Hardware Accelerator for Dijkstra Algorithm

Ibrahim Rupawala
irupawal@asu.edu

Saranya Govinda Raj
sgovin10@asu.edu

Vijayan Ramesh
vrames13@asu.edu

***Abstract*—Single source shortest path (SSSP) is a fundamental problem in graph theory. One of the approaches to solve this is DIJKSTRA Algorithm. However, the software implementation of this algorithm is time consuming. In this paper, we propose a hardware accelerator implementation on field-programmable gate array (FPGAs) to solve this problem. The experimental results of our Hardware accelerator for Dijkstra algorithm on FPGA can achieve a speedup of 10× over the Software implementation.**

Keywords— Single source shortest path (SSSP); Dijkstra Algorithm ; Hardware accelerator; FPGA ;

I. INTRODUCTION

Single source shortest path (SSSP) is a fundamental problem in graph theory. It requires finding the path of minimum total weight, given a source vertex s and a destination vertex t in a given n-vertex, m-edge directed graph G with the given edge weights. The most well-known and classic algorithm to solve the SSSP is Dijkstra's Algorithm.

Dutch computer scientist *Edsger Dijkstra* conceptualized a graph search algorithm and famously known as Dijkstra's algorithm these days. This algorithm can handle data with cycle and non-negative edges. Dijkstra's algorithm is the most efficient sequential implementation in terms of processing speed since it runs in time linear with the number of edges but it requires many iterations and is difficult to parallelize. In contrast, the Bellman–Ford algorithm involves fewer iterations, and each iteration is highly parallelizable. However, the algorithm may process each edge multiple times.

II. MOTIVATION

It is widely used in network routing, path planning in mobile robots, urban transportation, and many other fields. The motivation for our work was "maze routing" – as an example, a specific application of such an algorithm could be in CAD where conductive tracks on a printed circuit board must be routed between pins of components without overlapping or crossing other tracks.

. Dijkstra's algorithm in software is implemented largely with the help of a priority queue, which is a sequential data structure. As the number of nodes in a network increase, the advantages of parallelizing this queue become apparent, given the quadratic growth of the algorithm. We wanted to exploit this and implement an efficient parallel version of the Algorithm on hardware.

III. DIJKSTRA'S ALGORITHM

Dijkstra's algorithm extracts the minimum cost path from any given destination to the source node. The input to the algorithm consists of a directed graph G = (V, E) and a positive weight ei (representing distance) associated with each arc ei Є E. V the is the set of all vertices in the graph G. Each edge of the graph is an ordered pair of vertices (u, v) and represents a connection from vertex u to vertex v. The set of all edges is denoted by E. Dijkstra's algorithm works by visiting vertices in the graph starting with the object's starting point. It then over and again examines the closest not-yet-examined vertex, adds its vertices to the set of vertices to be examined. It expands outwards from the starting point until it reaches the goal. The algorithm works by keeping for each vertex v, the cost distance[v] of the shortest path found so far from source vertex.

When the algorithm finishes, d[v] will be the cost of the shortest path from s to v or infinity, if no such path exists. The basic operation of Dijkstra's algorithm is *edge relaxation*: if there is an edge from u to v, then the shortest known path from s to u can be extended to a path from s to v by adding edge (u, v) at the end. This path will have length d[u] + c (u, v). If this is less than d[v], the current value of d[v] is replaced with the new value. Edge relaxation is applied until all values d[v] represent the cost of the shortest path from s to v. The algorithm is controlled so that each edge (u, v) is relaxed only once, when d[u] has reached its final value. The algorithm maintains a flag register (flag[v]) for each vertex, which is set to 1, if the shortest paths from the source have already been determined. When flag[u] is set to 1, the algorithm relaxes every outgoing edge (u, v).

