OPEN SOURCE PYTHON LIBRARY FOR HARDWARE DESIGN

Report submitted to the SASTRA Deemed to be University
in partial fulfillment of the requirements
for the award of the degree of

BCSCCS801: **PROJECT WORK**

Submitted by

ARAVIND .B
(Reg. No.: 121003027,B.Tech Computer Science and Engineering)

June 2021

SCHOOL OF COMPUTING

THANJAVUR, TAMIL NADU, INDIA – 613 401

SCHOOL OF COMPUTING

THANJAVUR – 613 401

Bonafide Certificate

This is to certify that the project report titled "**Open Source Python Library For Hardware Design** " submitted in partial fulfillment of the requirements for the award of the degree of B. Tech. Computer Science and Engineering to the SASTRA Deemed to be University, is a bona-fide record of the work done by **Mr. Aravind .B**(Reg. No. 121003027) during the final semester of the academic year 2020-21, in the **School of Computing**, under my supervision. This report has not formed the basis for the award of any degree, diploma, associateship, fellowship or other similar title to any candidate of any University.

DAVID CASTELLS RUFAS - DNI 39355627M
Firmado digitalmente por DAVID CASTELLS RUFAS - DNI 39355627M
Fecha: 2021.06.28 10:46:48 +02'00'

Signature of Project Supervisor :

Name with Affiliation : Mr. David Castells, Professor,
Universitat Autònoma de Barcelona

Date : 28/06/2021

Project *Viva Voce* held on _____

Examiner 1 **Examiner 2**

SCHOOL OF COMPUTING

THANJAVUR – 613 401

Declaration

I declare that the project report titled "**Open Source Python Library For Hardware Design**" submitted by me is an original work done by me under the guidance of **Mr. David Castells, Professor,Universitat Autònoma de Barcelona** during the final semester of the academic year 2020-21, in the **School of Computing**. The work is original and wherever I have used materials from other sources, I have given due credit and cited them in the text of the report. This report has not formed the basis for the award of any degree, diploma, associate-ship, fellowship or other similar title to any candidate of any University.

Signature of the candidate(s) : B. Aravind

Name of the candidate(s) : Aravind B

Date : 28/06/2021

Acknowledgements

I would like to thank our Honorable Chancellor **Prof. R. Sethuraman** for providing us with an opportunity and the necessary infrastructure for carrying out this project as a part of our curriculum.

I would like to thank our Honorable Vice-Chancellor **Dr. S. Vaidhyasubramaniam** and **Dr. S. Swaminathan,** Dean**,** Planning & Development, for the encouragement and strategic support at every step of our college life.

I extend our sincere thanks to **Dr. R. Chandramouli,** Registrar, SASTRA Deemed to be University for providing the opportunity to pursue this project.

I extend our heartfelt thanks to **Dr. A. Umamakeswari**, Dean, School of Computing, **Dr. R. Muthaiah**, Associate Dean, Department of Information Technology and Information & Communication Technology and **Dr. V. S. Shankar Sriram**, Associate Dean, Department of Computer Science and Engineering for their motivation and support offered in materializing this project.

My guide **Mr. David Castells, Professor,Universitat Autònoma de Barcelona**, was the driving force behind this whole idea from the start. His deep insight in the field and invaluable suggestions helped me in making progress throughout our project work. I also thank the project review panel members for their valuable comments and insights which made this project better.

I would like to extend our gratitude to all the teaching and non-teaching faculties of the School of Computing who have either directly or indirectly helped us in the completion of the project.

I gratefully acknowledge all the contributions and encouragement from my family and friends resulting in the successful completion of this project. I thank you all for providing me an opportunity to showcase my skills through the project.

List of Figures

ABBREVIATIONS

FGPA	Field Programmable Gate Array
GUI	Graphical User Interface
HDL	Hardware Description Language
PY	Python
VHDL	Verilog Hardware Description Language

Abstract

Electronic devices are presently used in a lot of homes, industries and almost in every nook and corner of the world. Most of the devices are built based on circuits and circuit logic. Circuit designing and visualization is necessary before implementing them in real world applications. To perform this, programming languages like HDL and Verilog HDL (VHDL) are used. But the drawback is that simulation is a time consuming process and the software is a licensed one.

To overcome this Python is used for Hardware design which has richer programming semantics and also is much easier to compile. For visualization, the already existing packages do not give a good representation and hierarchy is not good. Hence a package for creating hardware design is built from scratch using some layout algorithms to give a beautiful, hierarchical and neat circuit with reduced number of crossovers.

Specific Contribution

- Providing hierarchical visualization for the circuits
- Reduce the crossover of circuits

Specific Learning

- Creating data structures for components of circuits
- Logic and Simulation for hardware designing

Table of Contents

CHAPTER 1

INTRODUCTION

Circuits play an important role in building electronic devices. Every diode, transistor and ic has a logic based on the circuits. To represent these circuits we need pictorial or graphical methods to give a good understanding about them. This representation is done from higher levels of abstraction like block diagrams to even simple circuit diagrams.

In modern industry, digital electronic circuits are commonly described with hardware description languages (HDL), which provide a convenient high-level of abstraction. However, logic diagrams are commonly used as a better method to explain the structure of the circuits. Refer Figure 1.1 for an example of a logic diagram. In the past, designers designed circuits by directly using diagrams. Now, it is preferred to use HDL for design entry, and generate the diagrams automatically from the code.

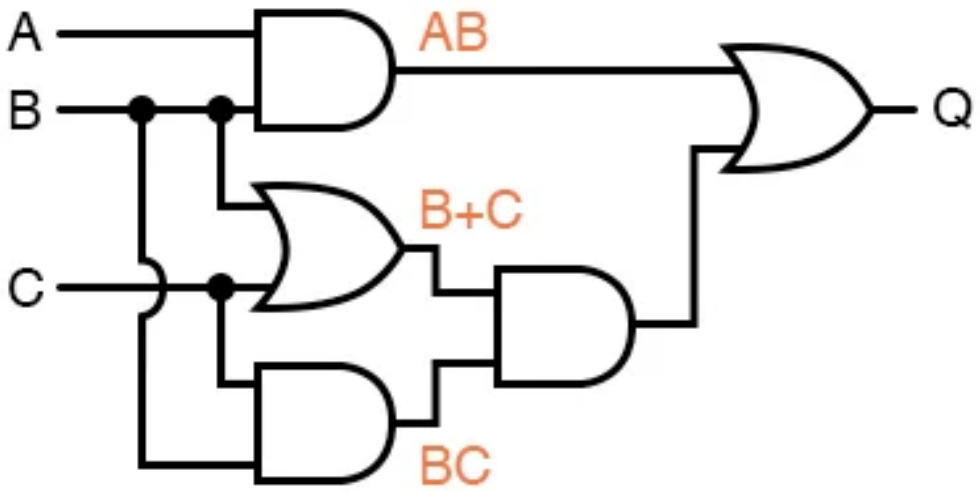

Fig. 1.1 Example of a logic diagram

It is cumbersome for doing any verification or simulation using HDL. Moreover it is a licensed software and is a complex tool as given by manufacturers. To overcome this Python is used for Hardware design which has richer programming semantics and also is much easier to compile. Moreover it is difficult to manipulate after generating the circuits in this manner.

The part where one has to verify the circuit is the most time consuming. In most of the softwares used for building digital circuits the time for designing is less than simulation testing and verification. Spent time is greater when HDL types of languages are used. The FGPAs are commercially ready for hardware emulation but are not collaborated with the algorithms based on HDL. Even though it's available over two decades, time consumption has not been reduced and insignificant when they are not integrated. The visualization is mostly needed to be done manually because of lack of integration techniques.

