**FINDING THE SHORTEST PATH BETWEEN**

**TWO LOCATIONS USING DIJKSTRA**

**ALGORITHM**

By

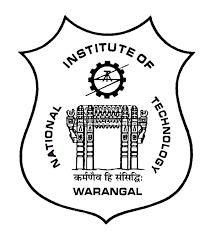
Ramya Laxmi 21ECB0B55

Rudra Kalyan 21ECBB056

V. Sai Teja 21ECB0B57

Guided by:

|  |  |
| --- | --- |
| **Dr. P. Prithvi** | **Dr V. Narendar** |
| **Assistant Professor** | **Assistant Professor** |



**DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING**

# NATIONAL INSTITUTE OF TECHNOLOGY, WARANGAL

**2022-2023**

**CERTIFICATE**

This is to certify that the dissertation work entitled “IMPLEMENTATION

OF “FINDING SHORTEST PATH BETWEEN TWO LOCATIONS

USING DIJKSTRA ALGORITHM” is a bonafied record of work carried outwork by Ramya Laxmi (21ECB0B55), Rudra Kalyan (21ECB0B56) and V. Sai Teja (21ECB0B57) submitted to faculty of “Electronics and

Communication Engineering Department” , in partial fulfilment of the requirements for the award of the degree of Bachelor of Technology in “Electronics and Communication Engineering” at National Institute of Technology, Warangal during academic year (2022-2023).

**Dr. P. Prithvi MAM Dr. V. Narendar SIR**

Assistant Professor

Department of Electronics and

Communication Engineering

National Institute of

Technology Warangal Assistant Professor

Department of Electronics and

Communication Engineering

National Institute of

Technology Waranal

# ACKNOWLEDGEMENT

I would like to express my deepest gratitude to my faculty in-charge **Dr. P. Prithvi, Assistant Professor**, Department of Electronics and Communication

Engineering and **Dr. V. Narendar**, **Assistant Professor**, Department of Electronics and Communication

Engineering, National Institute of Technology, Warangal for their constant supervision, guidance, suggestion and encouragement during this semester.

# ABSTRACT:-

Dijkstra algorithm is a path planning algorithm which is used to find the shortest path between two given nodes.

The Dijkstra algorithm has multiple applications like google maps. In google maps it is used to find the shortest path between two locations.

It is also used in various networking applications like networking applications like social networking applications and telephone network where it is used to find the shortest path for signal to travel.

It also has several applications in the robotics where We use this algorithm to deliver the packages to specified locations.

It is used in IP routing applications where we find the shortest path between transmitter and receiver.

TABLE OF CONTENT:

1)INTRODUCTION 1

2)MOTIVATION 1

3)HOW DIJKSTRA ALGORITHM WORKS 2 • INITIALIZATON • LOOP • TERMINATION • PATH RECOSTRUCTION

4)ASM FOR VERILOG IMPLEMENTATION 4

5)EXPLAINATION OF CODE 5 5.1. STATE ASSIGNMENT

6)MAP 8

7)VERILOG CODE 9

8)TEST BENCH 18

9)FINAL WAVEFORM 20

10)FPGA RESULTS 21

11)ELOBORATED DESIGN 22

12)CONCLUSION 23

## 13) REFERENCE 24

**1.INTRODUCTION:**

1. The Dijkstra algorithm finds many applications in hardware designs hence implementing the Dijkstra algorithm using Verilog can make it possible to interface this algorithm in specific cases when there is a constraint of shortest path.

1. By implementing Dijkstra's algorithm in Verilog, it could be possible to optimize the algorithm for a specific hardware platform.

This could lead to improved performance and reduced power consumption compared to running the algorithm on a general-purpose processor.

# 2.MOTIVATION

The algorithm has found practical uses both in offline and real-time applications across several domains, including mobile robots' path planning, image processing segmentation, and telecom network routing. Our work was specifically motivated by the need for "maze routing," which involves routing conductive tracks on a printed circuit board between the pins of components without overlapping or crossing other tracks. This algorithm can be particularly useful in CAD

design, where precision and efficiency are crucial for successful circuit board manufacture In Dijkstra algorithm, we consider starting nodes as source nodes. We assume every node is at infinite distance from the source Node. Then will update the weights of each of node with minimum distance. We continue this process this each and every node is visited. The final weights of the nodes are the shortest distances from the source node. There are many algorithms to find the shortest path but this the fastest algorithm with minimum time complexity

# 3. HOW DIJKSTRA ALGORITHM WORKS

The Dijkstra algorithm is a graph traversal algorithm that is used to find the shortest path between two nodes in a weighted graph.