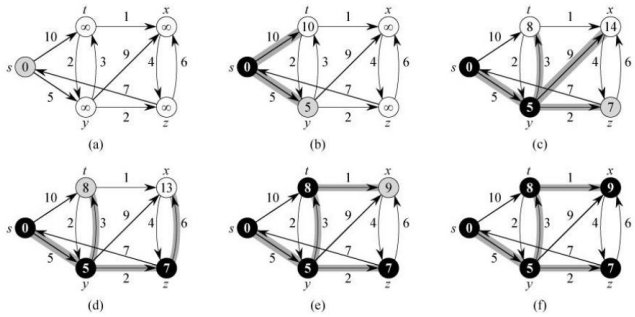

Figure 1. The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority

IV. Design Implementation

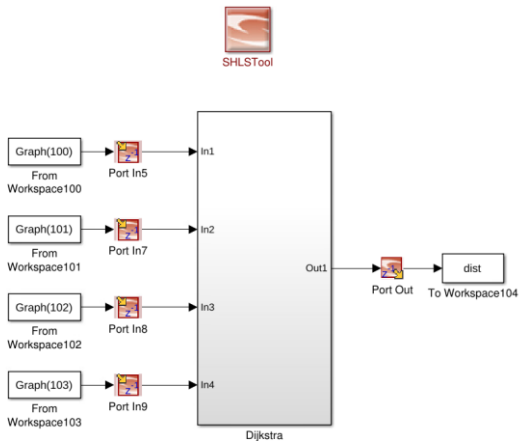

Figure 2: Hardware implementation of basic algorithm using SMC, indicating inputs taken from workspace.

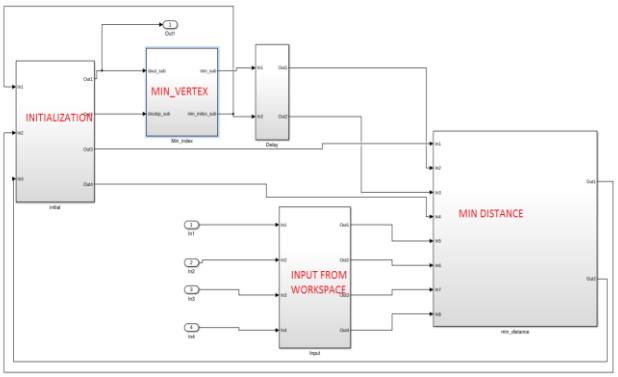

Figure 3: The basic algorithm implemented using sub-blocks

The basic algorithm was implemented using Synphony Model Complier blocks. The basic implementation consists of 5 major blocks

i. Input Block

ii. Initialization of arrays

iii. Min Vertex Block

iv. Min Distance Block

v. Delay Block

Input Block provides the weights of the nodes from the workspace using a vector matrix, where weight represents the distance between the connected nodes. During each cycle a single element (that is weight of the node) from each row is been fetched to the design to the Min Distance block

The design consists of two sets

sptSet (shortest path tree set) keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized

distanceSet will have the distance from the source vertex to the node of interest and will finally have the minimum distance from the source vertex to all other vertices. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

Initialization of arrays block will initialize these two sets. The *sptSet* is initialized to all 0s that is all the vertices are marked as unvisited initially. At the beginning of the algorithm the source vertex is updated to 1 and is being marked as the node of interest. Hence now the distance from this vertex to the neighboring vertex needs to be calculated. The *distanceSet* initializes all distances from the source vertex as INFINITE indicating that the distance from the source vertex to those particular nodes have not been found yet. It also initializes the distance of source vertex as 0 as the distance of the source vertex from itself is zero.

Min Vertex block calculates the vertex closest to the node of interest (initially the source vertex) from the set of vertices not yet visited (that is not included in *sptSet*). The minimum vertex obtained is made as the next node of interest.

The output of the Min Vertex block that is the node of interest has to be held constant until the distance of all nodes connected to the node of interest is calculated this is achieved by the Delay Block. In this particular design the node of interest is held constant for 4 clock cycles to ensure that the connection of the node of interest to all other nodes is being verified and the minimum distance of the same to other nodes is being checked.

Minimum Distance block calculates the minimum distance from the node of interest to all its connected nodes and updates the *distanceSet* matrix with the new values. There are several conditions which are checked in this particular block. First, the distance is calculated only if the vertex is not included in *sptSet* matrix. Second, the connection of the node of interest to all the other vertices in the graph is being verified. Third, it ensures that the distance of the node of interest is not marked as infinite. Finally, it ensures that sum of the distance of the node of interest from the source and the distance of the connected neighboring nodes from the node of interest is smaller than the distance of the connected node from the source node. If all the above conditions are satisfied, the distance of the neighboring connected node from the node of interest is calculated. The distance matrix is then updated with these new distances of the nodes connected to the node of interest from the source node.

