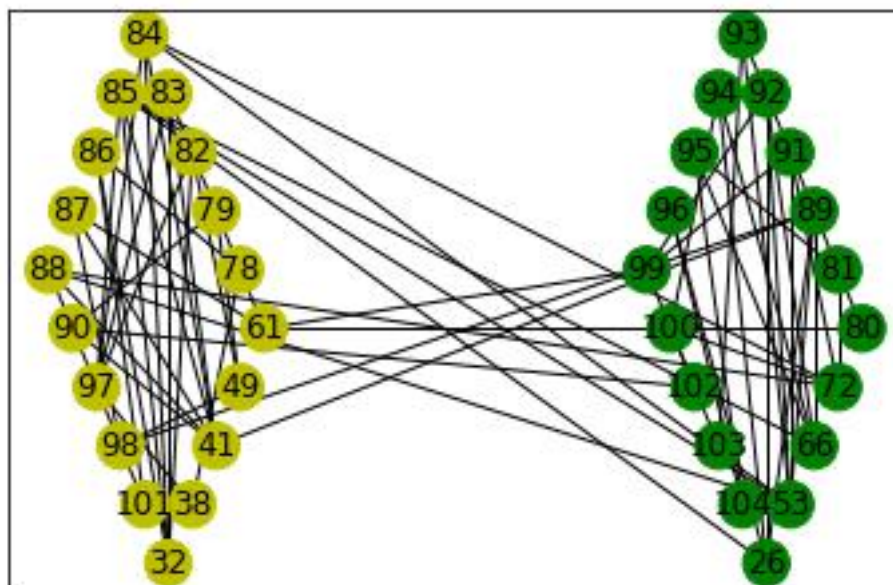
Initial Cut size is : 35
Initial Partition 1 : ['32gat', '41gat', '66gat', '72gat', '78gat', '80gat', '82gat',
'83gat', '84gat', '89gat', '91gat', '93gat', '94gat', '95gat', '96gat', '101gat', '102gat',
'104gat']
Initial Partition 2 : ['26gat', '38gat', '49gat', '53gat', '61gat', '79gat', '81gat',
'85gat', '86gat', '87gat', '88gat', '90gat', '92gat', '97gat', '98gat', '99gat', '100gat',
'103gat']
Final Cut size is : 12
Final Partition 1 : ['32gat', '38gat', '41gat', '49gat', '61gat', '78gat', '79gat', '82gat',
'83gat', '84gat', '85gat', '86gat', '87gat', '88gat', '90gat', '97gat', '98gat', '101gat']
Final Partition 2 : ['26gat', '53gat', '66gat', '72gat', '80gat', '81gat', '89gat', '91gat',
'92gat', '93gat', '94gat', '95gat', '96gat', '99gat', '100gat', '102gat', '103gat',
'104gat']
```

A graph visualization of the same has also been implemented. The partitions are represented with two different colors. The number of edges crossing from one group to another can easily be visualised and concluded that the cutsize has drastically been reduced.

Before Partitioning (Cutsizes = 35):



After Partitioning (Cutsizes = 12):



Thus, partitioning using KL algorithm has been successfully implemented.

Floorplanning

Simulated Annealing + Stockmeyer Algorithm

Chip planning, Floorplanning and Placement phases are very crucial in overall physical design cycle. It is due to the fact, that an ill-floorplanned layout cannot be improved by high quality routing. In other words, the overall quality of the layout, in terms of area and performance is mainly determined in the chip planning, floorplanning and placement phases. As this is definitely NP hard problem, many researchers have suggested heuristics and metaheuristic algorithms to solve the VLSI floorplan problem. The floorplan representation is an important impact on the complexity and search space of the floorplan design.

Simulated Annealing and Stockmeyer algorithms (*D.F. Wong and C.L. Liu*) are being used in this program wherein the temperature is set very high and one of the moves is selected randomly. After the circuit partitioning step, the area occupied by each block is estimated depending upon the number of inputs. The annealing function generates an initial polish expression and calculates the cost of the circuit. Depending upon that, the temperature is set high. The cost is calculated based upon the area occupied and wirelength. If the cost is lower than the previous move, the polish expression is saved and the iteration continues. This is repeated till the temperature falls to 0.1. Since this is a heuristic algorithm, the output may not be the same for every execution. Hence, a lot of iterations are required to find the best polish expression.