The reason for staying with HDL and VHDL is that these types of hardware definition languages have some set of unique and useful features for digital circuit logic. They are specialised and user requirements specific languages used in the field of FGPAs and simulation part of the digital circuit logic. One of the disadvantages of using these languages are they do not provide good visualization graphically.

The functionality provided by these languages is some sort of basic visualization which is good for smaller circuits. When the size of circuits goes on increasing and the number of elements are more, some complexity occurs and is not visually good. Too many crossings between elements and looping does not give a good schematic for users. These are the reasons for moving on to different languages from traditional HDL.

When circuits are represented visually, we need to create it in such a way that they are visually attractive and are easily understood. Too many connections between modules can create a state of confusion. A good design is to be implemented which overcomes the confusion state and gives a clear cut idea about the circuit. This also helps in testing circuits before building in realtime. There is a possibility that there are a number of errors encountered by testing it manually in the circuit built. Verification would become easier if everything is coded logically.

MOTIVATION

Simulating circuits helps us to save a lot of time before implementing them in real time. Most of the softwares already present is licensed and not affordable for everyone. Creating an Open Source package in Python overcomes the difficulties happening in previous methods. Also Python is a simple language which is easier to compile and has a lot of powerful programming semantics.

The most preferred language used for hardware design is HDL or even sometimes VHDL. But these languages are not powerful enough currently and Python can be the new alternative. But these are asynchronous in nature and very difficult to simulate and verify. It takes a lot of time to compile since it's going to be performing across various language domains.

Python is one of the easiest modern languages, which can be easily understood and useful for developing applications rapidly. For complex tasks, python provides us with various types of packages and features which makes the completion of tasks easier. Most of them worldwide would agree that python is simple in terms of producing software solutions. Also generally hardware development and simulation is a complex process. So bringing the part of implementing python for this hardware task would somewhat be effective and efficient.

This makes python the primary choice for the current problem statement. The existing packages which help in hardware design do not provide a good form of visualization. To produce a good visualization, suitable data structures for each component of the digital circuits should be created and a suitable algorithm for placing in a good hierarchy should be found out and be implemented. Also for implementing a good user interface and an interactive GUI, python provides the tkinter GUI library. For implementing these things, python will be the most suitable programming language, which can be used to create open source packages and easily imported and installed.

CHAPTER 2

OBJECTIVE

The main objective is to build a neat circuit with definite hierarchies present. Position of each and every object present in the logical circuit must be calculated and placed to give an attractive outcome.

The objects that should be taken care of while designing :

- Object or Gates location
- Orientation of the input and output parts of gates
- The wires and connecting element's path
- Apt space between the gates
- Size of gates

The above things should be taken care while designing. These are the main requirements for this problem statement. In order to achieve these requirements, some set of goals are to be created.

The goals for maximum readability are as follows:

- The flow of wires should be from left to right
- The first hierarchy should contain gates having inputs
- Minimize the path length
- Bending of wires should be minimal
- The crossovers of the wires should be minimized
- Gates should not overlap with each other
- Wires overlapping should be avoided

CHAPTER 3

METHODOLOGY

Algorithms and definitions need to be specified for this problem statement. Design specification is taken as input, and we have to design a schematic and readable circuit with certain conditions for feasibility. The schematic diagram is defined based on the number of modules, wires and terminals present in the logic circuit. These are basically enough to specify any digital circuit.

So let's define it as follows :

$$\text{Schematic diagram} = \{M,W,T\} ,$$

where M is a set of modules, W is a set of wires and T is a set of Terminals.

Further wires can also be defined as $W = \{T\}$, because each wire runs between 2 or more terminals.

And each terminal T_i can be defined as $T_i = \{\text{input} \mid \text{output} \mid \text{bidirectional}\}$.

Also each module has a certain number of terminals and a set of other modules connecting to it. Two modules are said to be connected if the output terminal of one module is linked with the input terminal of another module through a wire.

It can be defined as $\text{Conn}(M_i) = \{M_k \mid \exists\ W_m = \{T_a , T_b\} , T_a = \text{output} \in M_i , T_b = \text{input} \in M_k \}$.

3.1 Left to Right wires identification

The schematic diagram is built with making the flow of it from the left to right. For this data structure such as a directed graph is used. Each and every gate is represented as the vertex of the graph. The edges represent the flow of signal from one gate to the other. Acyclic graph should be formed if there are any cycles present in this structure.

The directed graph is for representing the same circuit, just for convenience to bring out some logic and definition which can be easier to understand. The terminals are also part of this directed graph and they are kept on specific points on the data structure. The output terminal and input terminal are exactly at the middle of the nodes whereas the other terminals of each node vary in the place according to the number of wires connecting to it.

The Fig 3.1 illustrates the conversion or the formation of directed graphs from a given sample logic circuit. It aptly converts each gate into a node and each wireflow is converted into an arc with direction represented by the arrow mark. Also the output terminals and input terminals are represented as a node which can be optional.

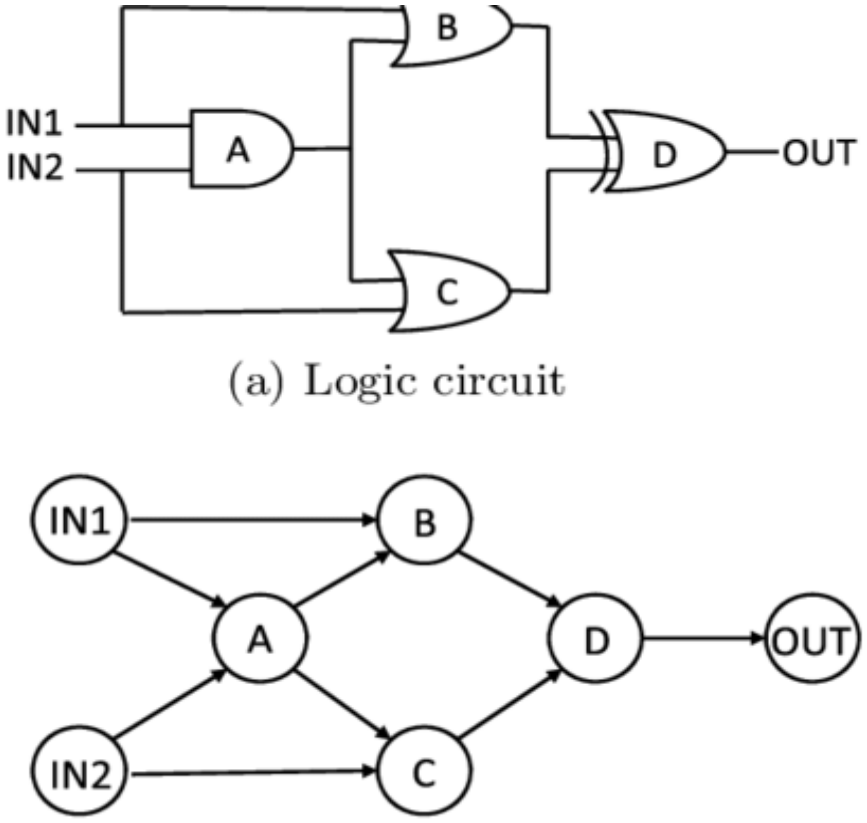

Fig. 3.1 A logic circuit in form of a directed graph

3.2 Grid Creation

A grid or a matrix is needed to be created. The need for matrix or grid like structure is because it gives uniformity for all the elements. They do not give haphazard design with each module placed at random space and angles. It precisely allocates a definite space for the nodes and confines them to a single cell of matrix. This in turn sometimes reduces the pathlength which is one of our primary objectives.

Using this, the schematic diagram looks well defined inside that space. Each row and column may represent each gate present in the logic diagram. This provides ample space for each node of the directed graph, since the circuit is converted in the form of the graph now.