The algorithm works by iteratively visiting nodes in the graph and updating the shortest path to each node based on the minimum distance between the node and its neighbours.

Here is a detailed explanation of how the Dijkstra algorithm works:

**3.1.Initialization:**

Set the distance of the starting node to 0, and the distances of all other nodes to infinity.

Add the starting node to a map1, where the priority is based on the distance from the starting node.

**3.2.Loop:**

For each neighbour of the node, calculate the distance to the neighbour by adding the distance from the current node and the weight of the edge between the two nodes.

If the new distance to the neighbour is smaller than its current distance, update the neighbour’s distance. We consider the node with the shortest path length after updating with respect to the present node and select this node as the next node.

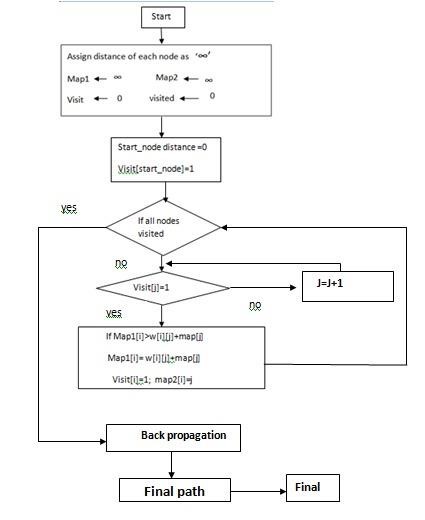
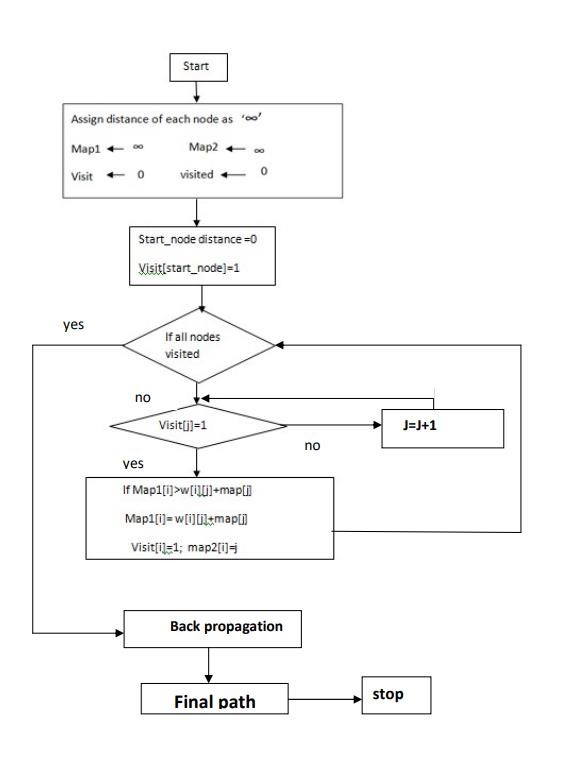
**3.3.Termination:**

The algorithm terminates when the destination node is reached.

**3.4.Path Reconstruction:**

The shortest path between the starting node and the destination node can be reconstructed by following the previous nodes from the destination node to the starting node.

# 4.ASM FOR DIJKSTRA ALGORITHM:-



**5.VERILOG CODE EXPLANATION:-**

The Verilog code takes in a clock signal, start signal, start node, end node, and outputs a final path and a

done signal.

The state register is used to keep track of the current state of the algorithm. The map1 and map2 arrays are to used store the current distance from the start node to each node and the previous node in the shortest path to each node respectively.

The visit and visited registers are used to keep track of which nodes have been visited and which nodes have been fully explored.

The w array is used to store the weight of the edges between nodes. The “cal” function is used to calculate the distance to neighbouring nodes and update the map1 and map2 arrays.

The final\_temp register is used to temporarily store the final path. The count register is used to keep track of which index of the final\_temp register is being written to.