Algorithm SIMULATED-ANNEALING

```
begin
  temp = INIT-TEMP;
  place = INIT-PLACEMENT;
  while (temp > FINAL-TEMP) do
    while (inner_loop_criterion = FALSE) do
      new_place = PERTURB(place);
       $\Delta C = \text{COST}(\text{new\_place}) - \text{COST}(\text{place});$ 
      if ( $\Delta C < 0$ ) then
        place = new_place;
      else if (RANDOM(0, 1) >  $e^{\frac{\Delta C}{\text{temp}}}$ ) then
        place = new_place;
    temp = SCHEDULE(temp);
end.
```

Functions:

- **move1():**
 - **Inputs:** Polish expression.
 - **Output:** Polish expression with 2 adjacent operands swapped.
- **move2():**
 - **Inputs:** Polish expression.
 - **Outputs:** Polish expression after complementing the entire chain.
- **move3():**
 - **Inputs:** Polish expression.
 - **Outputs:** Polish expression with adjacent operator and operand swapped.
- **move():**
 - **Inputs:** Polish expression.
 - **Outputs:** Selects a random move. Returns Polish expression.
- **normality_prop():**
 - **Inputs:** Polish expression.
 - **Outputs:** Bool value after checking whether the polish expression satisfies the normality property (NPE)
- **balloting_prop():**
 - **Inputs:** Polish expression.
 - **Outputs:** Bool value after checking whether the polish expression satisfies the balloting property.
- **wirelength():**
 - **Inputs:** Adjacency matrix, block dimensions, Node coordinates.
 - **Outputs:** Wirelength.
- **cost_func():**
 - **Inputs:** Polish expression, block dimensions, adjacency matrix.
 - **Outputs:** Cost, size, area occupied, bottom left coordinates of every block.
- **annealing():**
 - **Inputs:** Adjacency matrix, blocks names, block dimensions, node list.
 - **Outputs:** Best polish expression, best area, best block coordinates, best size, final temperature.

Inputs:

- ISCAS-85 netlist file.
 - Adjacency Matrix generated from the parsing program.
 - Nodes from the parsing program.
 - inputs.txt generated from the parsing program.
 - outputs.txt generated from the parsing program.

Outputs:

- Initial and Best Polish expression.
- Best size and area.
- Initial and Final temperatures.
- The initial and final floorplan images.
- Cost vs Iterations plot.
- **gates.txt**: Text file containing dictionary of all the gates / blocks along with their coordinates.
- **inputs_coord.txt**: Text file containing dictionary of all the input gates / blocks along with their coordinates.
- **outputs_coord.txt**: Text file containing dictionary of all the output gates / blocks along with their coordinates.
- **floorplan_coord.txt**: Text file containing dictionary of all the input, output and all other gates / blocks along with their coordinates.

The polish expression must be a normalized polish expression, i.e. there are no consecutive operators of the same type in the polish expression. Each block appears exactly once in the PE. Balloting property is checked so that the number of operands is larger than the number of operators at all positions in the PE. The PE corresponding to the least area computed in these iterations is chosen as the best PE.

If the cost is less than previous cost, this arrangement is accepted and coordinates of cells are changed accordingly. If the cost is greater than the previous cost, this perturbation is accepted with a probability which decreases with decrease in temperature.

Probability of accepting perturbation is:

$$p = e^{\delta T}$$

where

δ : Difference between present cost and previous cost.

T : Present temperature.

p : Probability of accepting move with increased cost.

The following images shows the outputs of the *floorplanning.py* for the Fast Adder Circuit (74283.isc).