The output obtained from the Minimum Distance node is fed again to the Min Vertex block and the Min vertex from the set of unvisited nodes will be computed again.

The steps above mentioned are repeated iteratively for the number of times equal to the number of nodes in the graph. In the end, the *distanceSet* matrix is updated with the distance of all the vertices in the graph from the source node and the *sptSet* matrix is updated with all the nodes as visited.

V. Architectural Optimization

Parallelism of Dijkstra increases the throughput. Parallelism is using additional functional units which process instructions in a parallel manner. By this way, the number of instructions processed in a given amount of time increases and thereby the throughput is increased manifold. The amount of parallelism depends on the application. An application of Dijkstra shortest path algorithm is the spanning tree.

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. In a software implementation of the dijkstra for MST, the tree has to be traversed in a sequential manner from one node to the other. The minimum distance is also calculated in a sequential manner. A regular spanning tree will look like the image below,

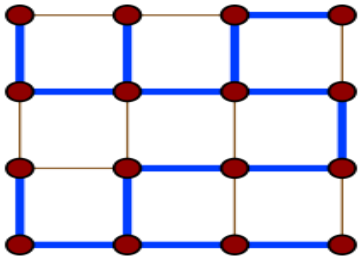

Figure 4: Spanning Tree

The nodes are connected in a way that it will not create cycles. The advantage of using this structure is that it avoids redundancy and having the goal as sending the packet to the destination only.

The complexity of the software implementation of Dijkstra algorithm increases with increasing vertices in the tree. With parallelism on hardware, we can reduce this complexity by having smaller parallel blocks. A major advantage of parallelism is we can reduce the VDD of the network for the given throughput to have increased power efficiency. ASIC implementation of the algorithm with varied VDD in the results section will provide a greater insight on the power efficiency obtained. Parallelism adds small amount of latency to the circuits as the outputs obtained from the various blocks has to be computed to obtain the final result. Parallelism have direct impact on the throughput of the implementation. The amount of parallelism depends on the extent of independent functionalities in the design and also the availability of resources. Compared to the other architectural technique like pipelining, parallelism is rather easier to implement and provides good performance improvement.

A. Architectural optimization 1

In the architectural optimization 1 a 16 vertex matrix is fed to the SMC blocks. These blocks will be processed by 4 parallel blocks. So as soon as the values of the first block is obtained, the parallel blocks will provide outputs for the computation of the next stages. Since they are readily available we added a delay and an adder unit to add the values from the minimum distance found in the previous stage. This ensures non-wastage of clock cycles and increased throughput. With the tradeoff of increased energy due to a large amount of devices added, the throughput is increased which makes it suitable for data transmission intensive applications.

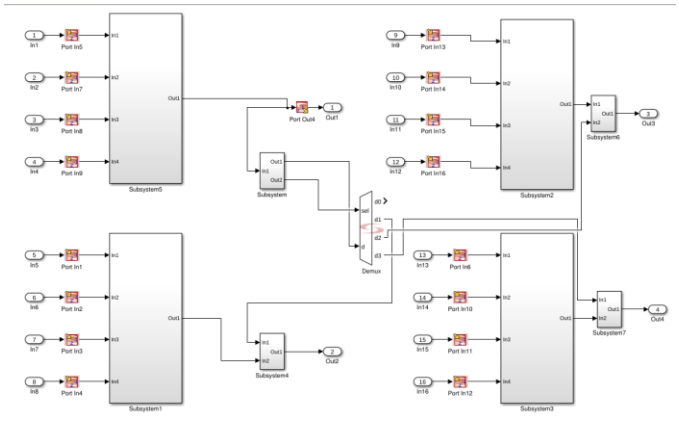

Figure 5: Architectural Optimization - 4X Throughput

B. Architectural optimization 2.

In this optimization technique, we have increased the througphut further by Increasing the amount of parallellism 4x times with additional parallel units implemented as shown below.