Even the wires which have to be connected between the modules can be given a streamlined path. The grid boundary can be used to form the wire path, which will place them in that position. Also it includes formation of a net ot bus type circuit where all the wires going through the same grid path can be connected and sent as a single line. Which means the arcs can be connected easily among the directed graphs.

The following function is for representing the definition of a matrix:

$$f : M_k \rightarrow \{R_i , C_j\} \qquad (1.1)$$

R_i and C_j are row and column numbers respectively.

3.3 Assigning columns

Columns are formed from the acyclic graphs done before. The terminals play an important role in assigning the columns for each node. Main task is to find the input terminal, because that's the one which is going to start the signal flow. So the basic idea is to start with the nodes having initial input terminals and prioritize them to be given a place in the first column. Sometimes there may be certain gates or modules which do not even have an input terminal. They maye the ones which give constant value for the other gates. So these kinds of gates can be placed in the second column. The next type of nodes which have a signal flow from previous modules have certain conditions. They can have input from multiple nodes and hence recent and closest nodes should be found for this. The column allocation for this particular node is basically next to the column of the immediate previous connected node.

The column starts from the left side and each node gets assigned.
The criteria is as given below :

$$\begin{aligned} &C_m = 0 \text{ , if } \exists \text{ input } T_j \text{ for } M_m \\ &C_m = 1 \text{ , if } \nexists \text{ input } T_j \text{ for } M_m \\ &C_m = \max (C_{m-1}) + 1 \end{aligned} \qquad (1.2)$$

Here C_m represents the column number for that particular node. T_j represents the terminal which is a condition to check if any direct input terminals are there for the module. M_m represents the module name for which the column is to be assigned or allocated.

Since the nodes with multiple inputs have a logic of maximum placed previous input nodes added to one, the nodes with huge connections are placed at the right side part of the schematic diagram. This also helps in getting rid of right to left movements. Any place where a node requires output of another gate is always placed after the node from which output has to be gained. This also helps in gaining good readability.

As a result, nodes with a lot of connections are placed in the rightmost part, reducing right to left movement of circuits.

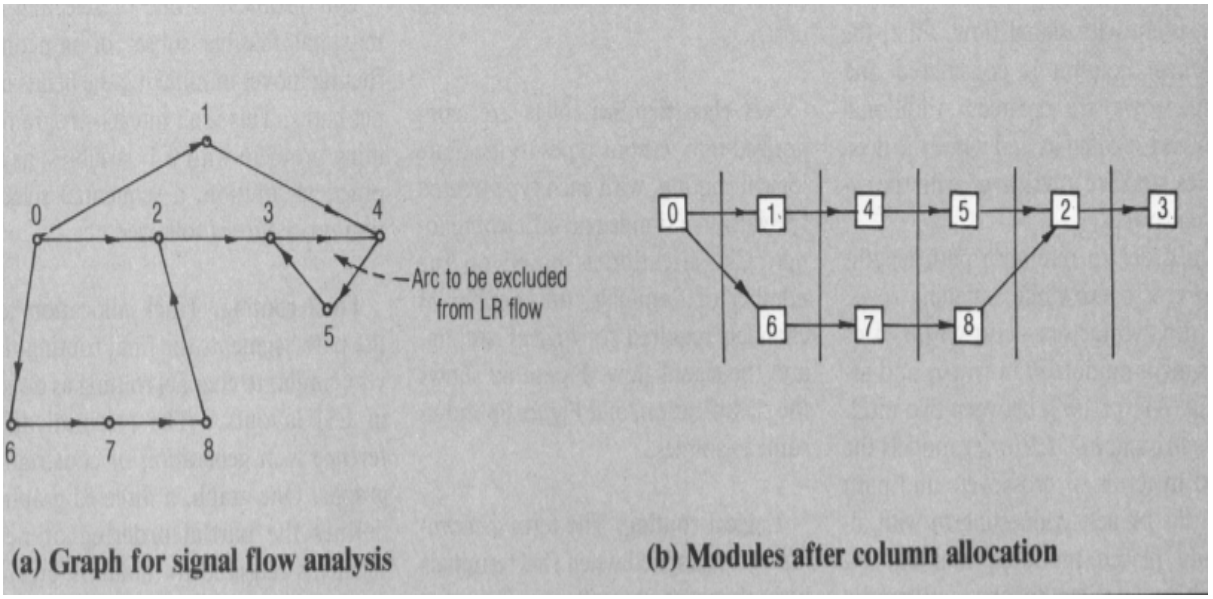

Fig. 3.2 Converting the graph into columns

3.4 Some Definitions

Paths - The arcs present in the graph are referred to as paths.

Crossovers - The intersections of different arcs one over another is referred to as crossovers. It reduces readability and is dependent on the path of signal flow. It is one of the main factors reducing compactness.

Path Length - It is the distance between two nodes in the grid.
Path length can be calculated between two modules m1(c1,r1) and m2(c2,r2).

$$\text{Length}(M1, M2) = |c2 - c1| + |r2 - r1| \quad (1.3)$$

3.5 Assigning rows

Row is assigned for each node from the leftmost column. Extra rows are created when the number of nodes in a column increases than the leftmost column. This helps in conserving space of the grid or matrix. Unlike assigning columns, row allocation is somewhat different. It completely depends on the path length that will be reduced and to calculate it two functions are required.

To reduce path lengths, the cost of placing a row for a module ‘m’ defined as CP(M,R) and affinity between two modules ‘m1’ and ‘m2’ which are defined as AF(M1,M2) are used as factors. In CP(M,R), Cost is calculated in terms of crossovers and path length of nets connecting m with the already placed module. In AF(M1,M2), Affinity models the need to place the modules ‘M1’ and ‘M2’ in consecutive rows minimizing crossover between ‘M1’ and ‘M2’ to modules yet to be placed.

3.6 Wires Classification

Based on the number of terminals between two nodes or gates in a logic circuit various types of wires can be classified. There are basically two categories : (a) Two pin nets and (b) Multi pin nets. There are four types in two pin nets and three types in multi pin nets. The following Fig 3.3 will give a best illustration about the types of wires.