```
Enter the name of the file containing the required netlist
74283.isc

Initial Polish Expression ['26', '32', 'V', '38', 'V', '41', 'V', '49', 'V', '53', 'V',
'61', 'V', '66', 'V', '72', 'V', '78', 'V', '79', 'V', '80', 'V', '81', 'V', '82', 'V', '83',
'V', '84', 'V', '85', 'V', '86', 'V', '87', 'V', '88', 'H', '89', 'H', '90', '91', 'H', 'V',
'92', 'H', '93', 'H', '95', 'H', '94', 'H', '96', 'H', '97', 'H', '98', 'H', '99', 'H', '100',
'H', '101', 'H', '102', 'V', '103', 'H', '104', 'H']

Initial Size 2508 = 66x38

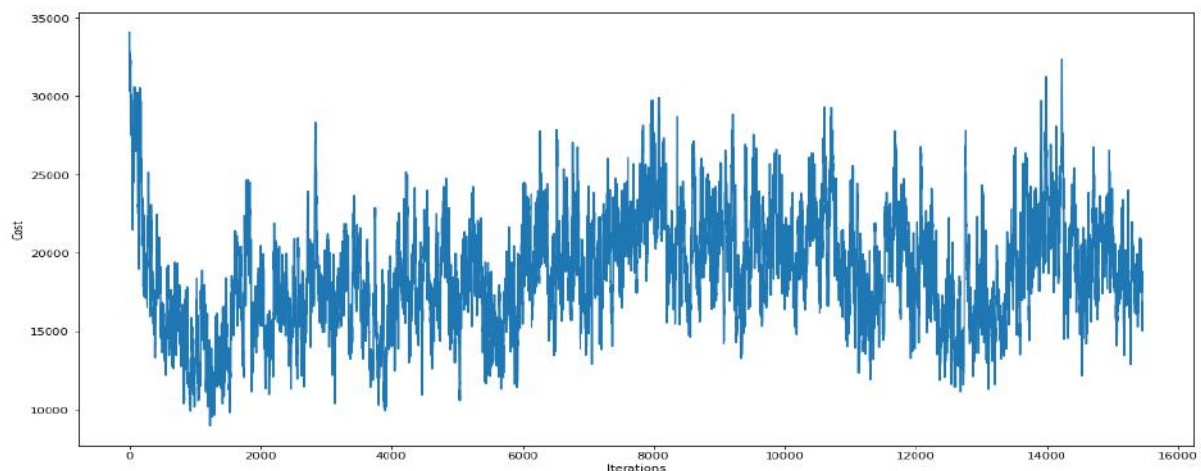
Initial Temperature calculated: 4053.8814579203977

Best Polish Expression: ['53', '49', '41', '72', 'H', 'V', '32', 'V', '61', '38', 'V', 'H',
'66', 'H', '78', 'H', 'V', '26', 'V', '79', '83', 'H', '85', '84', 'H', '80', 'V', '81', '97',
'V', 'H', 'V', '86', 'V', 'H', '82', 'V', '91', '89', '94', 'V', 'H', '92', '87', 'H', 'V',
