EC704 - VLSI Design Automation
Project Documentation (First Evaluation)

4-Bit Wallace Multiplier

Yashwanth G (16EC154)

Department of Electronics and Communication Engineering
National Institute of Technology Karnataka, Surathkal

24-04-2020

Contents

1. Summary

The 4 bit Wallace multiplier has 2 four bit inputs and one 8 bit output. The initial stage of the project deals with the partitioning of the gates/nodes functioning as part of the circuit. The Wallace multiplier is designed into a gate level list consisting of only the gates and interconnections. This is written into the .isc file. This is an intermediate file that is generated by the software for any code written. This is then customized to fit into the fabric through various processes.

For the Wallace multiplier, the .isc file is written from scratch based on the working of the circuit. We have 144 nodes present in the file (including the inputs and the outputs). The .isc file consists of a gate level net-list constructing using only AND, OR and NOT gates.

The 144 nodes present in the .isc file now undergoes the entire design automation procedures in order to be fabricated. In the program the .isc file is imported from the google drive into the python work space. The first step in the partitioning of the nodes. Both simulated annealing and Kernighan Lin algorithms for partitioning of the nodes.

They are both done using python. The .isc file is first processed through a parser function which returns the adjacency matrix for partitioning.

2. Package Dependencies

There are a few python packages that are imported in the project for various purposes.

- **numpy** - It is used for handling large multi-dimensional array. It is used in the project for handling the various matrices and node interchanges between functions.
- **networkx** - It is used to convert the matrices into graphs and for dealing with large graphs. It is helpful to map the interconnections between nodes using this package.
- **matplotlib** - this package allows visual representation of data. This project using the circular layout function of this package to show the interconnect between partitions.
- **time** - Time package allows us to measure the time taken for our procedures to run and generate the respective outputs. It easily lets us compare the timing between two different algorithms.

3. Partitioning Algorithms

From the input .isc file that contains the complete information about the Wallace multiplier in gate level netlist, we aim to partition the nodes present in the circuit. This is done using two algorithms. The Kernighan Lin (KL) algorithm and the simulated annealing algorithm. These algorithms cannot work on the .isc file directly. Therefore, we need a parser function to convert the .isc file data into an adjacency matrix. We can perform the algorithms on this matrix.

3.1 Parser

The parser used is an ISCAS (International Symposium on Circuits and Systems) parser model. This parser takes the .isc file as the input and generates the adjacency matrix. In the program this adjacency matrix is refered to the variable **adj_mx**. The path for the .isc file is an input through **file_root**.

3.1.1 Calling the parser

Use the **with** and **as** functions of python, open the file in read mode and store it into a variable. This is then passed into the function *parser_iscas(input)*. The output that is given by this function is the adjacency matrix. An example of reading the file from file root and passing it into the function is shown below.

```
with open (file_root, "r") as myfile:
    file=myfile.read()

adj_mx = parse_iscas(file)
```

INPUT : .isc file stored in path **file_root**
OUTPUT: Adjacency matrix **adj_mx**

3.2 Plotting

The plots in the program are done using the **matplotlib** package present in python. Since we are using the plots to indicate the partitioned graphs. The functionality of the program is limited to plotting these graphs alone.