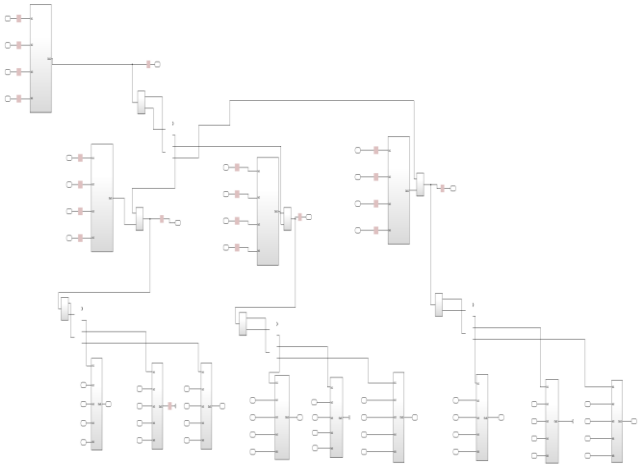

Figure 6: Architectural Optimization - 16X Throughput

A graph with 52 vertices can be fed into the circuit simultaneously and the outputs will be obatained almost at the same time. The advantage of this circuit is any large network can be sampled and the minimum path in spanning tree can be obtained without delay. The area and energy of the SMC blocks increases due to the additional hardware units. The execution time is reduced than the equivalent software version due to the parallel implementation of several blocks.

VI. Implementation Results

A. FPGA Implementaion:

The design was implemented using Vertex 7 XC7VX485T-2FFG1761 FPGA target and the resource utilization of the design was obtained using synplify_pro compiler.

Project Settings			
Project Name	Dijkstra_Final_Working	Implementation Name	rev_1
Top Module	Dijkstra_Final_Working	Pipelining	1
Retiming	1	Resource Sharing	1
Fanout Guide	10000	Disable I/O Insertion	0
Disable Sequential Optimizations	0	Clock Conversion	1

Run Status

Area Summary			
I/O ports (io_port)	302	Non I/O Register bits (non_io_reg)	748 (0%)
I/O Register bits (total_io_reg)	0	Block Rams (v_ram)	0 (1030)
DSP48s (dsp_used)	0 (2800)	LUTs (total_luts)	795 (0%)
Detailed report		Hierarchical Area report	

Timing Summary			
Clock Name (clock_name)	Req Freq (req_freq)	Est Freq (est_freq)	Slack (slack)
clk	4.0 MHz	158.5 MHz	243.690
Detailed report		Timing Report View	

(a)

Area Summary			
I/O ports (io_port)	1205	Non I/O Register bits (non_io_reg)	3177 (0%)
I/O Register bits (total_io_reg)	0	Block Rams (v_ram)	0 (1030)
DSP48s (dsp_used)	0 (2800)	LUTs (total_luts)	3470 (1%)
Detailed report		Hierarchical Area report	

Timing Summary			
Clock Name (clock_name)	Req Freq (req_freq)	Est Freq (est_freq)	Slack (slack)
clk	4.0 MHz	158.5 MHz	243.690
Detailed report		Timing Report View	

(b)

Area Summary			
I/O ports (io_port)	3923	Non I/O Register bits (non_io_reg)	10403 (1%)
I/O Register bits (total_io_reg)	0	Block Rams (v_ram)	0 (1030)
DSP48s (dsp_used)	0 (2800)	LUTs (total_luts)	12019 (3%)
Detailed report		Hierarchical Area report	

Timing Summary			
Clock Name (clock_name)	Req Freq (req_freq)	Est Freq (est_freq)	Slack (slack)
clk	4.0 MHz	154.3 MHz	243.518
Detailed report		Timing Report View	

(c)

Figure 7: Resource utilization of the a) Basic dijkstra algorithm b) With 4*throughput c) with 16*throughput for the same frequency

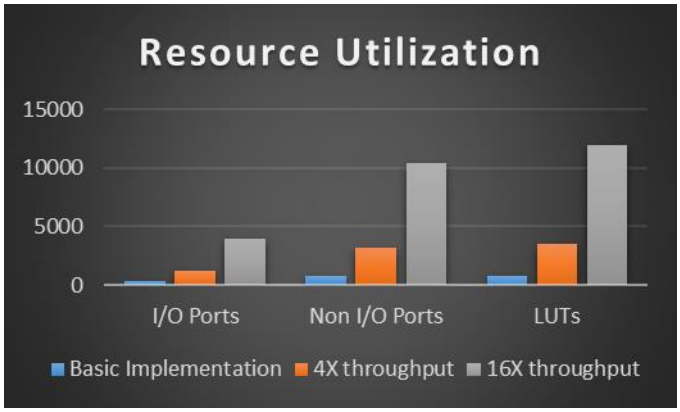

Figure 8: The graph indicates the resource utilization based on the parallelism extent.