This code implements a finite state machine for path planning algorithm. The state machine has four states, which are explained below:

# 5.1.STATE ASSIGNMENT:-

**State 0:**

This is the initial state of the state machine. It waits for the start signal to be asserted. When the start signal is detected, it transitions to state 1.

**State 1:**

In this state, the map is initialized, and the starting node is marked as visited. Then, the state machine transitions to state 2.

**State 2:**

In this state, the algorithm performs path planning by iteratively selecting the unvisited node with the lowest tentative cost, and updating the costs of its neighbouring nodes. Once all nodes have been visited, the state machine transitions to state 3.

**State 3:**

This state is responsible for backtracking from the goal node to the start node, and generating the final path. It starts by setting the current node to the goal node, and then iteratively adding the previous node in the path until the start node is reached.

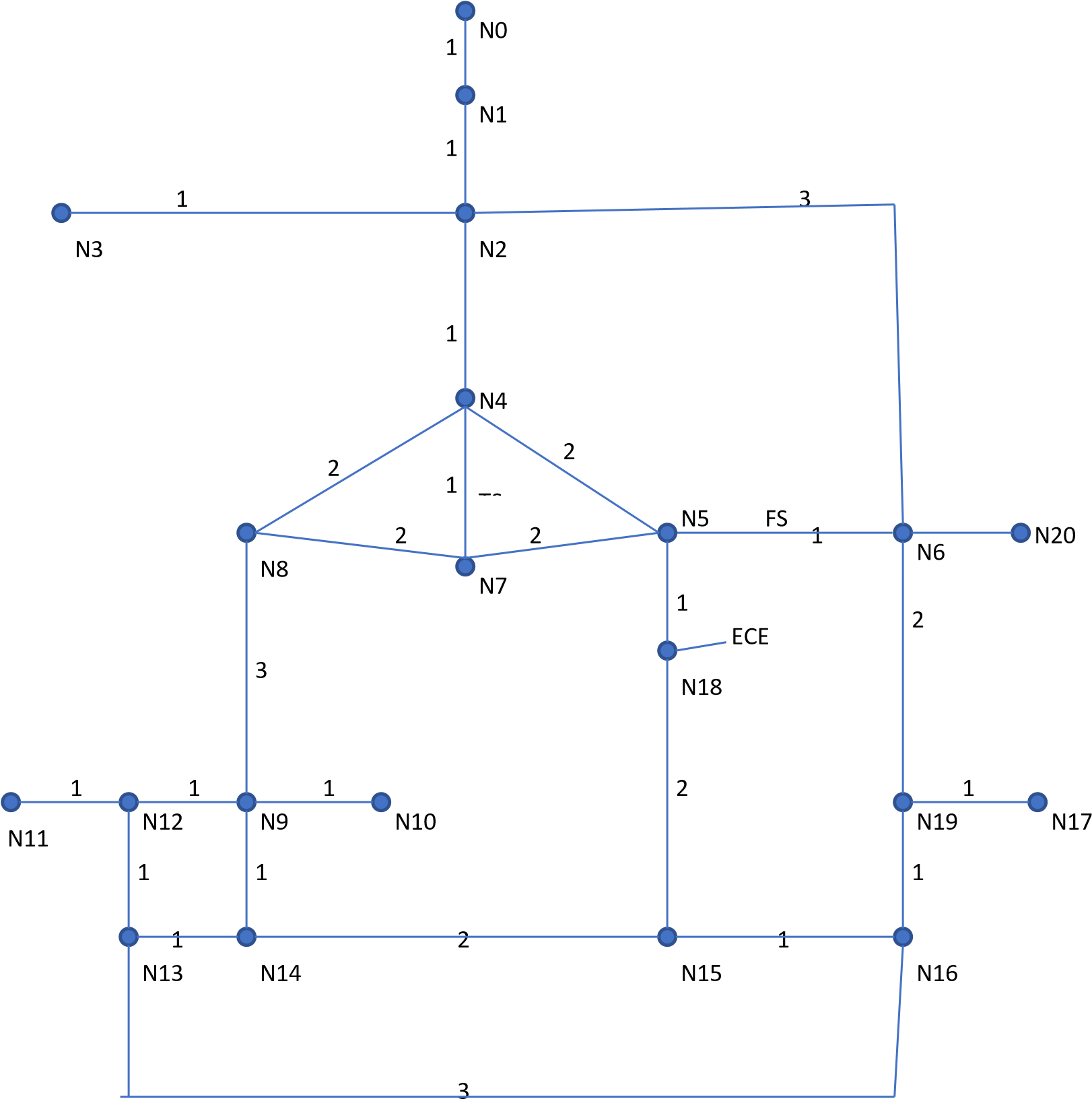
The implementation uses an adjacency matrix to represent the graph, and the costs of the edges are stored in a twodimensional array.