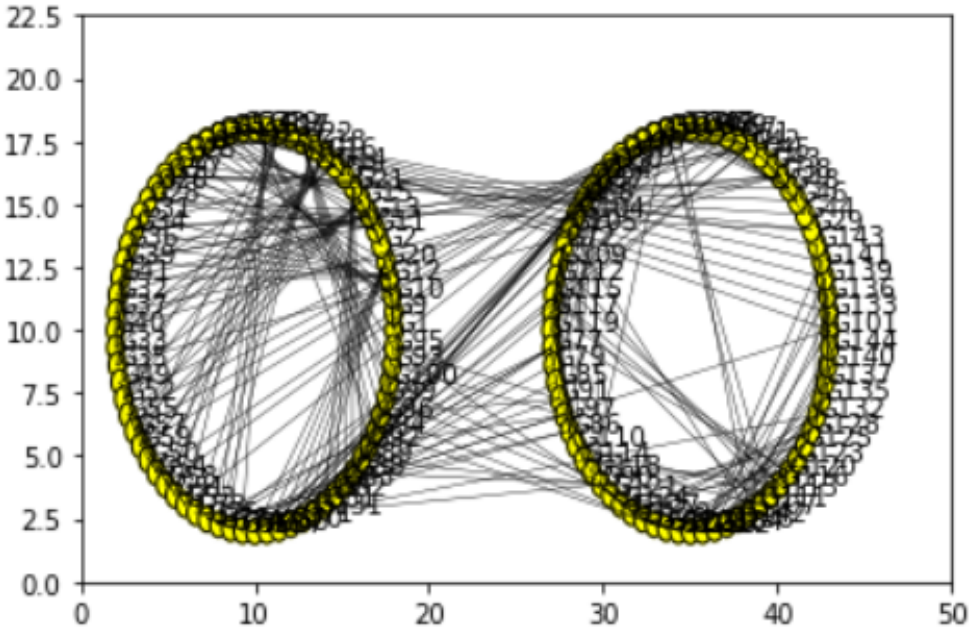
`Draw_Partitioned_Graph(ad_mx, part_a, part_b)`

This function takes in three inputs and plots the partitioned graph.

- **ad_mx** This is the adjacency matrix that is generated from the parser. This matrix remains constant for one .isc file.
- **part_a** Since the algorithms split the nodes into 2 groups, each group is passed individually to the plotting function. **part_a** is the first half of the nodes.
- **part_b** This contains the second half of the nodes. This depends on the partitioning of the graph.

This function does not return any value. It outputs the graph in a visual form.

Figure 3.1: Example plot generated using the Draw_Partitioned_Graph function.



3.3 Classes and methods for graphs

We use graphs extensively for these algorithms. Therefore, it is required to have some specific classes for creating and manipulating graphs.

3.3.1 Edge

This class is for an edge. The functions in this class are used for connecting the edge to a vertex.

- *set_left_vertex(self, left_vrtx)* Connects the left end of the edge to the vertex passed as input.
- *set_right_vertex(self, right_vrtx)* Connects the right end of the edge to the vertex passed as input.

3.3.2 Vertex

This class defines each vertex.

- *get_id()* Returns the id of the vertex.
- *get_edges()* Returns the edges the vertex is connected to.
- *get_group()* Returns the group 'A' or 'B' the vertex is a part of.
- *set_group(group)* Adds the vertex to the group inputted.
- *add_edge(edge)* Adds an edge to the vertex.
- *get_cost()* Returns the number of connections the vertex has.

3.3.3 Graph

- *__creat_links()* Called in the **__init__** it connects the edges and the vertices as passed to the class initialisation.
- *create_random_groups()* It sets the vertices of the graph to randomly group 'A' or group 'B'. This is necassary to start the KL algorithm.
- *get_random_groups()* It returns the two random groups of the vertices.

- *get_ groups()* It returns the groups of the vertices during the running of the algorithm. These groups may or may not be random.
- *get_ cut_ size()* Returns the number of interconnections between the two groups.
- *get_ edges()* Returns the edges in the graph.
- *get_ vertexs()* Returns the vertices in the graph.

3.4 Kernighan Lin Algorithm

The input to the KL algorithm is a graph that is derived from the adjacency matrix of the circuit. This is the adjacency matrix obtained from the .isc file. This matrix is converted into an undirected graph where all connections between the gates/nodes are indicated through an edge and the nodes themselves are vertices in the graph. To perform the necessary operations on the adjacency matrix, the **netrworkx** package of python is used to form and manipulate the data. Since each edge and vertex are independent entities there are classes created for them individually. As and when required in the graph, edges and vertices are formed.