As seen in the above comparison graph, It clearly indicates that the number of units increases with parallelism and which inturn provides increased throughput.

B. ASIC Implementaion:

The design was implemented with basic implementation, architectural optimization for 4*throughput and 16*throughput on 28nm ASIC target and the following results were obtained.

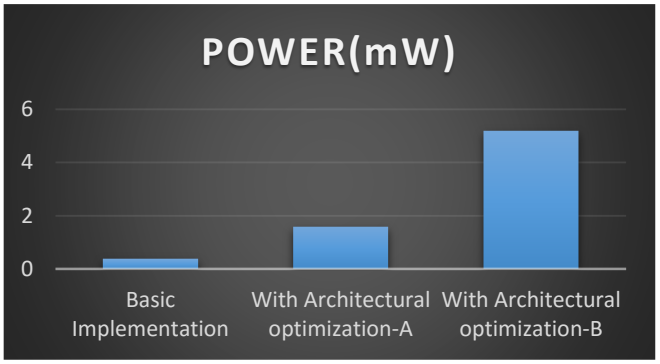

Figure 9: Power comparison for the various dijkstra implementation.

Power increases with parallelism as the number of hardware units increases.

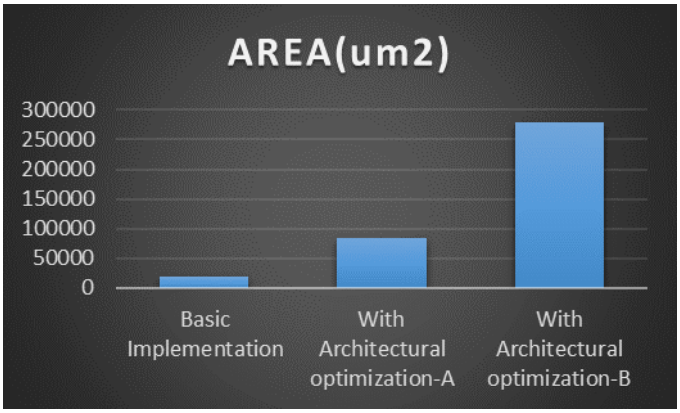

Figure 10: Area comparison for the various dijkstra implementation. Area increases with parallelism as the number of hardware units increases

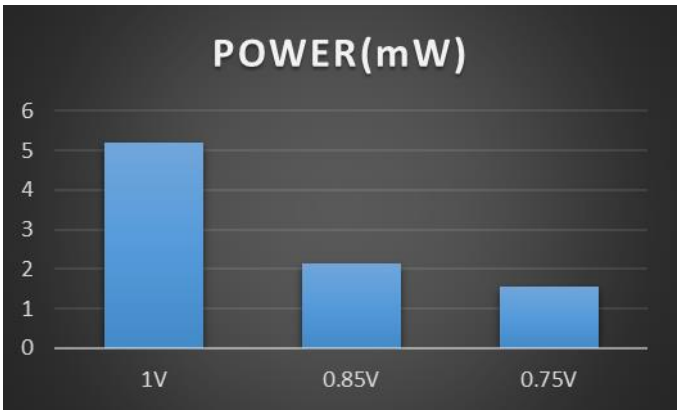

Figure 11: Power comparison of the optimized architecture with circuit level optimizations (VDD, VTH)

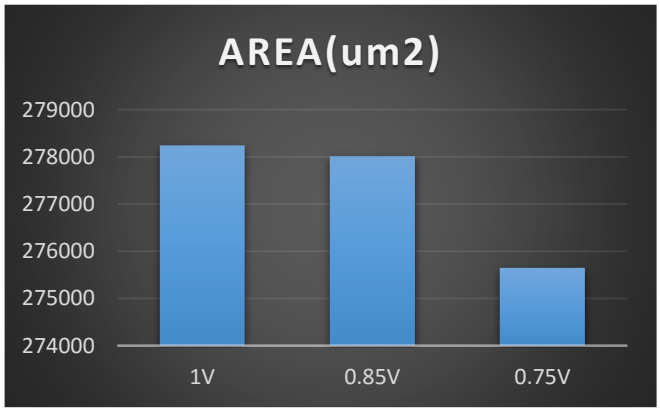

Figure 12: Area comparison of the optimized architecture with circuit level optimizations (VDD, VTH)

The simulations using Sizing, VDD and VTH variations for the ASIC implementation, we can observe that operating the device at .75V VDD achieves least power and area for the same throughput. This is because as VDD decreases the leakage energy decreases and with sizing optimization area decreases.