'90', '98', '93', 'H', 'V', '103', 'H', '100', 'V', 'H', '99', '88', '102', 'H', 'V', '95',
'V', 'H', '104', '101', 'V', 'H', 'V', '96', 'H']

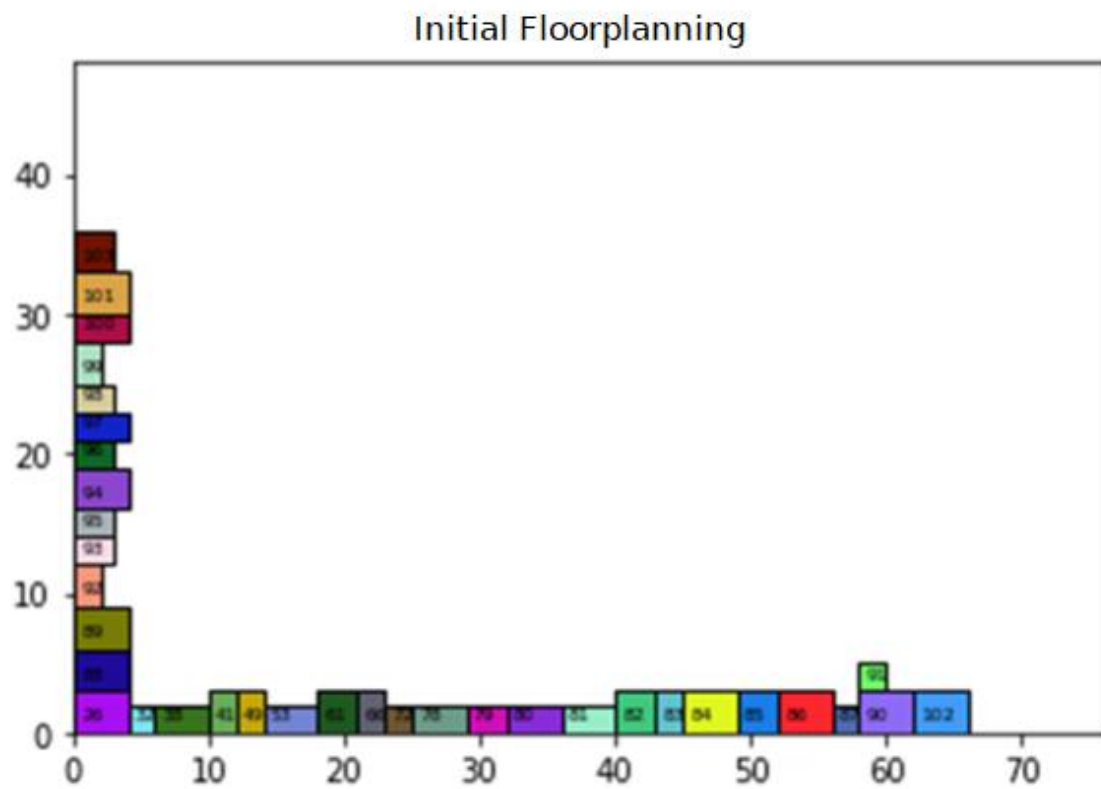
Best Size: 696 = 29x23

Final Temperature: 0.05057018501591647
```

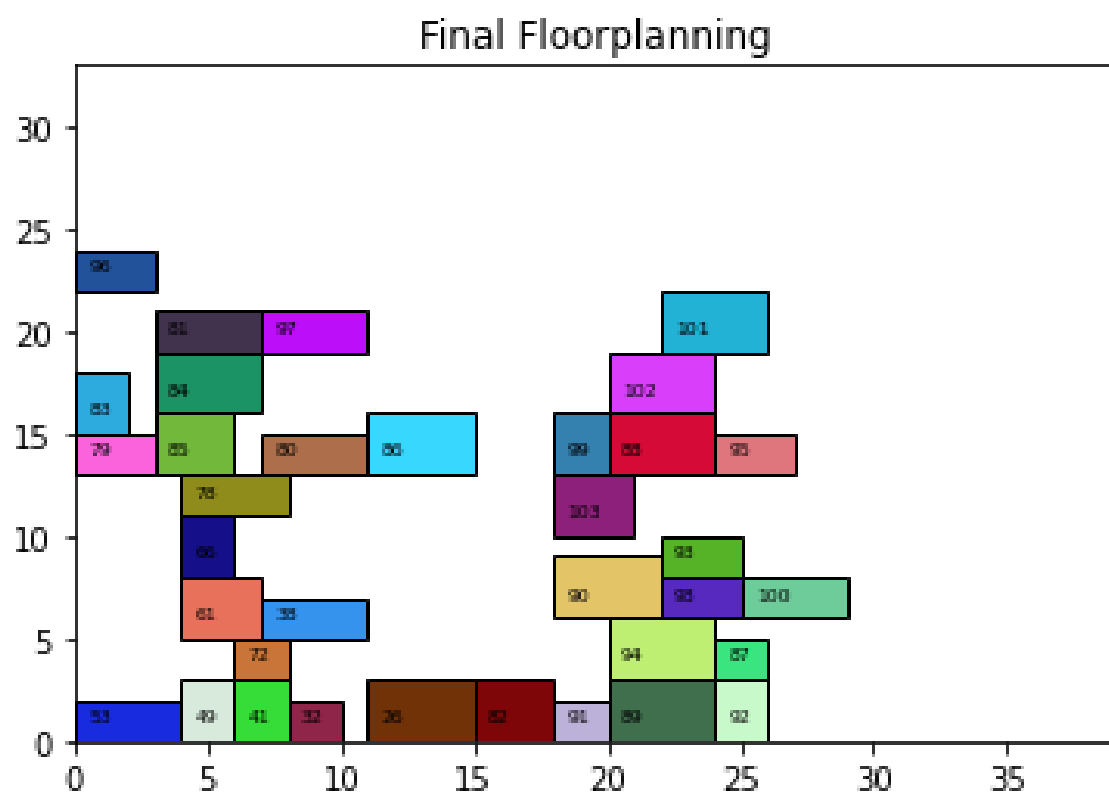
Cost vs Iterations plot:



The initial floorplan:



The final floorplan:



Placement

Simulated Annealing Algorithm

Placement is a key step in physical design cycle. A poor placement consumes larger areas, and results in performance degradation. It generally leads to a difficult or sometimes impossible routing task. Placement phase is very crucial in overall physical design cycle. It is due to the fact, that an ill-placed layout cannot be improved by high quality routing. In other words, the overall quality of the layout, in terms of area and performance is mainly determined in the placement phase.