The goal of the KL algorithm is to partition the graph into two sets with equal or almost equal sizes with minimum number of interconnections between them. Therefore the algorithm aims to return two sets by recursively swapping the nodes between the two sets to find the minimum cut size (Number of interconnections between the two partitions). If the graph has n vertices the each iteration runs for $O(n^2 log(n))$.

The KL algorithm works on a graph. Therefore, we need a method to convert the adjacency matrix into a graph.

```
Adj_mx_to_graph(adj_mx)
```

This method converts the adjacency matrix into a graph. It calls the **Graph** class discussed above to form a graph out of the edges and vertices the method aggregates.

To perform the KL algorithm easily there is a class written for it which can be invoked directly to perform the KL algorithm. The class is called **KernighanLin** in the program. We need to create a method of the class to begin the algorithm. **kl** is the object created and the **graph_KL** is obtained from the *Adj_ mx_ to_ graph* method.

```
kl = KernighanLin(graph_KL)
```

The methods of this class are:

- *start()* This function is where the main algorithm runs. Since it is a method of a class, it can only be invoked after creating the object. It returns the two groups of vertices after the KL algorithm is performed. An example function call is shown below.

```
group_a,group_b=kl.start()
```

- *single_swaps()* It determines the best heuristic for a swap. This heuristic is determined by the costs of the vertices and edges. It is called in the *start()* method.

- *get_nominal_cut_size()* Returns the cut size. It is the number of interconnections between group 'A' anbd group 'B'.

- *do_swap((vertice_a, vertice_b))* It changes the group name of the vertices thereby interchanging them in the graph. It taken two vertices as inputs and returns no value.

3.4.1 Outputs

There are 144 nodes inputted into the algorithm. The KL algorithm divides them into 2 groups and outputs the minimum interconnection groups. The partitioning before and after is as follows:

```
The initial Partioning is as follows:

PARTITION A= [1, 9, 10, 12, 20, 2, 11, 13, 15, 21, 3, 14, 16, 18, 22,
4, 17, 19, 23, 24, 5,6, 7, 8, 25, 27, 30, 26, 28, 31, 34, 36, 39, 41,
32, 37, 40, 33, 35, 49, 52, 55, 57, 59,50, 54, 58, 51, 53, 73, 75, 78,
74, 76, 63, 64, 65, 70, 71, 131, 81, 82, 83, 88, 89, 92,94, 96, 99,
100, 93, 95]

PARTITION B=[101, 133, 136, 139, 141, 143, 29, 44, 46, 48, 38, 43, 45,
42, 61, 67, 69, 47,103, 106, 108, 56, 62, 66, 60, 80, 84, 87, 68, 104,
105, 72, 109, 112, 115, 117, 119, 77,79, 85, 91, 97, 86, 110, 114, 118,
90, 121, 124, 127, 129, 98, 122, 126, 130, 102, 134,138, 142, 107, 111,
113, 116, 120, 123, 125, 128, 132, 135, 137, 140, 144]
```