The algorithm keeps track of the tentative cost of each node, which is the cost of the path from the start node to the current node, and the previous node in the path, which is used for backtracking.

Once the goal node is reached, the algorithm backtracks from the goal node to the start node to generate the final path.

The final path is stored in the final\_path output signal, which is a concatenation of the node indices in the pa

# 6.NITW MAP



# 7.CODE:-

module path\_planner

#(parameter node\_count = 22, parameter max\_edges = 4)

(

input clk, input start, input [4:0] s\_node, input [4:0] e\_node, input [2:0]mode, output reg done, output reg [9:0] finall

) ;

reg [10\*5-1:0] final\_path; reg [4:0]in=5'b10110; reg [1:0]state=0; reg [4:0]map1[21:0]; reg [4:0]map2[22:0]; reg [20:0]visit; reg [20:0]visited; integer i,k,count; reg [4:0]w[20:0][20:0]; reg [10\*5-1:0]final\_temp;

reg [4:0]prev;

reg [4:0]j;

initial begin

for(i=0;i<21;i=i+1)begin for(k=0;k<21;k=k+1) w[i][k]=5'b10110; end

j=0;

final\_path=0; map1[21]=5'b10101; map2[21]=5'b10101; map2[22]=5'b10110; done=1; //Node0= Starting point

//Node1= GATE

//Node2= LOGO

//Node3= NITW Quarters

//Node4= LH

//Node5= FS Entry

//Node6= FS Exit

//Node7= Administration Building

//Node8= OLD Auditorium

//Node9= DISPENCERY

//Node10=CIVIL BLOCK

//Node11=11TH BLOCK

//Node12=BLOCK Junction

//Node13=8TH BLOCK

//Node14=STADIUM

//Node15=PED

//Node16=CRIF

//Node17=1K HOSTEL //Node18=ECE

//Node19=PARKING AREA //Node20=1.8K HOSTLE w[0][1]=1; w[1][0]=1; w[1][2]=1; w[2][1]=1; w[2][3]=1; w[3][2]=1; w[2][6]=3; w[6][2]=3; w[2][4]=1; w[4][2]=1; w[4][8]=2; w[8][4]=2; w[4][7]=1; w[7][4]=1; w[4][5]=2; w[5][4]=2; w[5][18]=1; w[18][5]=1; w[5][7]=2; w[7][5]=2; w[6][19]=2; w[19][6]=2; w[6][20]=1; w[20][6]=1; w[7][8]=2; w[8][7]=2;

w[8][9]=3; w[9][8]=3; w[9][10]=1; w[10][9]=1; w[9][12]=1; w[12][9]=1; w[12][11]=1; w[11][12]=1; w[12][13]=1; w[13][12]=1; w[9][14]=1; w[14][9]=1;

w[13][14]=1; w[14][13]=1; w[13][16]=3; w[16][13]=3; w[14][15]=2; w[15][14]=2; w[15][16]=1; w[16][15]=1; w[15][18]=2; w[18][15]=2; w[16][19]=1; w[19][16]=1; w[17][19]=1; w[19][17]=1;

end always@(negedge clk)begin

case(state)

0: begin if(start)beg in done=0;

state=1;

end end

1: begin for(i=0;i<21;i=i+1)b egin map1[i]=in; visit[i]=0; visited[i]=0;

end map1[s\_node]=0; map2[s\_node]=5'b10110;

visit[s\_node]=1; state=2;

end

2: begin if(visited!=21'b111111111111111111111

)begin if(j<21)begin if(visit[j]) cal(j);

j=j+1'b1;

end else

j=0; end

else begin state=3; count=10; end end

3: begin case(count

)