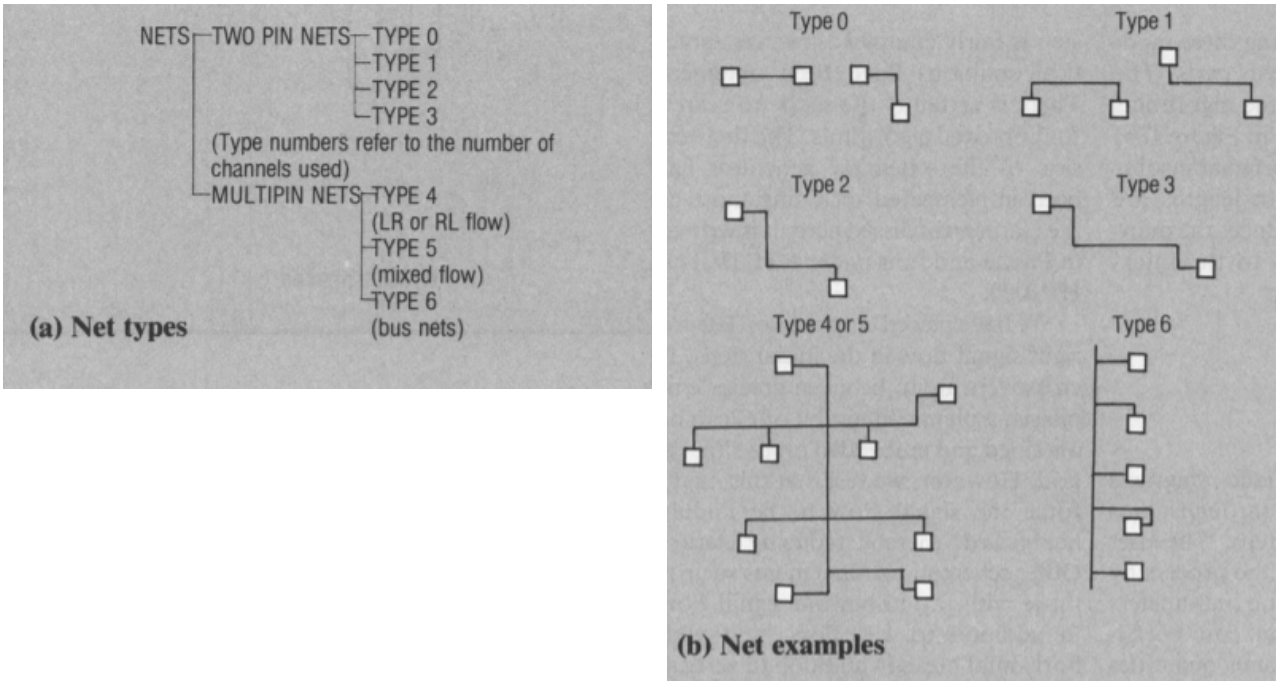

Fig. 3.3(a) Type of Wires (b) Examples

3.7 Final Placement

Every slot contains only one module according to the assignment we performed for the nodes. It will be too difficult to manage if we allot exactly one node to each slot. Hence adjusting smaller modules together in a single slot is expected. This is because some modules might be scattered and this might lead to unnecessary bends and increase path length. Wire types can also be changed to give the optimal placement.

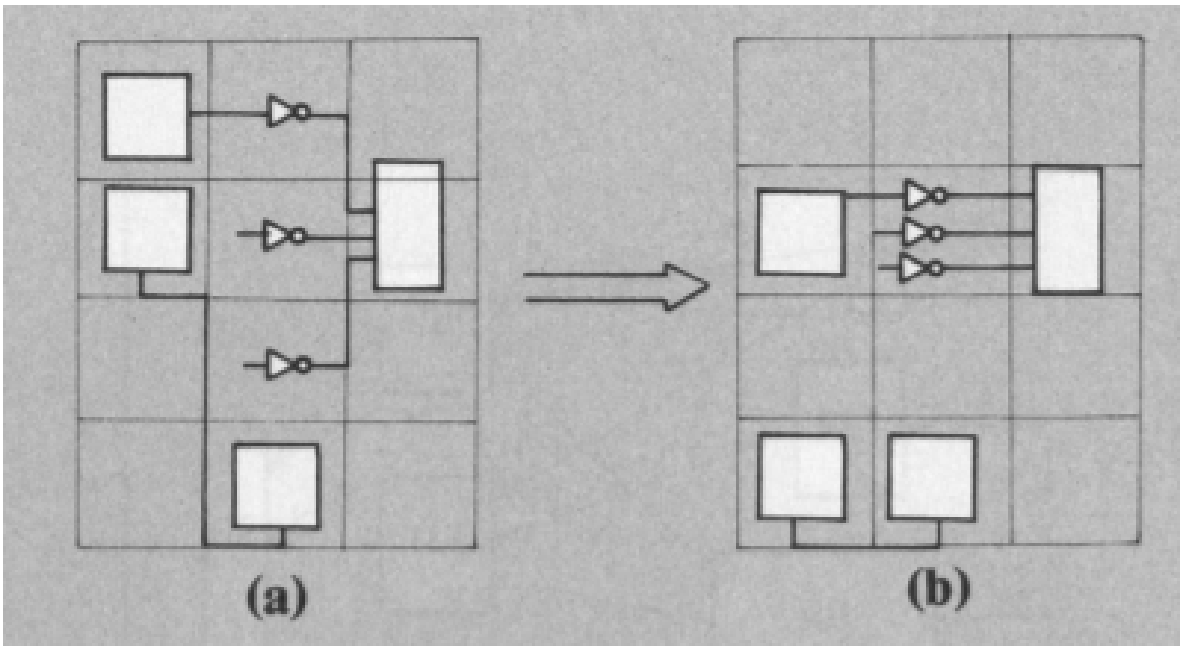

Fig. 3.4 Transformation of schematic design in Final Placement

Lot of free space has been reduced which can be observed from Fig 3.4. Similar gates can be grouped and kept in one cell of the matrix. The main placement technique is to check once again for the free spaces and path length. Sometimes connections may seem to have significantly less path length but should also be checked for the possibility of lowest path length to which it can be optimized.

3.8 Final Routing

One of the main objectives is to reduce overlapping and crossover in the schematic diagram. So correct track allocation must be done for the wires to avoid unnecessary bends present. For this specifically a directed graph and an undirected graph are required. The directed one gives an idea of segments of wires to reduce crossovers. The undirected graph specifies the parts which should be kept in unique tracks to avoid overlapping.

There might be a lot of tracks and different channels through which the wires can reach the other node. Allocation should be made such that the sub graphs created should not go through other tracks even though path is less. Main objective is to bring standard straight lines without deviations. So any chance of changing from horizontal to vertical should also be avoided.

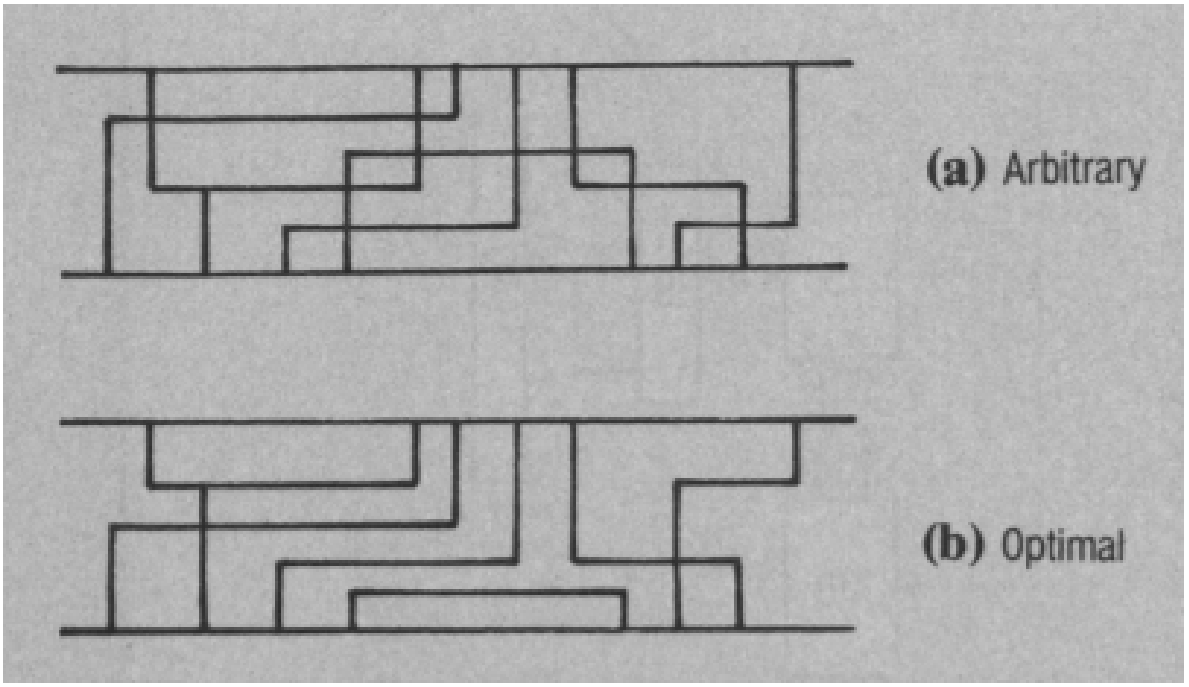

Fig. 3.5 Transformation after final routing

From Fig 3.5 too many absurd connections are bifurcated and normal connections are routed. Unnecessary passing through other tracks have been avoided.