The same Simulated Annealing algorithm is used for placement. A checker board model, with each block of size of 10 x 10 is used for placing the gates / cells.

Functions:

- **interconnects_length ():**
 - **Inputs:** Wire connections, block coordinates.
 - **Output:** Total wirelength.
- **pinout_assign ():**
 - **Inputs:** Chip size.
 - **Output:** Assigns Pins to the blocks
- **moving_block ():**
 - **Inputs:** Block coordinates (occupied and non-occupied).
 - **Outputs:** Selects a cell randomly and moves it to a block which is vacant.
- **exchange_blocks ():**
 - **Inputs:** Block coordinates (occupied and non-occupied).
 - **Outputs:** Selects two cells randomly and their locations are exchanged.
- **boundaries ():**
 - **Inputs:** Block coordinates.
 - **Outputs:** The maximum values of x and y coordinates of cells in the entire graph.

Inputs:

- **gates.txt**: Text file containing dictionary of all the gates / blocks along with their coordinates after floorplanning.
- **inputs_coord.txt**: Text file containing dictionary of all the input gates / blocks along with their coordinates after floorplanning.
- **outputs_coord.txt**: Text file containing dictionary of all the output gates / blocks along with their coordinates after floorplanning.
- **floorplan_coord.txt**: Text file containing dictionary of all the input, output and all other gates / blocks along with their coordinates after floorplanning.

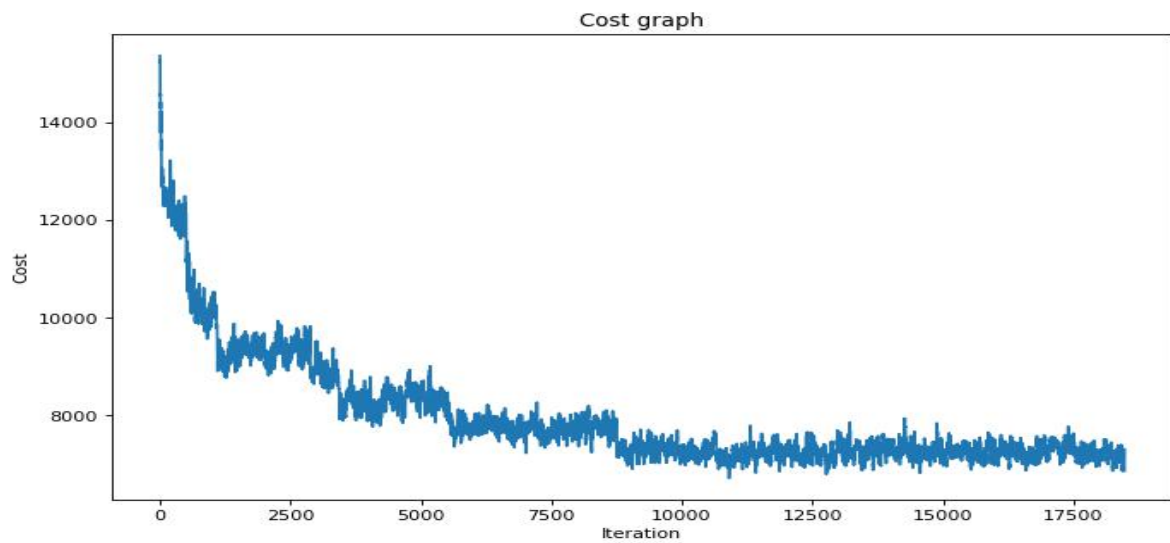
Outputs:

- Final Placement.
- Initial and Final cost.
- Cost vs Iterations plot.
- **pinouts.txt**: Text file containing dictionary of all the pinout coordinates after pin assignment.
- **placement_coord.txt**: Text file containing dictionary of all the input, output and all other gates / blocks along with their coordinates.