10: begin final\_temp[4:0]=e\_node; prev=e\_node; count=9;

end

9: begin final\_temp[9:5]=map2[prev]; prev=map2[prev];

count=8;

end

8: begin

final\_temp[14:10]=map2[prev];

prev=map2[prev];

count=7;

end

7: begin

final\_temp[19:15]=map2[ prev]; prev=map2[prev]; count=6;

end

6: begin final\_temp[24:20]=map2[prev];

prev=map2[prev]; count=5;

end

5: begin

final\_temp[29:25]=map2[

prev]; prev=map2[prev];

count=4;

end

4: begin final\_temp[34:30]=map2[prev];

prev=map2[prev];

count=3;

end

3: begin

final\_temp[39:35]=map2[

prev]; prev=map2[prev];

count=2;

end

2: begin final\_temp[44:40]=map2[prev];

prev=map2[prev];

count=1;

end

1: begin final\_temp[49:45]=map2[prev];

prev=map2[prev];

count=0;

end

0: begin

final\_path=final\_te

mp; done=1;

state=0;

count=10;

end

endcase

end

endcase

end

always@(mode) begin case(mode) 2'b000: begin finall=final\_path[9:0]; End 2'b001: begin finall=final\_path[19:10];

end 2'b010: begin finall=final\_path[29:20]; end 2'b011: begin finall=final\_path[39:30]; end 2'b100: begin finall=final\_path[49:40]; end 2'b111: begin finall=0; end endcase

end

task automatic cal(input [4:0]j); begin for(i=0;i<21;i=i+1)begin if(w[j][i]!=in)begin if(!visited[i])begin

if(w[i][j]+map1[j]<map1[i])begin map1[i]=w[i][j]+map1[j]; visit[i]=1;

map2[i]=j;

end end

end end visited[j]=1; visit[j]=0;

end

endtask

endmodule

# 8.TEST BENCH:-

module tb; reg clk = 0; reg start = 0; reg [4:0] s\_node; reg [4:0] e\_node; reg [2:0]mode;

wire done;

wire [9:0] finall;

//wire [10\*5 -1:0]final\_temp;

path\_planner uut (clk, start, s\_node, e\_node,mode,

done, finall); always begin clk <= 0; #1; clk <= 1; #1;

end initial begin

s\_node <= 9; e\_node <= 15; mode<=2'b111; #300 mode<=2'b000; #30 mode<=2'b001; #30

mode<=2'b010;