3.9 Locating the terminals

Terminals should be kept in such a way that most probably a direct horizontal connection is available. Too much deviation is not appreciated and leads to confusion. Universal rule of incoming terminals on the left side of nodes and outgoing terminals on the right side of nodes should be followed.

Sometimes due to disorientation, the terminals in random order cause crossovers. For example the terminal which is first on the node may have the connecting wire below it and the one which is last on the node may have connecting wire above the node. Like this crossovers are caused in the circuit.

The final schematic diagram should avoid all these types of disorientations and the elements causing unwanted crossovers. However there could be some exceptions such as in certain modules or gates, the input has to be got from the bottom side of the node. In this case there is no other way than to place the terminals in that position. The below figure would help in providing insight.

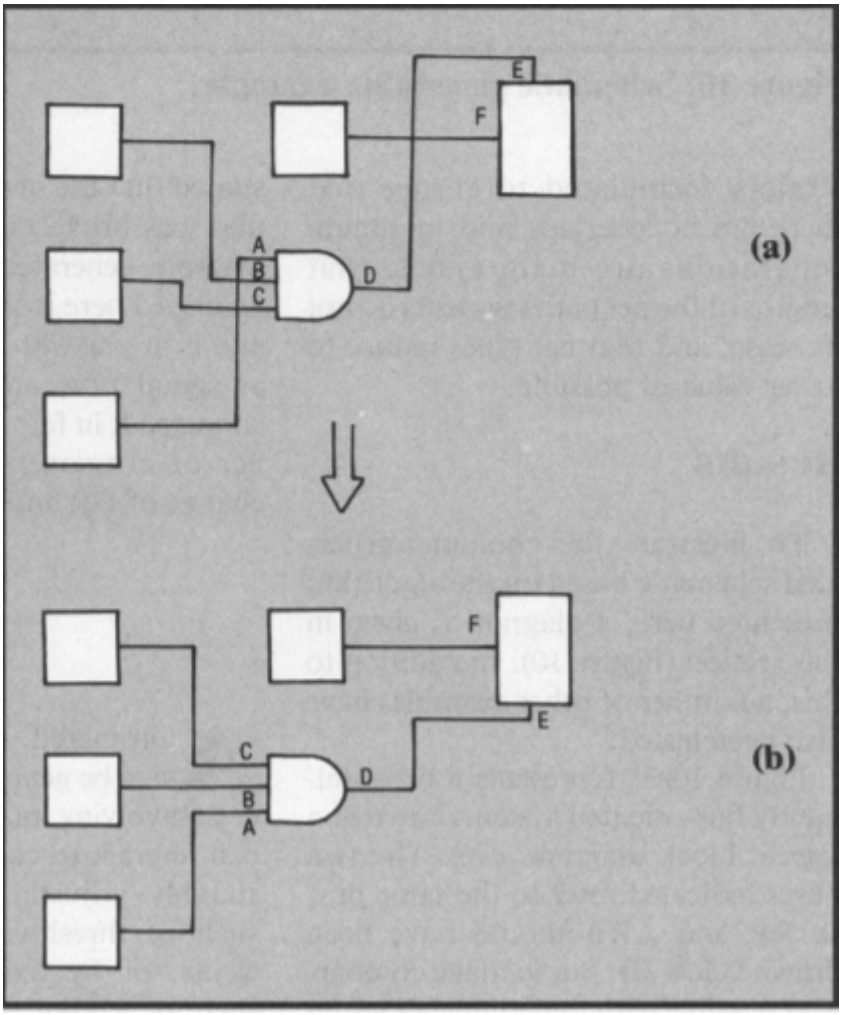

Fig. 3.6 Optimal Terminal Positioning

Fig 3.6 shows how the transformation needs to be done for gaining the objectives. As mentioned all the cases have been covered in this only. Like how the placement of terminals A, B and C need to be done to minimize the crossover based on the node connectivity. The terminals need to be interchanged and have thus given a good visual. Similarly the terminal E had to be placed on the bottom part only to reduce crossover with wire of terminal F and gate orientation.

3.10 Matrix Realization

To find the number of crossovers that's gonna take place in a given circuit matrix realization is needed. We convert each consecutive column into a matrix and find the number of crossovers. Matrices are formed between two consecutive columns. It basically defines the connection of two vertices in the 2 different columns.

The total number of matrices for a circuit = Total number of columns - 1 (1.4)

These dimensions of matrices are based on the number of elements present in each column. The number of rows in the matrix represent the number of elements in the first chosen column of the schematic diagram. The number of columns in the matrix represents the number of elements in the second chosen column of the schematic diagram.

Let the matrix be represented as S. Let ‘i’ and ‘j’ represent the element number in the first column and the second column respectively. Each element of matrix is filled based on the following:

S[i][j] = 0 , if there does not exist an edge between ‘i’ and ‘j’
S[i][j] = 1 , if there exist an edge between ‘i’ and ‘j’ (1.5)

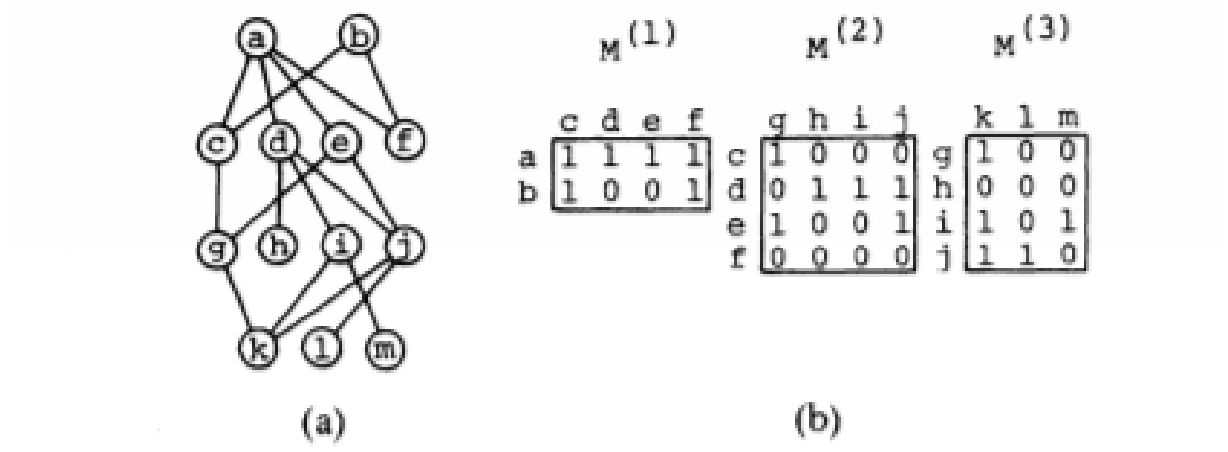

Fig. 3.7 Matrix realization of a sample graph

Fig 3.7 tells how matrix realization is done for a multi level hierarchy. The number of nodes plays a role in this. Each node from one hierarchical level forms a columnar data and each node from the connecting hierarchical level forms the row data. This matrix should be done for each and every consecutive hierarchy, because crossover occurs between the succeeding nodes.

3.11 Algorithm for Number of Crossovers

After converting the graph to matrices, we can calculate the number of crossings.
For a matrix of two hierarchies, and let's say it has ’m’ rows and ’n’ columns represented as Matrix[m][n].