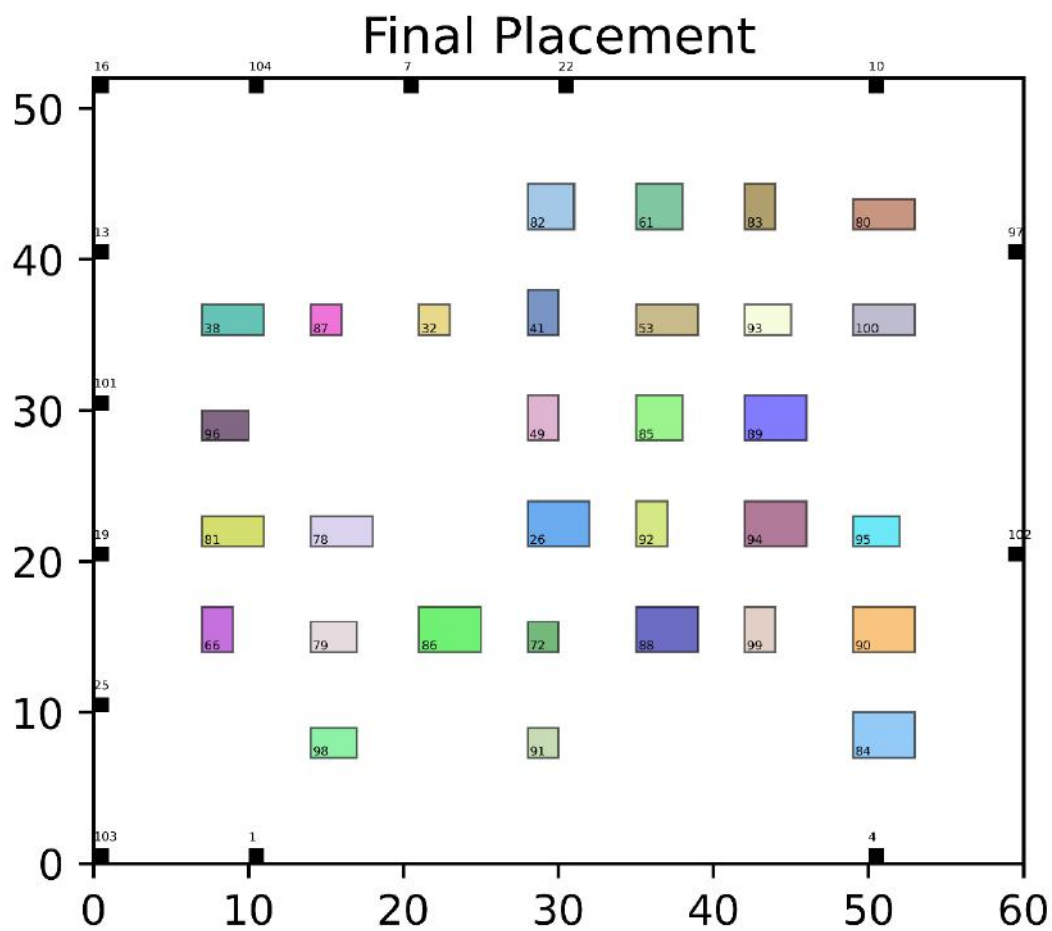
The following images shows the outputs of the *placement.py* for the Fast Adder Circuit (74283.isc).

```
Initial cost= 15324
Final cost= 6719
The Final Cost is 56.153745758287656 % less than that of Initial cost
```

Cost vs Iterations plot:



The final placement:



Routing

Lee's Algorithm

The exact locations of circuit blocks and pins are determined from the placement part. The locations are then used for the last step of the design process, i.e. Routing. Nets are routed within the routing regions and they must not short-circuit. The objective is to find the paths of the nets as well as minimize the wirelength.

Lee's algorithm is used in this process. It is simple and its guarantees finding an optimal solution if one exists. The exploration phase of Lee's algorithm is an improved version of the breadth-first search. The search can be visualized as a wave propagating from the source. The source is labelled '0' and the wavefront propagates to all the unblocked vertices adjacent to the source. Every unblocked vertex adjacent to the source is marked with a label '1'. Then, every unblocked vertex adjacent to vertices with a label '1' is marked with a label '2', and so on. This process continues until the target vertex is reached or no further expansion of the wave can be carried out. Due to the breadth-first nature of the search, Lee's maze router is guaranteed to find a path between the source and target, if one exists. In addition, it is guaranteed to be the shortest path between the vertices.