end initial begin

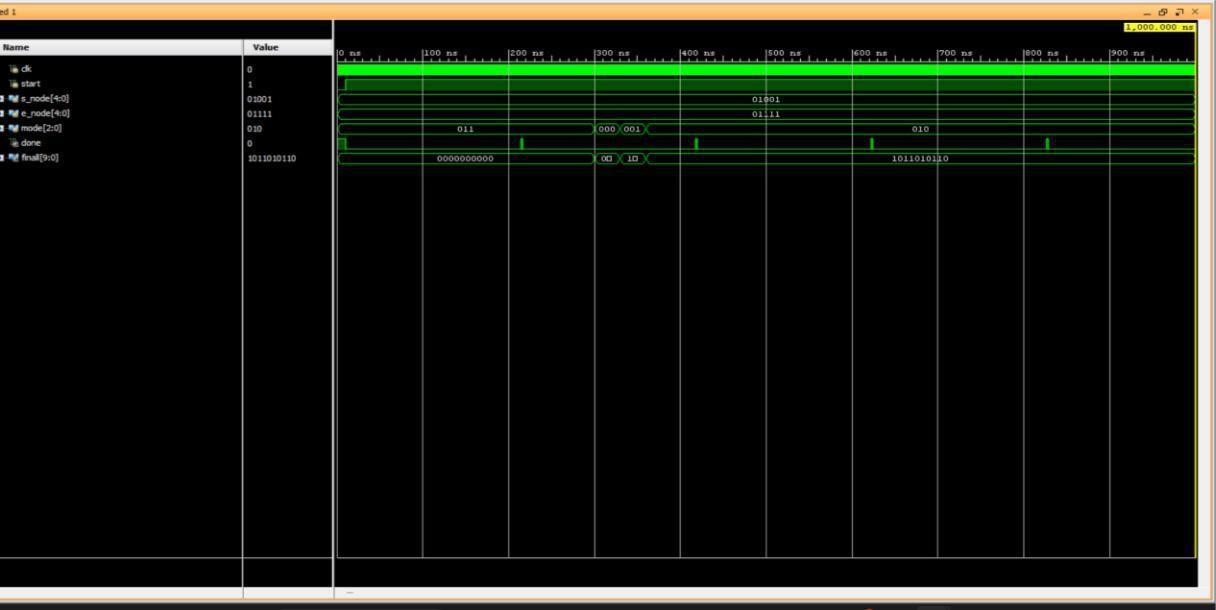
start <= 0; #10

start <= 1;#1000 start <= 0;