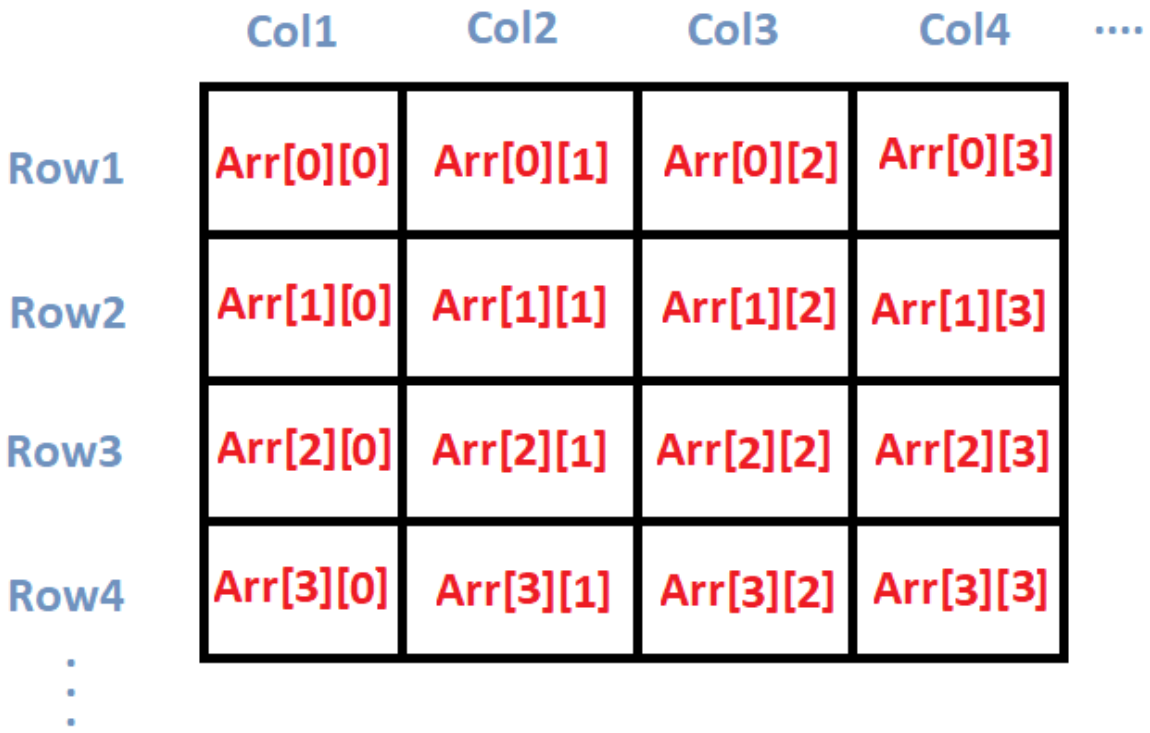

Fig. 3.8 Matrix example

The algorithm is as follows :

1) declare numCross = 0
2) for ’i’ from ’1’ to ’m-1’ loop 1
3) for ’j’ from ’i+1’ to ’m’ loop 2
4) for ’k’ from ’1’ to ’n-1’ loop 3
5) for ’l’ from ’k+1’ to ’n’ loop 4
6) numCross += Matrix[j][k]*Matrix[i][l]
7) end loop 4
8) end loop 3
9) end loop 2
10) end loop 1
11) return numCross

The total number of hierarchies is noted and a global variable for the number of crossovers is declared with zero. Then as said in the previous section, for each consecutive hierarchy the crossover calculations are going to be done. So a local variable for calculating the localized crossover called “numCross” is defined and declared with initialization of zero. From this the calculation for the crossover between 2 matrices starts.

Since for access to a single element of matrix we need two different variables as column and row number needed to be accessed, the double amount of variables is needed to access two hierarchies. This is the reason four variables and loops are present in the algorithm.

The main idea is to calculate the crossings from the next columns and other rows of the matrix. When observed keenly, there can never be a crossover with the currently iterating column or row. Also it is pointless to check with previous columns as it wont give significance for valid crossover. They will never crossover and they are avoidable calculations.

When crossover occurs we use mathematical functions of multiplication with addition to increase the count. Using binary elements to fill in the matrix brings significance now. That is, the product is done with elements of the realized matrix.

When no edges are present the multiplication value eventually turns out to be zero. So adding that to the result wont change anything to the final result. At the same time when two elements are forming connections then their multiplication leads to one, which when added gives a significance to the final number of crossovers. Each multiplication gives out that one crossover has been detected.

If there are supposed to be say 'N' no. of hierarchies, the above algorithm is applied for all the 'N-1' matrices that are formed and summation is applied.

Lets see a solved example for this :

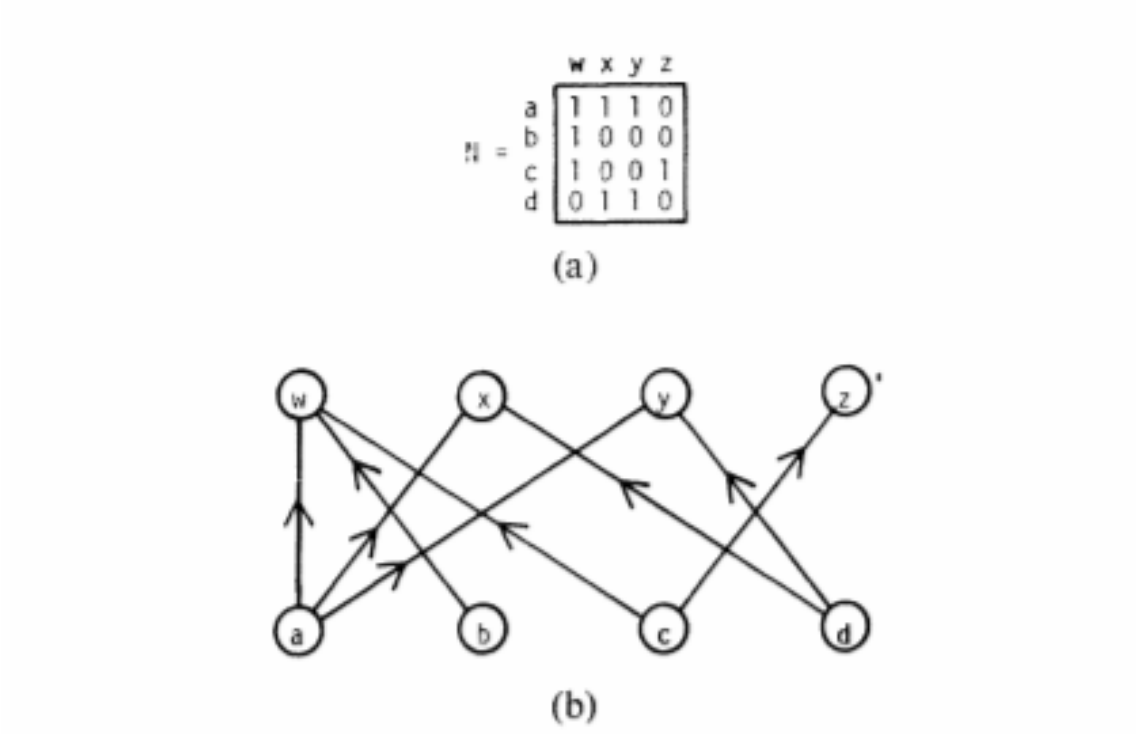

Fig. 3.9 An example to illustrate the algorithm

When calculation is done using the above algorithm for the above example, we arrive at the result of numCross = '7', which can be verified manually by counting from the graph also.

The crossovers are found in the junctions and intersection points of the consecutive hierarchy connections. Element a and w do not create crossover however the connection of a with other elements x and y increases crossover to five. This is because the connections of elements b and c with element w brings in points of intersection with connections of elements of a. Two connections with intersection of another two connections gives four crossovers. The connection of d with x adds one more crossover with element a connection. Element c is connected with z and d is crossing over to connect with elements x and y which gives two more crossovers. Hence total number of crossovers to 7.