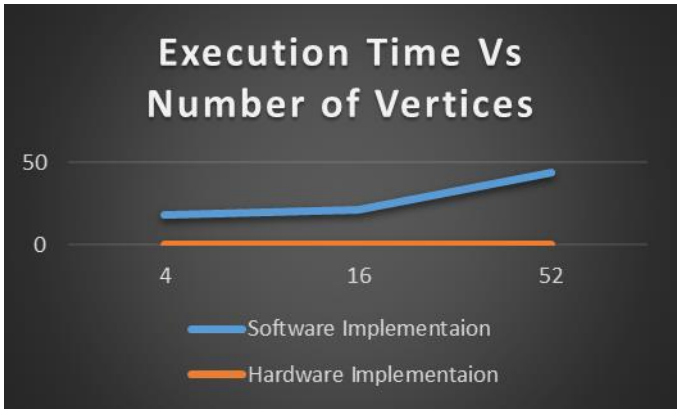

Figure 13: Performance comparison with increased load

Execution time for the software simulation increases rapidly with increased number of vertices while by applying parallelism to the basic hardware implementation the execution time remains constant even with increased number of vertices at the expense of increased energy and area.

VII. Conclusion

The Hardware accelerator for Dijkstra Algorithm can perform much faster than the software counterpart which can be highly beneficial for applications like path planning in mobile robots, routing packets in telecom networks and segmentation in image processing. The architectural optimization helps in increasing the throughput while maintaining the frequency of operation. This can be attributed to the following factors: multiple assignments to variables are executed concurrently, multiple arithmetic operations, including comparisons, are executed in parallel and the data structures and tables are implemented in the internal memory blocks. The implementation can be further improved by interleaving of inputs.

Acknowledgment

We would like to extend our gratitude to Professor **Fengbo Ren** for his support and guidance for this project. The concepts he taught in this class and the tutorials provided really helped in completing this project successfully.

We would like to thank Teaching Assistant **Yixing Li** for her timely inputs and enthusiasm in answering our queries which helped us whenever we encountered a deadlock in our project.

The Department of Computer Engineering which provided with the lab facilities for our project work have to be appreciated without which all this would have just been on paper.

References

The following papers gave a lot of insight in going forward with our project.

[1] DIJKSTRA algorithm implementation on fpga card for telecom calculations. Feb. 2013. ISSN: 2231 – 6604 Volume 4, Issue 2, pp: 110-116 ©IJESET

[2] An FPGA Implementation for Solving the Large Single-Source-Shortest-Path Problem, VOL. 63, NO. 5, MAY 2016

[3] Research of shortest path algorithm based on the data structure IEEE-6269416.

[4] I. Fernandez, J. Castillo, C. Pedraza, C. Sanchez, J. Ignacio Martinez, "Parallel Implementation of The Shortest Path Algorithm on FPGA", 2008 4th Southern Conference on Programmable Logic, pp.245-248, 26-28 March 2008.

[5] K. Sridharan, T. K. Priya, P. R. Kumar, "Hardware Architecture for Finding Shortest Paths", TENCON 2009 - 2009 IEEE Region 10 Conference, pp.1-5, 23-26 Jan. 2009.

[6] G.R. Jagadeesh, T. Srikanthan, C.M. Lim"Field programmable gate array-based acceleration of shortest-path computation",IET Comput. Digit. Tech., 2011, Vol. 5, Iss. 4, pp. 231–237.

[7] E. W. Dijkstra, "A note on two problems in connection with graphs", Numerische Mathematik, vol1.1, pp.269–271, 1959

[8] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen, "Low- Power CMO S Digital Design," IEEE J. Solid- State Circuits, vol. 27, no. 4, pp. 473- 484, Apr. 1992.