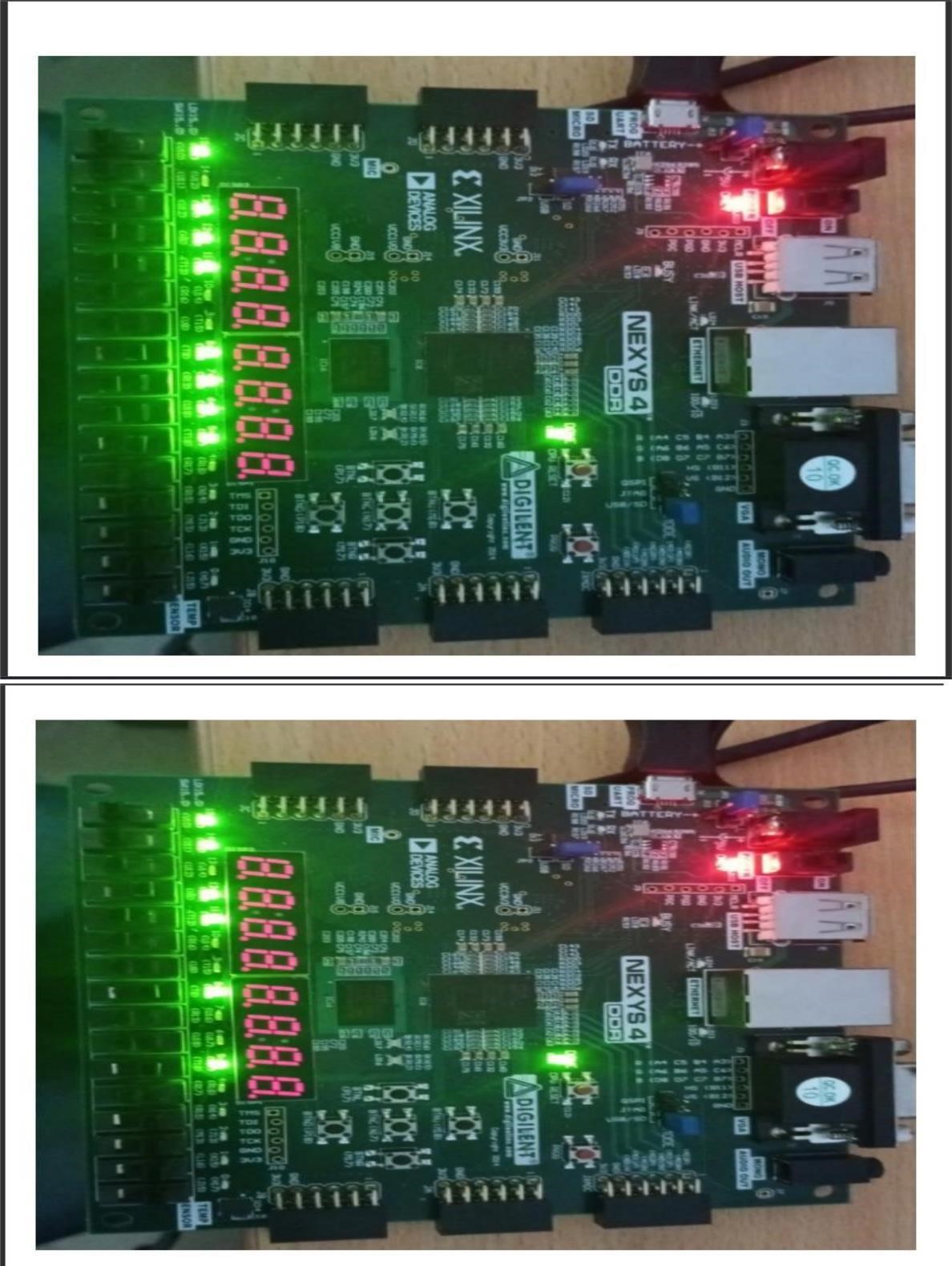
end

end module

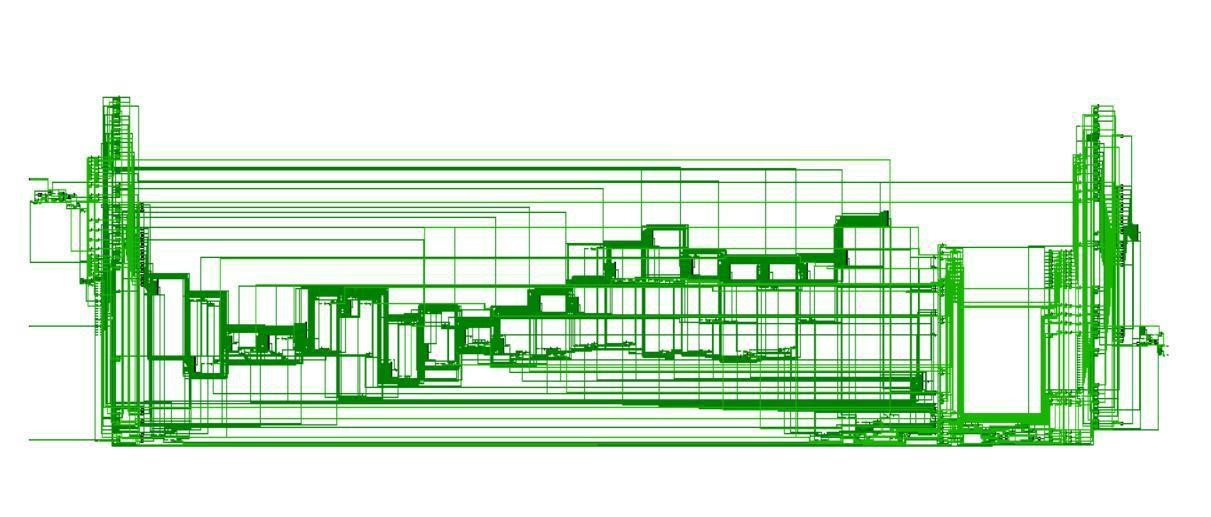
# 9.WAVEFORM:-



# 10.FPGA RESULTS:-



# 11.ELABORATED DESIGN:-



# 12.CONCLUSION:-

In conclusion, the Dijkstra algorithm is a highly efficient and effective algorithm that can be used to find the shortest path in a graph. The Verilog implementation of the algorithm is an excellent way to understand how the algorithm works and can be used to optimize its performance.

Through the implementation of the algorithm using Verilog, we were able to demonstrate the versatility and power of hardware description languages. The Verilog implementation allowed

In addition, the Verilog implementation of the Dijkstra algorithm can be extended to incorporate additional features such as parallel processing and dynamic programming. These features can further enhance the performance of the algorithm and make it suitable for a wide range of applications.

Overall, the Verilog implementation of the Dijkstra algorithm is a highly valuable tool for anyone interested in understanding the algorithm and its implementation. It provides a great opportunity to learn about hardware description languages and their potential applications in various fields.

# 13.REFERENCE:-

* MIT OpenCourseware, 6.046J Introduction to Algorithms.

< [http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-ComputerScience/6-046JFall-2005/CourseHome/>](http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/CourseHome/) Accessed 4/25/09

* Department of Mathematics, University of Melbourne. *Dijkstra’s*

*Algorithm.*

[*<*http://www.ms.unimelb.edu.au/~moshe/620-](http://www.ms.unimelb.edu.au/~moshe/620-261/dijkstra/dijkstra.html)

[261/dijkstra/dijkstra.html >](http://www.ms.unimelb.edu.au/~moshe/620-261/dijkstra/dijkstra.html) Accessed 4/25/0

THANK YOU