3.12 Reduction of Crossovers

From previous sections it is evident that the number of crossovers is dependent on the columns and connections with other elements across different columns. The only way to reduce this type of crossover is by changing the orientations of the columns. As we know, it is pointless to have chronological ordering in digital circuits. Added advantage of circuit elements can work normally when placed anywhere and at whatever position.

When Fig 3.9 is looked into, some connections literally avoid intersection with other connections. There can be two reasons for this. One of them is, the element has only one connection with the other element in successive hierarchies. The other one is the placement of the element such that crossover is avoiding other interconnections.

From a logical perspective it is not apt to make every element to have singular connections. Every engineer knows that circuits work with various inputs and connections. So restricting the number of connections for a module or a gate is literally a bad way. Any algorithm or theory based on this current conclusion will not work for every use case. Further instead of solving the problem, it leads to more problems and more crossovers in the latter stage of the design.

So the only way to expand this ideology is to change the orientation of the modules and gates. For this purpose some mathematical way is needed. It is difficult to decide by just rearranging to a particular configuration and concluding that will be the best possible combination. So a greedy approach won't work out for solving this problem. This is because total possible solutions will be numerous and should not be settled with the first possible solution.

For getting the best possible solution, an exhaustive search method must be used. Here a lot of combinations are formable and sometimes a number of solutions may occur. To get out the best and optimal solution all possible combinations must be tried out and stored. Hence a suitable backtracking way of permuting the places of all the hierarchical elements needed to be executed. Crossover for each combination should be stored and after calculating all possible combinations crossover value, the minimum from that should be taken. We need to make use of a map data structure to map the combination and corresponding crossover value, so that after all calculations we can easily get the required solution from this data structure only.

The above given example can be used to formulate this permuting algorithm and results can be seen. It shows how the crossover has significantly reduced when various combinations are tried and an exhaustive search is carried out.

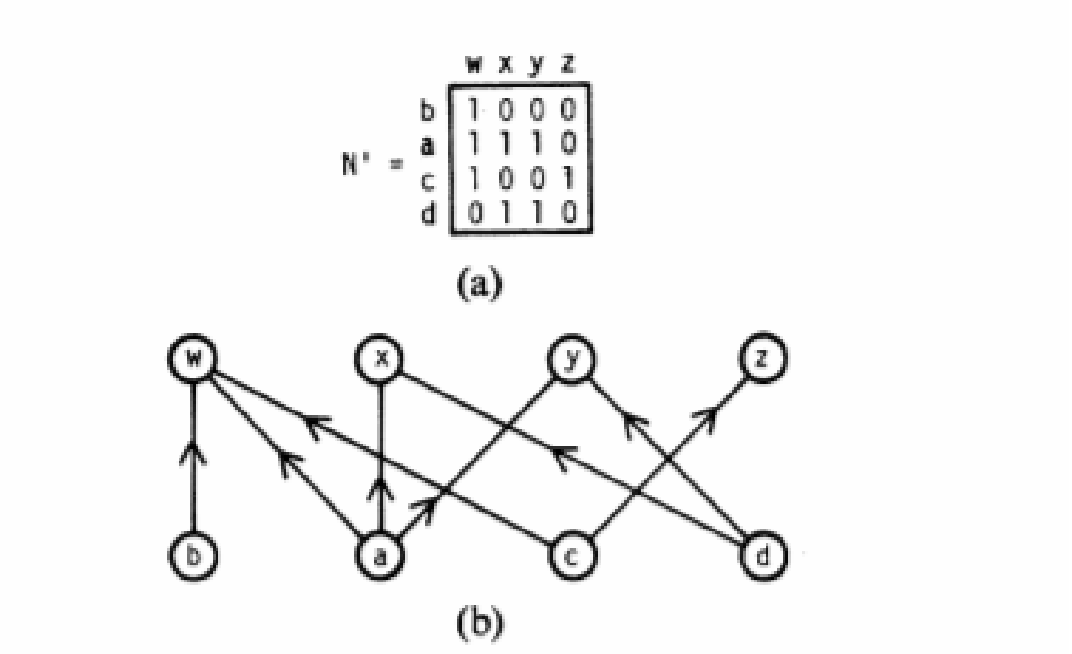

Fig. 3.10 Reduction of crossovers

This particular permutation gives only 5 crossovers. This is one of the possible combinations from the given matrix example. The search is performed exhaustively for all combinations and minimum of them is chosen. Though we can count it manually and verify, it's difficult when the number of elements in each hierarchy are too large. That's why appropriate data structures are used for it and necessary algorithms are implemented.

If it is viewed mathematically the permutation size is huge and it is a disadvantage in the view of time and space complexity. Suppose there are 'r' rows and 'c' columns, then the total number of combinations becomes r!*c!. Factorial itself is a big value, then product of two factorials is really a huge huge number. Hence this type of algorithm suits only for small examples and is useful in smaller applications.

Also there is another approach to bring out no crossover model.

3.13 No Crossover Model

There is a possibility that can bring the number of crossovers to zero. Instead of worrying about various combinations, just create a new hierarchy at the place of crossover.

From the previous section it is observed that reduction of crossovers increases the time and space complexity of the matrix. To avoid this, construction of new hierarchies can easily lead to solutions which are free from crossovers.

The following example illustrates it:

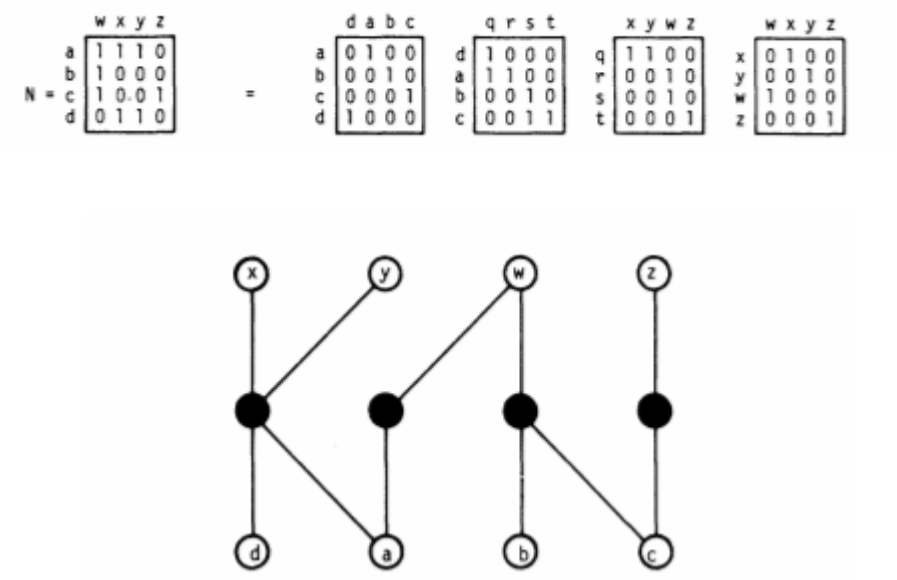

Fig. 3.11 No crossover map

So by creating a new hierarchy, the crossovers can be nullified. It is evident from Fig 3.11. Using the digital circuit logic we can just add dummy modules to connect and bring in a beautiful visualization for the problem statement. Though this is a tedious method, one can use it for large applications as it will be easy to grasp the logic with a beautiful schematic diagram. Also it completely nullifies the cycles and tangles got from the wires. This can also be used as an alternative for gaining our primary objectives.

CHAPTER 4

RESULTS AND DISCUSSION

The expected output after the above algorithm is followed is given below.