Lee's Algorithm starts with two vertices and checks whether the vertex is blocked or unblocked. The algorithm uses an array L , where denotes the distance from the source to the vertex. This array will be used in the retracing to join the vertices to form a path P , which is the output of the Lee's Algorithm. Two linked lists $plist$ (Propagation list) and $nlist$ (Neighbour list) are used to keep track of the vertices on the wavefront and their neighbour vertices respectively. These two lists are always retrieved from tail to head. We assume that the neighbours of a vertex are visited in counter-clockwise order, that is top, left, bottom and then right.

The Lee algorithm's pseudo code is represented in the next figure.

Algorithm LEE-ROUTER (B, s, t, P)
 input: B, s, t
 output: P
begin
 $plist = s$;
 $nlist = \phi$;
 $temp = 1$;
 $path_exists = FALSE$;
 while $plist \neq \phi$ **do**
 for each vertex v_i **in** $plist$ **do**
 for each vertex v_j **neighboring** v_i **do**
 if $B[v_j] = UNBLOCKED$ **then**
 $L[v_j] = temp$;
 $INSERT(v_j, nlist)$;
 if $v_j = t$ **then**
 $path_exists = TRUE$;
 exit while;
 $temp = temp + 1$;
 $plist = nlist$;
 $nlist = \phi$;
 if $path_exists = TRUE$ **then** $RETRACE(L, P)$;
 else $path$ does not exist;
end.

Functions:

- **wave_propagation ():**
 - **Inputs:** Source, target.
 - **Output:** Marks the source from source to target.
- **retrace ():**
 - **Inputs:** Source, target.
 - **Output:** Path after retracing from target to source.
- **layer_separation ():**
 - **Inputs:** Path, target, metal 1, metal 2, metal contact.
 - **Output:** Metal 1, metal 2, metal contact after checking whether these two metals touch each other or not.
- **routing ():**
 - **Inputs:** Connections between cells, targets, sources, metal 1, metal 2 and contact.
 - **Outputs:** Metal 1, metal 2, metal contact and path.
- **boundaries ():**
 - **Inputs:** Block coordinates.
 - **Outputs:** The maximum values of x and y coordinates of cells in the entire graph.

Inputs:

- **placement_coord.txt**: Text file containing dictionary of all the gates / blocks along with their coordinates after floorplanning.
- **gates.txt**: Text file containing dictionary of all the input gates / blocks along with their coordinates after floorplanning.
- **wires.txt**: Text file containing dictionary of all the output gates / blocks along with their coordinates after floorplanning.

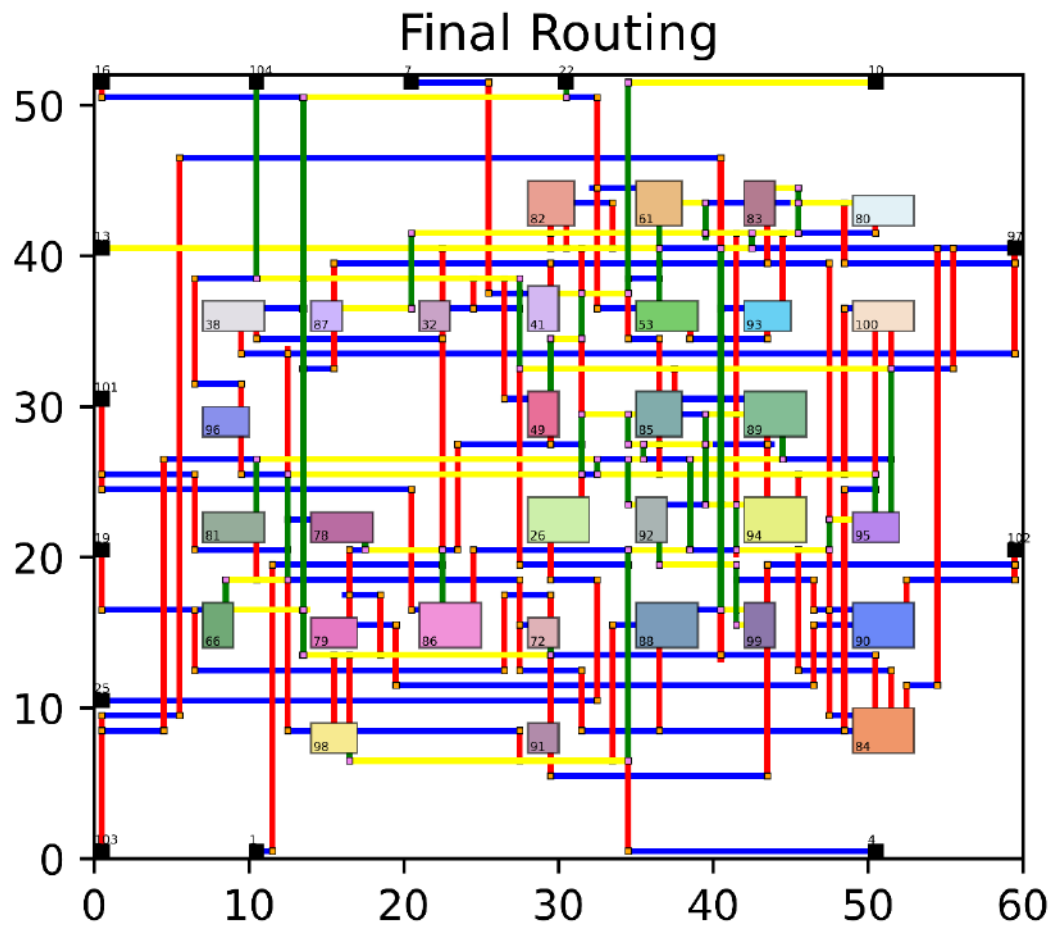
Outputs:

- Final successful routing visual.
- **pins.txt**: Text file containing dictionary of all the pinout coordinates after pin assignment.

The following images shows the outputs of the *routing.py* for the Fast Adder Circuit (74283.isc).

```
Routing is completed
Total cost = 5506
Time taken= 12.73613649999993 seconds
```


The final successful routing visual:

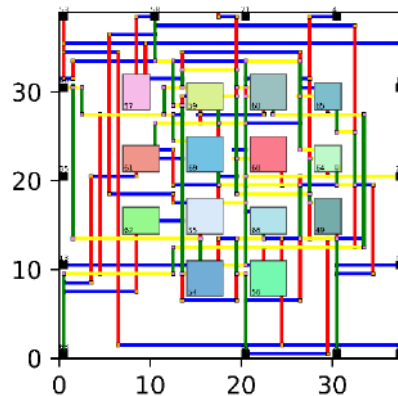


Connections in various layers are represented by distinct colours as follows:

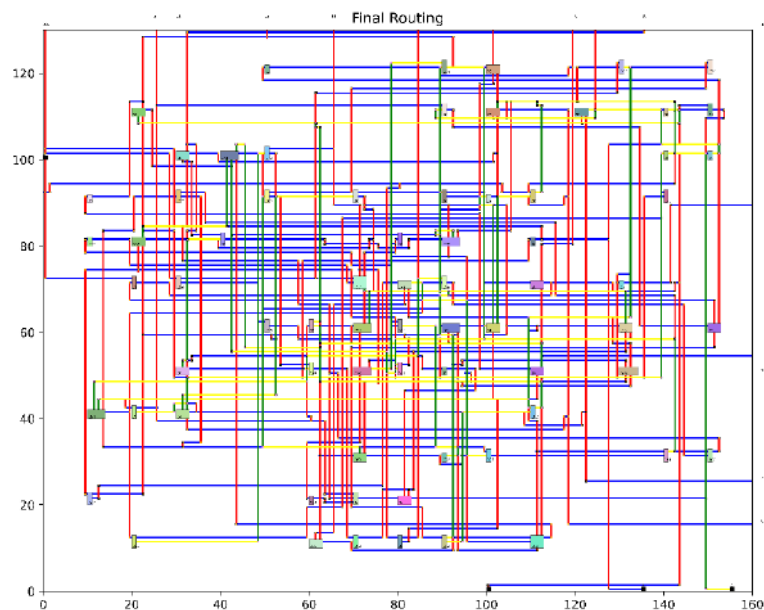
- Layer 1: Blue
- Layer 2: Red
- Layer 3: Yellow
- Layer 4: Green

Other Netlists

74182 Carry Look-Ahead Circuit:



74181 4-Bit ALU/Function Generator:



References

1. "Algorithms for VLSI Physical Design Automation", Naveed Sherwani, Springer 1998.
2. "Simulated Annealing for VLSI Design", D. F. Wong, H. W. Leong, C. L. Liu, Springer 1988.