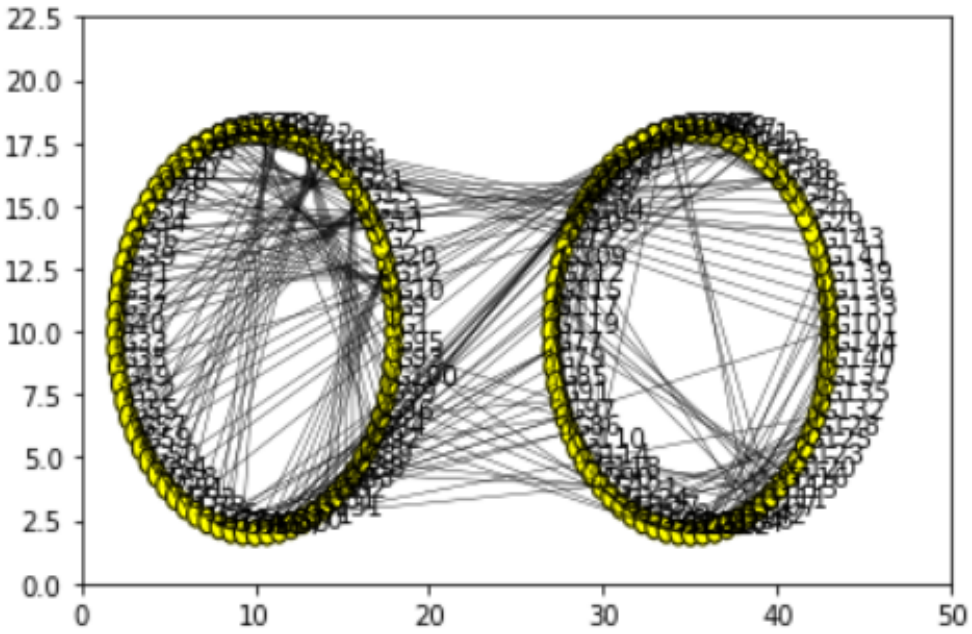
The Partioning after applying KL Algorithm is as follows:

PARTITION A=[1, 9, 10, 12, 2, 11, 13, 15, 3, 14, 16, 18, 22, 4, 17, 19, 23, 5, 6, 7, 25,27, 26, 28, 31, 34, 36, 39, 41, 32, 37, 40, 33, 35, 49, 52, 55, 57, 59, 50, 54, 58, 51, 53,73, 75, 78, 74, 76, 64, 92, 94, 96, 99, 100, 93, 95, 101, 29, 44, 46, 38, 43, 45, 42, 61,69, 47, 56, 66, 91, 97]

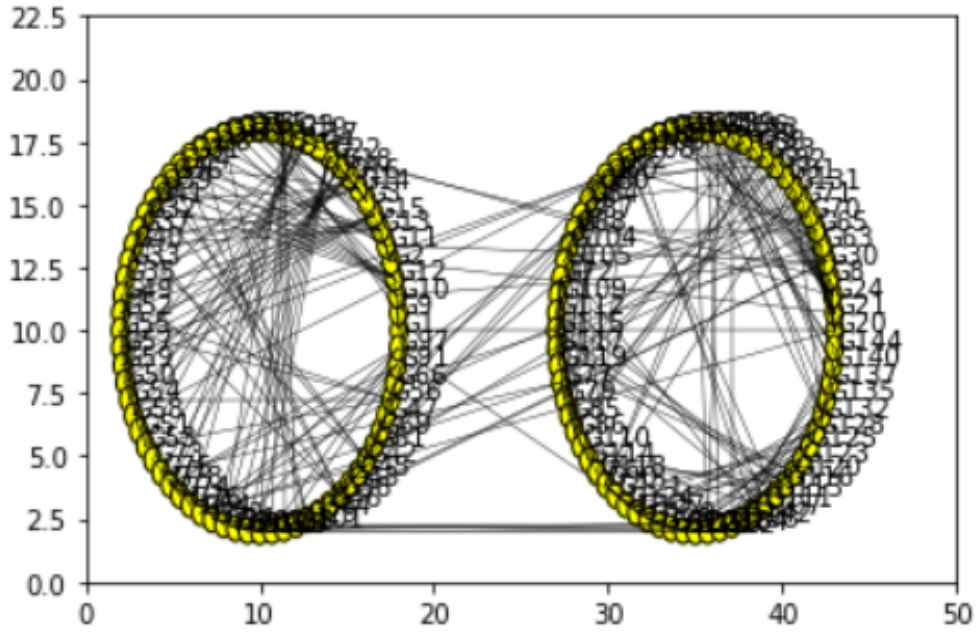
PARTITION B=[20, 21, 24, 8, 30, 63, 65, 70, 71, 131, 81, 82, 83, 88, 89, 133, 136, 139,141, 143, 48, 67, 103, 106, 108, 62, 60, 80, 84, 87, 68, 104, 105, 72, 109, 112, 115, 117,119, 77, 79, 85, 86, 110, 114, 118, 90, 121, 124, 127, 129, 98, 122, 126, 130, 102, 134,138, 142, 107, 111, 113, 116, 120, 123, 125, 128, 132, 135, 137, 140, 144]

We can plot this data before and after the running of the KL algorithm. The graphs obtained are:

The Partition graph before applying KL Algorithm:



The Partition graph after applying KL Algorithm:



3.4.2 Results

The aim of the algorithm is to minimize the cutsize between the two groups. We display the cutsizes at both times.