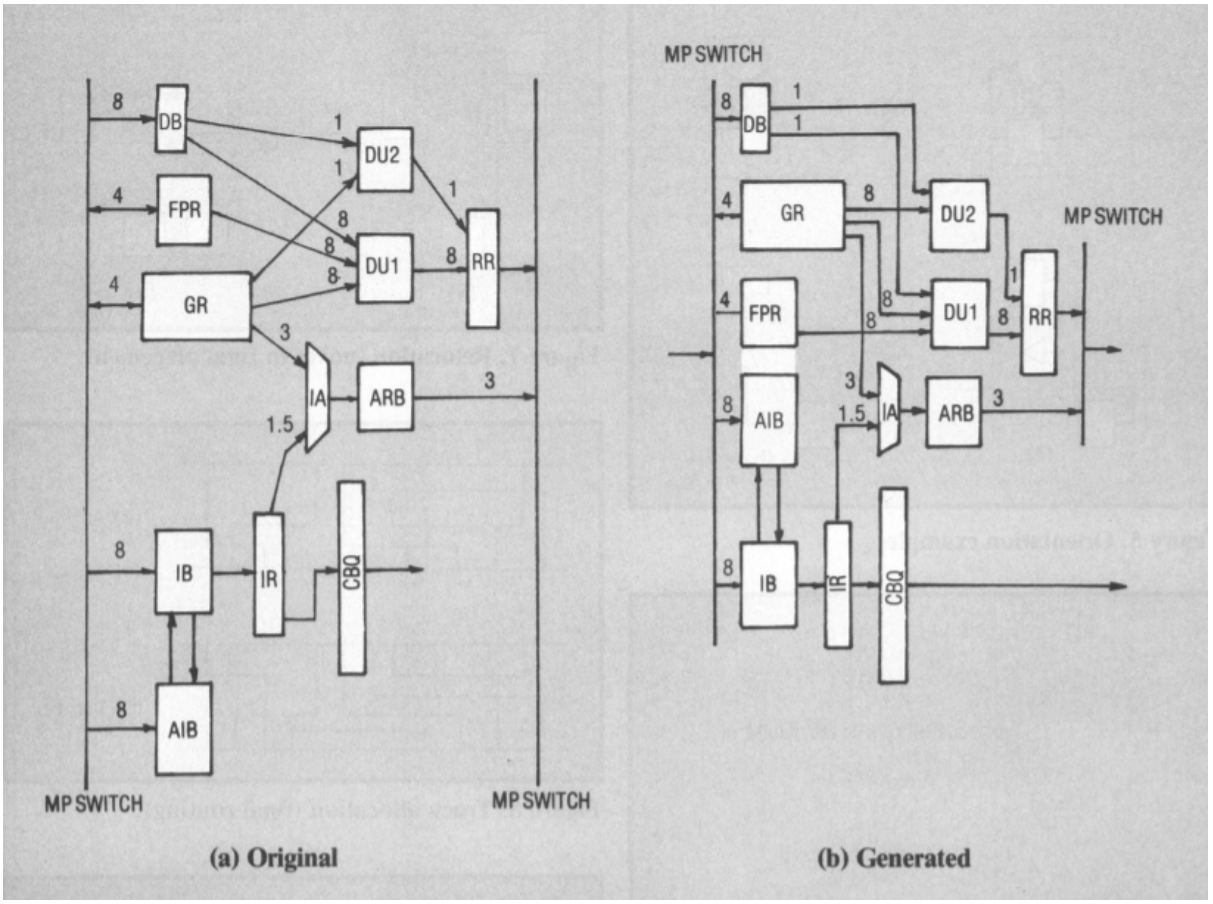

Fig. 4.1 Expected Output

From Fig. 4.1, it is observed that almost all the objectives are met for this problem statement. Comparing the original circuit and the generated one, the new one has less crossovers and bends. The wires are streamlined to only horizontal and vertical instead of criss-cross nature. This gives a good visualization.

CHAPTER 5

CONCLUSION AND FUTURE WORK

A suitable digital circuit design method and algorithm is implemented. Based on these the visualization will be good and can be used in various applications. As everyone knows digital circuits are used in most of the appliances around the world and most homes are incomplete without these appliances. So designing circuits for those appliances before implementing them can be checked based on this. This avoids a lot of problems compared to when practically done first itself. This is very useful for testing and debugging purposes.

In future this can also be used for teaching purposes. Like if any professor wants to explain digital circuits as a subject to students, they can use this to shoe a visually powerful design. Also this can be improved by using reinforcement learning models to implement dynamic circuits. Like if an additional element is needed to be added, the chage and alteration can be done dynamically and changes are also viewed instantaneously.

CHAPTER 6

REFERENCES

A. Arya, V. V. Swaminathan, A. Misra and A. Kumar, "Automatic Generation of Digital System Schematic Diagrams," 22nd ACM/IEEE Design Automation Conference, 1985, pp. 388-395, doi: 10.1109/DAC.1985.1585970.

K. Sugiyama, S. Tagawa and M. Toda, "Methods for Visual Understanding of Hierarchical System Structures," in IEEE Transactions on Systems, Man, and Cybernetics, vol. 11, no. 2, pp. 109-125, Feb. 1981, doi: 10.1109/TSMC.1981.4308636.

J. N. Warfield, "Crossing Theory and Hierarchy Mapping," in IEEE Transactions on Systems, Man, and Cybernetics, vol. 7, no. 7, pp. 505-523, July 1977, doi: 10.1109/TSMC.1977.4309760.

D. Castells-Rufas and J. Carrabina, "Jumble: A Hardware-in-the-Loop Simulation System for JHDL," 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007), 2007, pp. 345-348, doi: 10.1109/FCCM.2007.54.

CHAPTER 7

APPENDIX

Sample Code:

```
class AdjNode:

    def __init__(self, data):
        self.vertex = data
        self.next = None


# Generate graphs based on the ports and nets


rows,cols = (gridsize,gridsize)
matrix = [[0]*cols]*rows


class genGraph:


    def __init__(self,sources):

        self.V = len(sources)
        self.graph = [None]*self.V


    def add_edge(self,sources):

        for i in range(len(sources)-1):
            for j in range(i+1,len(sources)):
                if sources[j].parent == sources[i]:
                    node = AdjNode(sources[j])
                    node.next = self.graph[sources[i]]
                    self.graph[sources[i]] = next


    def print_graph(self):

        for i in range(self.V):
            print("Vertex adjacency list {}\n head".format(i), end="")
            subs = self.graph[i]
            while temp:
                print(" -> {}".format(subs.vertex), end="")
                subs = subs.next
            print(" \n")
```

```
    def col_assignment(self):

        for i in range(self.V):
            k=0
            if(self.graph[i].isPrimitive()):
                matrix[0][k] = self.graph[i]
                k+=1
            else:
                matrix[self.graph[i].col][self.graph[i].row+1] = self.graph[i]



    def row_assignment(self):

        cp_score = []
        af_score = []
        for i in range(self.V):
            for j in range(matrix.size()):
                cp_score[i] = crossover_calc(matrix[i],matrix[i+1]) +
path_length(matrix[i][j],matrix[i+1][j])
                af_score[j] = affinity(matrix[i][j],matrix[i][j+1])
        for i in range(self.V):
            for j in range(matrix.size()):
                count = matrix[j].size()
                for k in range(count):
                    if(cp_score[j]>cp_score[i] and af_score[j]>af_score[i]):
                        matrix[k][0] = self.graph[i]



    def path_length(self,matrix):

        for i in range(self.V):
            for j in range(matrix.size()):
                len = abs(matrix[i][j].col - matrix[i+1][j].col) + abs(matrix[i][j].row -
matrix[i+1][j].row)

        return len


    def affinity(self,matrix):

        for i in range(self.V):
            for j in range(matrix.size()):
                if(path_length(matrix[i][j],matrix[i+1][j]) < gridsize):
                    return 1

        return 0
```

```
def crossover_calc(self,matrix):

    numCross = 0

    for i in range(cols-1):
        for j in range(cols):
            for k in range(rows-1):
                for l in range(rows):
                    numCross += Matrix[j][k]*Matrix[i][l]

    return numCross
```