The No of Cuts before applying KL Algorithm:65
The Cut Size After applying KL Algorithm is:31

The other parameter that is considered here is the time taken for the algorithm to run completely. We use the **time** package in python to record the times before and after the algorithm is run. This difference gives the time taken to run. The time recorded to run KL algorithm is:

The Total Time elapsed:0.4722099304199219 Seconds

3.5 Simulated Annealing

Simulated annealing is also carried out using a class of its own - **Simulated_Annealing**. To run the algorithm the object of this class must be first created. This class has its own methods to run the simulated annealing algorithm. The class again takes the input as a graph. Therefore the adjacency matrix must be converted into a graph for this algorithm to work on. The functions in the Simulated annealing class are:

- *score(part_a,part_b)* It returns the cutsize of the two groups. Initially these groups are ramdomly generated by the **Graph** class.
- *select(part)* It returns a random vertex from the chosen group passed as input. Here the **numpy** package is used for randomly selecting a vertex.
- *exchange(vertice_a,vertice_b)* It exchanges the two vertices from one group to another. It returns two trial groups 'A' and 'B'. They are not the same as the original group.
- *Move_part(vertice_a,vertice_b)* Interchanges the vertices' groups. This reflects in the parts present used in the algorithm.
- *start()* This functions starts the SA algorithm. It returns no value and takes in no inputs. The timer is run for this function to get the time of the SA algorithm.
- *Draw_cost_plot()* It is used to graphically represent the cost vs iterations graph. It is a simple graph that correlates the cost and iterations on a 2D scale.

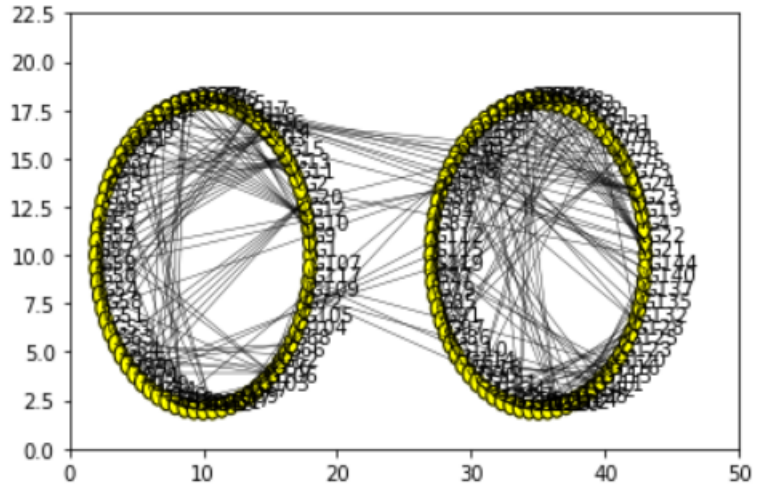
3.5.1 Results

On running the program for the .isc file present. The results generated are as follows.

```
The Cut Size before Simulated Annealing:65

Performing Simulated Annealing Algorithm..... Please wait!!!

Simulated Annealing completed Successfully!!!

The total time elapsed:6.711315870285034 Seconds

The Cut Size after Simulated Annealing:24
```

Graphically we can represent the partitions from the previous plotting. The cut size here is less but the time taken for the algorithm to run is higher. The cost vs iterations plot shows the minimization of the cost over time.

The Partitioned Graph after Simulated Annealing:



The Cost vs No Iterations Graph: