EE705 -VLSI Design Lab

Course Project Report

Implementation of Single Source Shortest Path on FPGA using Bellman-Ford algorithm

Submitted by

Name	**Roll Number**
Arjun Raj	193079014
Aswathi E.	19307R014
Nikhil Ajith	193079027
Robin James Payyappillil	193079026

Contents

1 Implementation of Bellman – Ford Algorithm

1.1 Introduction

Reference [1] suggests an efficient implementation of Single Source Shortest Path using Bellman Ford algorithm. Data parallelism has been exploited, to concurrently process multiple edges in each clock cycle, regardless of the data dependencies.

Graphs are important to represent real world data. We use FPGA methods to solve the graph problem. On – chip memory is used in this approach, since the demonstration only uses a subset of the graph . Single-Source Shortest Path (SSSP) is a fundamental graph algorithm, which finds the shortest paths from a source vertex to all other vertices in the graph. Many applications require high speed Single Source Shortest Path computations. There are many implementations done for SSSP computations, the reference [1] uses Bellman – Ford Algorithm. It relaxes the weights of an edges in an iterative manner until shortest path to all vertices in the graph are computed. The computation complexity under the worst case is $O(v.e)$, where v is the number of vertices and e is the number of edges. The implementation uses an early – stop logic, since in real practice the number of iterations are much less. The algorithm used by Bellman – Ford is shown in the below figure.

Let $G = (V, E)$ denote the graph that consists of a set of vertices V and a set of edges E.
Let v denote the number of vertices
Let e denote the number of edges
Let edge(i, j) denote the edge from vertex i to vertex j
Let $w(i, j)$ denote the weight of edge(i, j)
Let $w(i)$ denote the weight of vertex i

Bellman-Ford$(G(V, E))$

1: **for** each vertex x in V **do**
2: **if** x is *source* **then**
3: $w(x) = 0$
4: **else**
5: $w(x) = \infty$
6: predecessor$(x) = null$
7: **end if**
8: **end for**
9: **for** $i = 1$ to $v - 1$ **do**
10: **for** each edge(i, j) in E **do**
11: **if** $w(i) + w(i, j) < w(j)$ **then**
12: $w(j) = w(i) + w(i, j)$
13: predecessor$(j) = i$
14: **end if**
15: **end for**
16: **end for**

Figure 1: Algorithm of Bellman–Ford. (Source : [1])

In this implementation, a pipeline architecture is implemented which process multiple edges in parallel and mitigating the data dependency hazards by data forwarding. Also since the iteration may not happen for v*e number of times, an early termination logic is implemented. If no updates are performed during an iteration, then the computation will be immediately terminated which significantly reduces the run time .

1.2 Implementation

1.2.1 Stage I - Memory Read Edges

We use a graph of 23 vertices, and 31 edges which is stored in on chip memory in FPGA. 4 memory words for p edges are streamed every clock cycle from the on-chip ROM in Memory Read Edge stage which is the first stage in the pipelined architecture as shown in the figure below.

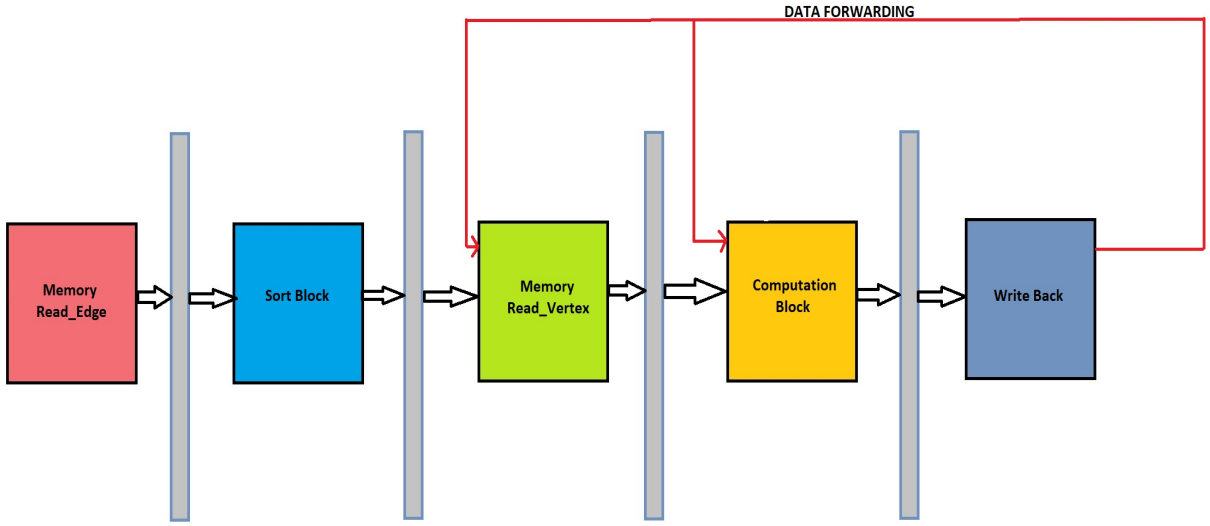


Figure 2: Implementation of Pipelined Bellman – Ford Algorithm.

1.2.2 Stage II - Sorting Block

The input to the sorting block is 4 memory words, each memory word has the weight of the source vertex (w(i)), the weight of the edge between source vertex and destination vertex (w(i, j)) and the indices i and j. These 4 words are sorted in the sorting block to ensure that each destination index of 4 edges will have only one valid update. There are chances that more than one edges target the same destination vertex but produce different update values. For example, memory word $< w(10, 2) = 8, w(10) = 3 >$ and memory word with $< w(6, 2) = 2, w(6) = 4 >$ both target vertex 2, but the possible update values for vertex 2 based on them are 11 and 6, respectively. For each destination vertex, only the minimum update value should be considered (6 in this example). The sorting block is used to identify the possible minimum update value for each destination vertex.

A 1 bit update signal indicates whether the target to be updated is valid or not. Initially the update of all words are set to 1 which indicates all the updates are valid. Each comparator then checks the update signals of 4 memory words. If both the update signals are 1 and the destination vertex is same, then the comparator compares the update values w(i, j)+w(i), and it sets the update value as 0 for the memory word with a larger value. Hence after the sorting block, even if multiple memory words point to same destination vertex, each destination vertex will only have 1 valid update.

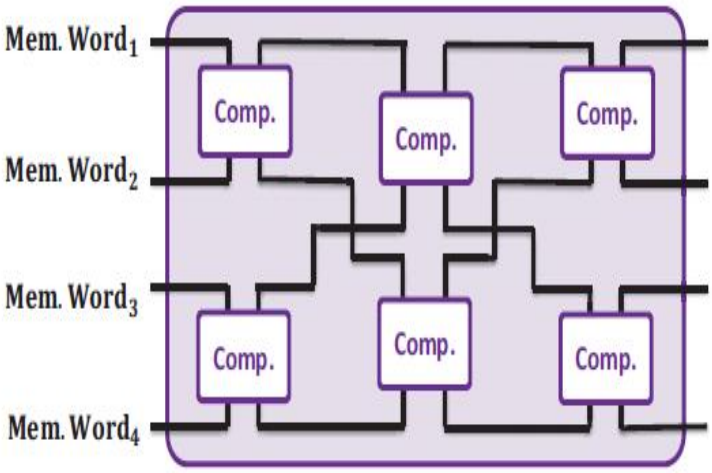


Figure 3: Sorting Block. (Source : [1])

Thus each of the 4 memory words from the output of sorting block will have update signal which indicates whether the target to be updated is valid, w(i) ,the weight of the source vertex , w(i,j) the weight of the edge between vertices i and j and the indices i and j. This is passed on to Memory Read Vertex Stage.

1.2.3 Stage III - Memory Read Vertex

The Read Vertex Stage fetches w(j), the weight of the destination vertex from the ROM. Finally the output of Read Vertex stage that is passed on to computation block will be the vector update, w(i,j), i , j, w(i), w(j) for each memory word.

1.2.4 Stage IV - Computation Block

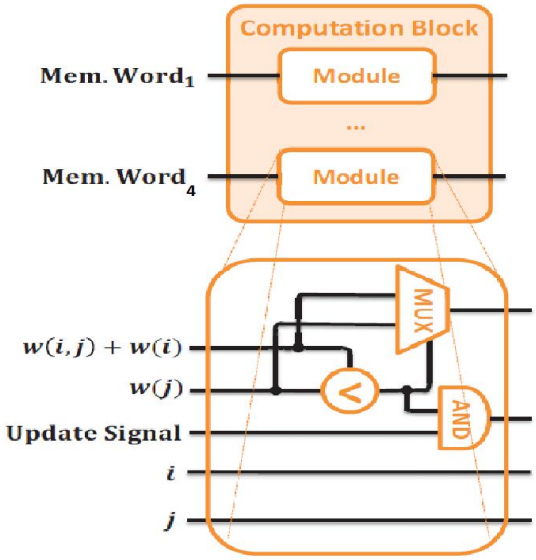


Figure 4: Computation Block. (Source : [1])

The computation block computes $w(i)+w(i,j)$ for each memory word, and the result is compared with the current w(j) using a comparator. If $w(j) < w(i) + w(i,j)$ and the corresponding update signal from the previous stage is 1, then the final update signal is set as 1, otherwise it is kept as 0. Each comparison module as highlighted is responsible for 1 memory word.

1.2.5 Stage V - Memory Write Stage

The memory write stage updates $w(j)$ for each memory word, whose final update signal is set as 1.

If there is no valid update for consecutive iterations, an early stop logic detects this and terminates the computation. Hence as soon as the weights of all vertex converges, the iteration is stopped which helps in significantly reducing the computation time.

1.2.6 Data Forwarding

Data hazard can occur when the same vertex is updated in two consecutive clock cycles. This may result in increasing the weight of vertex. In the figure shown below [1] , the values in the latter clock cycle has $< w(8,6) = 6, w(8) = 11 >$ and w(6) = 20 while the values in the former clock cycle has $< w(10,6) = 3, w(10) = 12 >$and w(6) = 20 in the computation block. The output of computation block will update w(6) to 15, however the value of w(6) in the computation block will still be 20. Thus at the latter clock, the value of w(6) will be over-written as 17 and results in a hazard.

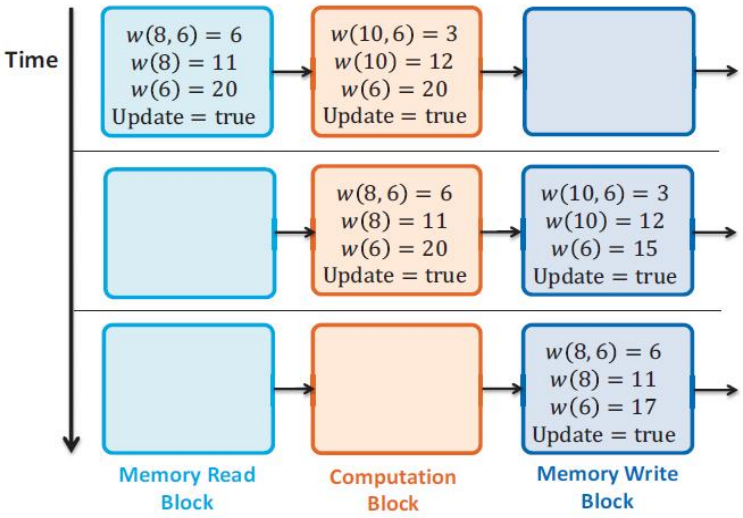


Figure 5: Example of Data Hazard. (Source : [1])

Rather than stalling the pipeline which affects the throughput, data forwarding is implemented to mitigate these hazards as shown in figure below. The output from Write Back stage is forwarded to Computation Block Stage and Memory Read Vertex stage as shown in the figure below [1].

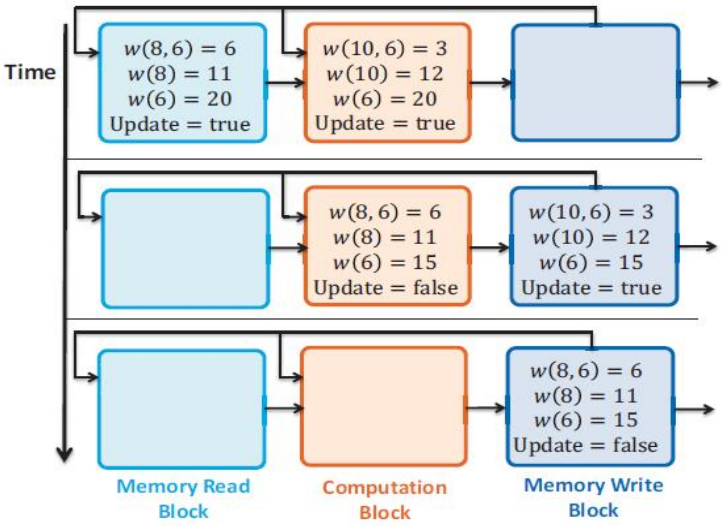


Figure 6: Implementation of Data Forwarding. (Source : [1])

1.3 Detailed Pipelined Architecture

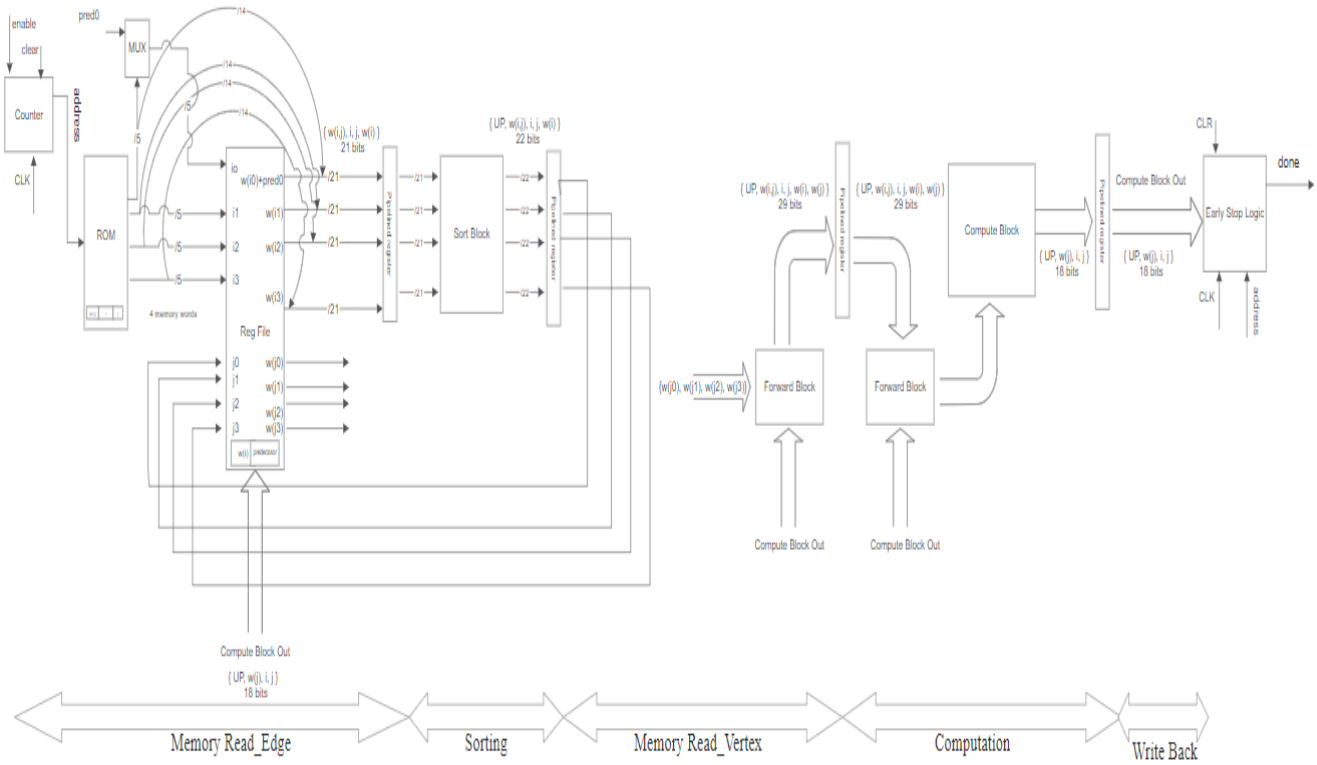


Figure 7: Detailed pipelined architecture

The number of bits taken for the respective elements is tabulated as shown below

Element	Notation	No. of bits
Weight of edge between vertices i and j	w(i,j)	4
Weight of vertex i or j	w(i) or w(j)	7
Vertex index i or j	i or j	5
Update signal	UP	1

The format in which the combination of above elements stored in ROM and Reg File is as follows.

Parameter	Format					
One row in ROM	w(i, j)				i	j
One row in Reg File	w(i)			predecessor		
Input to Sorting Block	w(i, j)			i	j	w(i)
Output to Sorting Block	UP	w(i, j)		i	j	w(i)
Input to computational block	UP	w(i, j)	i	j	w(i)	w(j)

1.4 Sample Graph and Expected Results

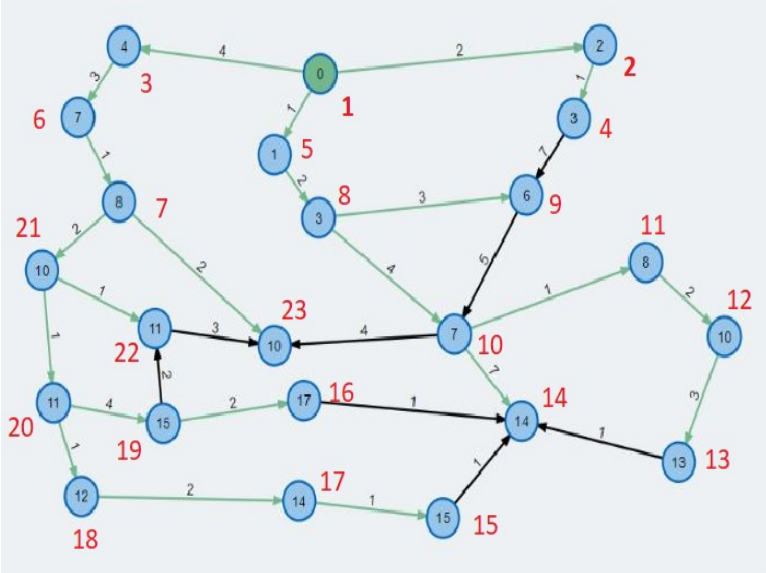


Figure 8: Sample graph with bellman ford algorithm solution (Source : [2])

A graph has been plotted using [2] for verification purpose. The graph consists of a total 23 nodes and 31 edges. The number marked in brown denote the node number and number adjacent to arrows denote edge weights. The value inside the circle denote the final weights of respective nodes upon completion of Bellman Ford algorithm. Upon giving the source node as 1, the lines in green denote the shortest path to respective nodes.

1.5 Simulation Results from ModelSim

A testbench has been created for pipelined Bellman Ford algorithm verification. Upon on simulation on Model Sim, the following results were obtained.

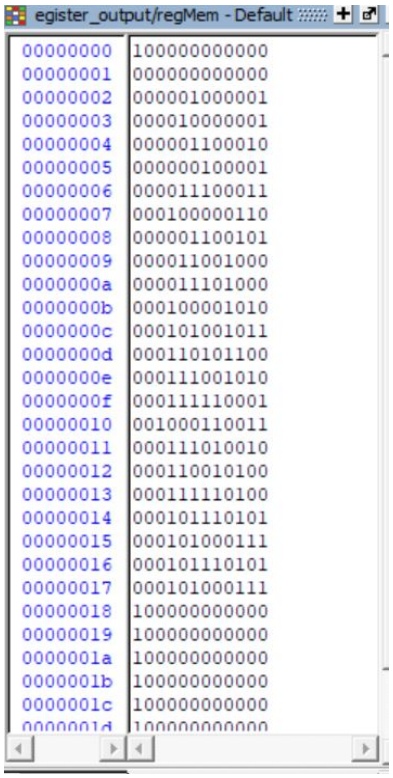


Figure 9: Memory List state of Reg File upon completion of Bellman Ford algorithm

For evaluation purpose let's take destination node as 7.

Node	Content of Reg File	Final wt. of node (7 bit MSB)	Predecessor Addr (5 Bit LSB)
00000007	000100000110	0001000 (8)	00110 (6)
00000006	000011100011	0000111 (7)	00011 (3)
00000003	000010000001	0000100 (4)	00001 (1)

The observed result in simulation is same as that obtained from [2]. Similarly several destination nodes were checked and verified.

1.6 Timing Analysis of pipeline

Model and Clock Period	Setup Slack	Hold Slack	fmax
Low 1200 mV 85C Model and 16 ns	0.550 ns	0.356 ns	64.72 MHz

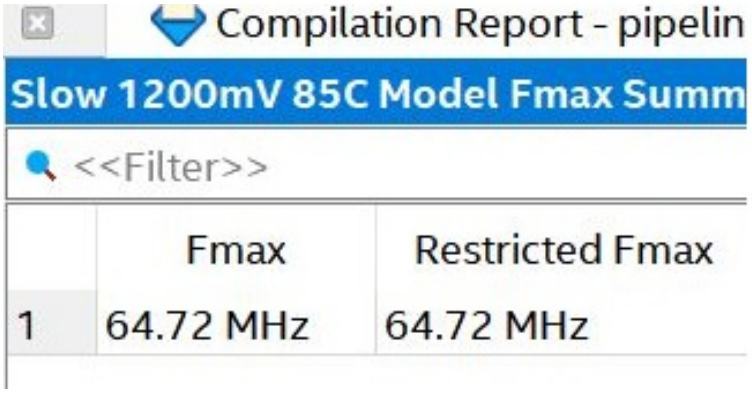


Figure 10: fmax summary

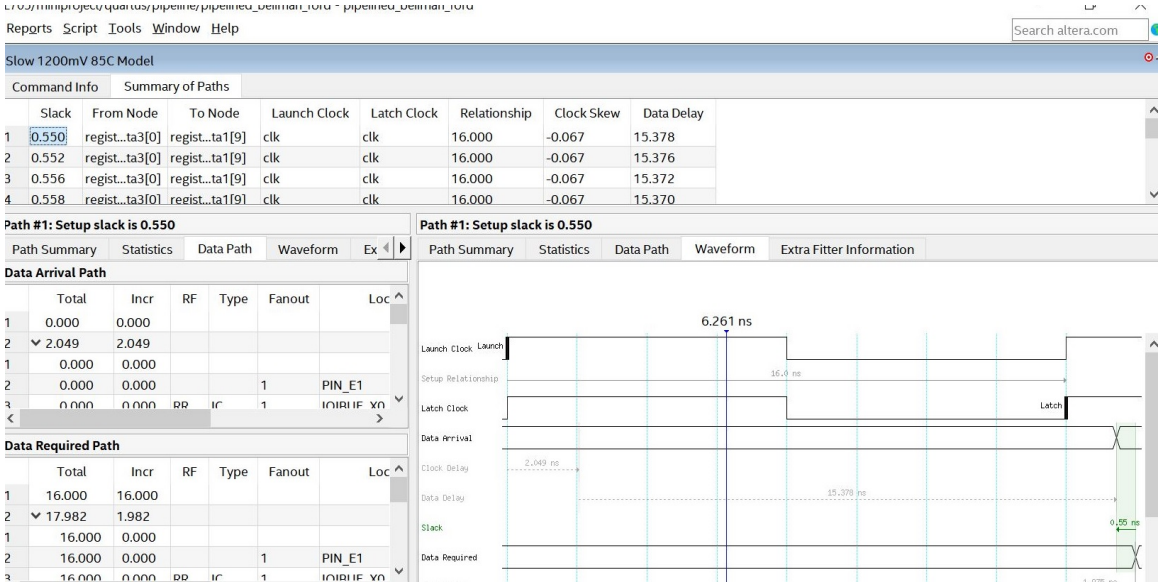


Figure 11: Setup Time Summary

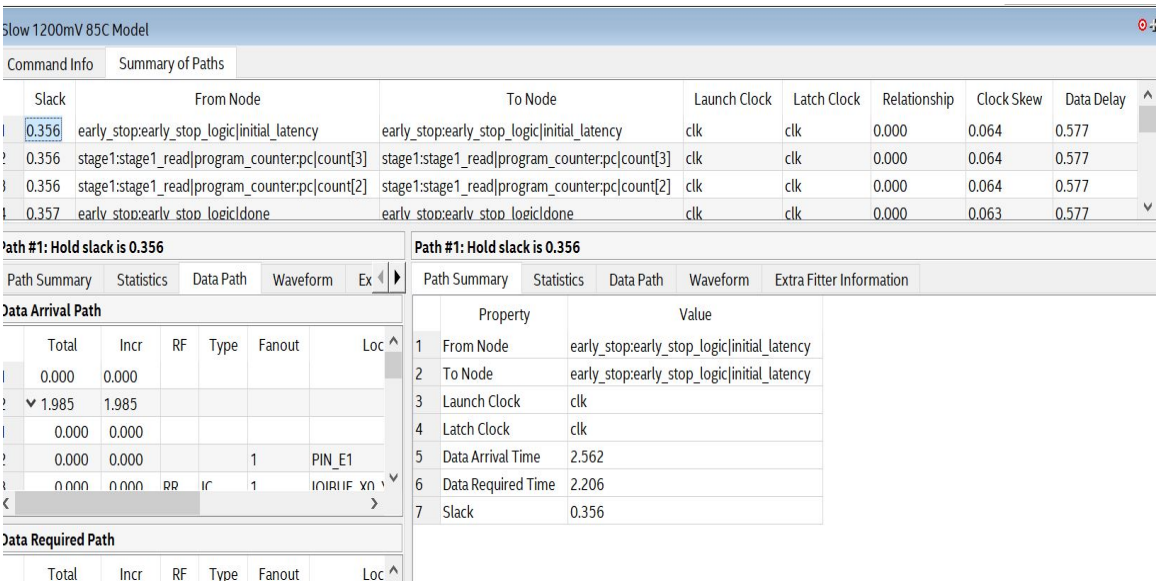


Figure 12: Hold Time Summary

2 Interfacing the VGA display and associated Logic

2.1 Display Section

In order to display the default graph when no computation is done and to display the modified graph after source and destination nodes had been specified, we need the following sub modules. When display_on is not asserted or if register file is cleared (no bit is updated to 1 in register file) then what we see on VGA monitor is the original stored graph. If one or multiple entries of the register file are changed to 1, then the corresponding node and its 8 nearest neighbours in the image will be made white. We are able to see the path (aka shortest path), if we assert/set the bits corresponding to source, destination and intermediate nodes in the register file.

2.1.1 ROM-graph

The graph on which the algorithm needs to be run was drawn using paint. This image was resized to 250x250 and made binary. It is stored in a ROM of size 62500x1. We can access individual pixels with a single 16 bit index 'k', rather than two separate coordinates. 'k' is related to x and y coordinate as follows: $k = x + 250y$

2.1.2 Pixel to index block

This blocks takes this 16 bit value 'k' corresponding to each pixel and generate a 5 bit index. We had made a mapping such that center of each node and its 8 neighbourhood maps to index of the node.And remaining pixels (other than the center of node and its 8 neighbourhood) maps to index valued 0(00000).

2.1.3 Register File (32x1)

It is a sync write and async read register file. Each bit corresponds to one particular node. First bit (for address 00000) is always zero. When 'index' (node number) ,'write_en' is given by FSM , and on positive edge of the clock , that particular node/index in the Register File is updated to 1. We can sync clear the entire Register File by asserting 'clr'. We can read the entry corresponding to each node by giving corresponding index to the read port(async). This circuit is clocked by 50MHz and is decoupled from the VGA block, which runs on 25MHz

2.1.4 VGA Block

VGA uses analog interface (we need analog voltages as input to 3 channels). We used 3 bits for red channel, 3 bit for green channel and 2 bit for blue channel. This can give us maximum of 256 colors. We used R-2R DAC network (although we didn't use it). Since the refresh rate of the screen

was 60Hz and the resolution was 640x480 (800x521- including front and back porch), we had to use 25 MHz (To keep track of individual pixels). When scan pixels in left to right, up to down fashion. Whenever we are in 250x250 (within 640x480- active region)region, we increment k (16 bits corresponding to pixel in 250x250 region) and set a signal named 'video_on', otherwise that signal is kept 0 . This is very crucial ,as it will decide what to be displayed and what not to be displayed on the screen.

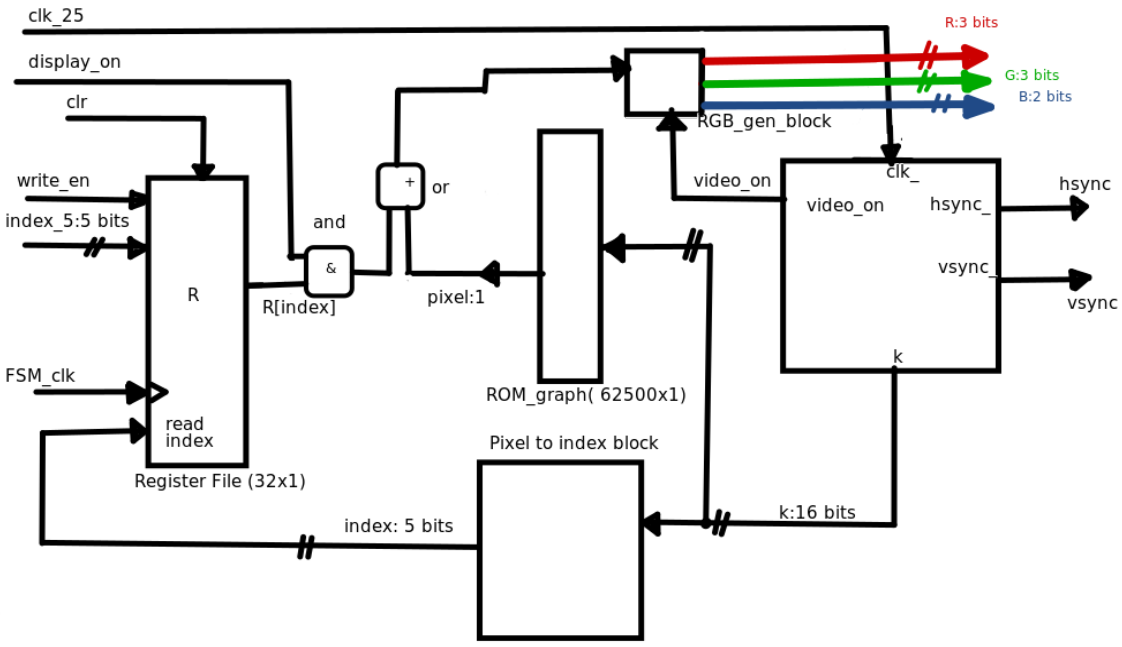


Figure 13: VGA interface and associated logic

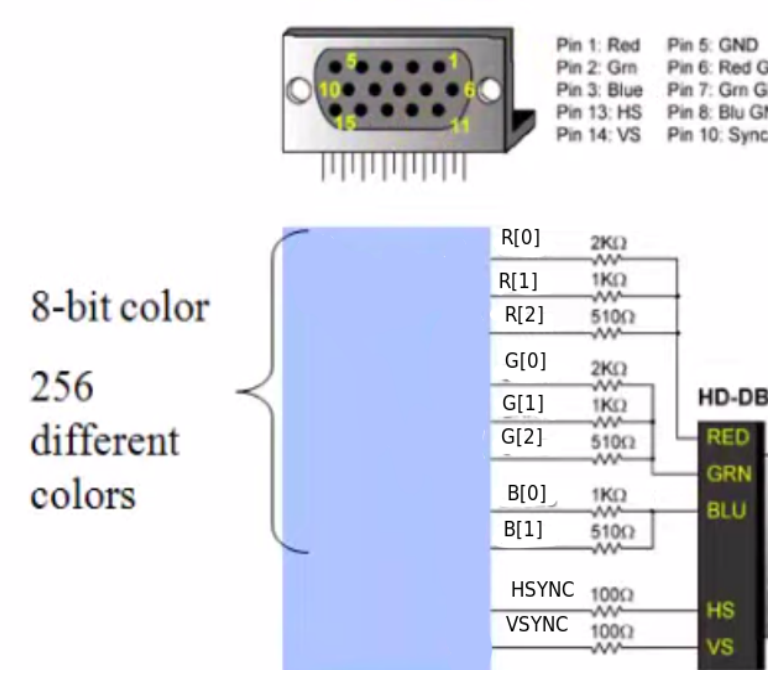


Figure 14: Connection between FPGA pins and VGA monitor. (Source : [3])

3 System Integration

3.1 Finite State Machine(FSM)

VGA interfacing module and module implementing the Bellman-Ford algorithm are integrated using a Moore Model FSM which controls the entire work flow of applying the source address and destination address to displaying the test results on VGA display

FSM is controlled by the on-board 50MHz clock source of DE0-Nano FPGA board. 25MHz clock to the VGA module is derived from the 50MHz global clock using **ALTPLL IP core from the Quartus IP catalog**. 5-bit source address and destination address are applied using external switches .The system has incorporated check-in points using on-chip LEDs which gives an indication of the state of the system

Conceptual state diagram of the system is shown below.

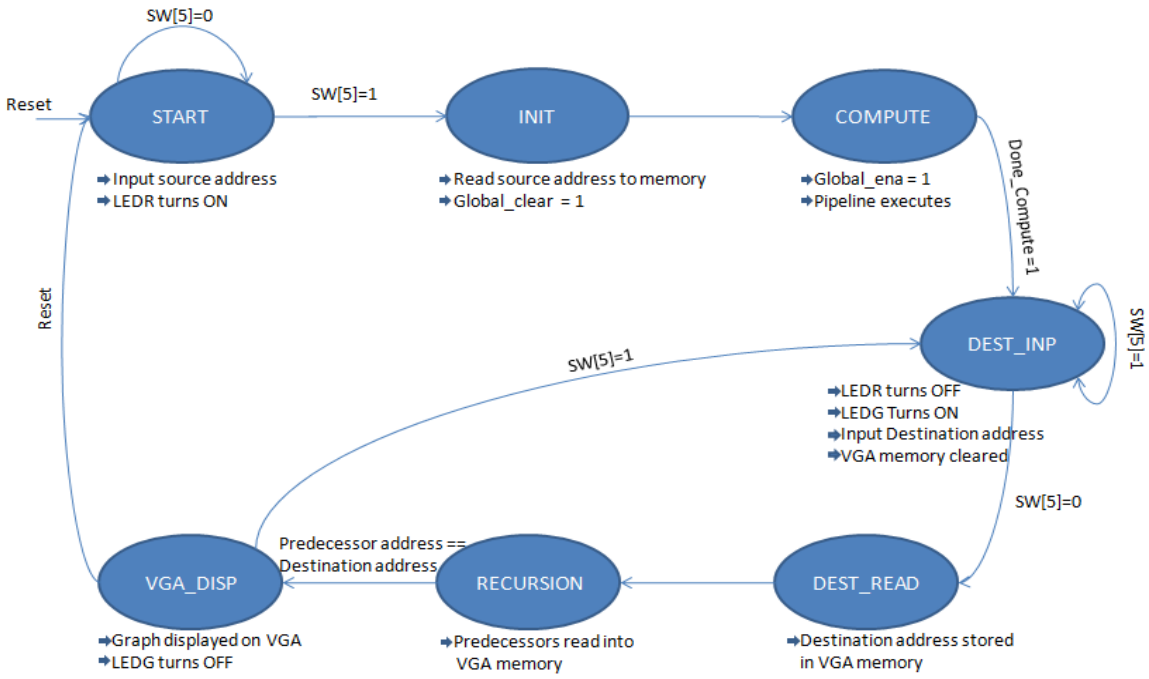


Figure 15: FSM - Conceptual state diagram

State-1 : START

START state is the entry point, accessed by applying the reset signal using on-chip push button. An LED (LEDR) on the board is turned ON indicating the start of the workflow. It remains ON until the Bellman-Ford algorithm is computed.Pre-stored graph is displayed on VGA in START state. User inputs the source address using the switches (SW[4:0]) . After the source address is entered, SW[5] is turned ON to move to the next state-INIT

State-2 : INIT

State INIT is the initialization state where the registers are cleared and source address entered is read into the RegFile.

State-3 : COMPUTE

In the COMPUTE state, the Bellman-Ford algorithm executes and shortest path between source and all the other nodes is computed. End of computation is indicated by a flag signal and user is taken to the next state DEST_INP. LEDR turns off indicating the end of computation of Bellman-Ford algorithm and another on-chip LED(LEDG) is turned ON that signals the DEST_INP state.

State-4 : DEST_INP

User inputs the destination address using 5 switches (SW[10:6]). VGA memory gets cleared in this state. After the destination address is entered , the user turns OFF the switch SW[5] which was turned ON in START state. This takes user to the next state- DEST_ READ.

State-5: DEST_READ

In this state, VGA module is enabled and destination address is read into VGA memory. System moves to the next state-RECURSION.

State-6: RECURSION

The predecessor nodes from the destination to source nodes are found recursively and written to VGA memory block in synchronous with the 50MHz global clock. When the predecessor address is same as the source address, system has identified all the intermediate nodes in the shortest path from source to destination. System moves to the state VGA_DISP.

State-7: VGA_DISP

This is the state where the shortest path between source and destination node is highlighted in the graph displayed on VGA . This completes one cycle which is indicated by LEDG turning OFF. At this point , user can choose to go back to START state or input new destination address. Turning on SW[5] will take the user to DEST_INP state where a new destination address can be entered. To change source address, system is restarted by applying the reset signal from the on-chip push button which lands the system in START state and the whole process is repeated.

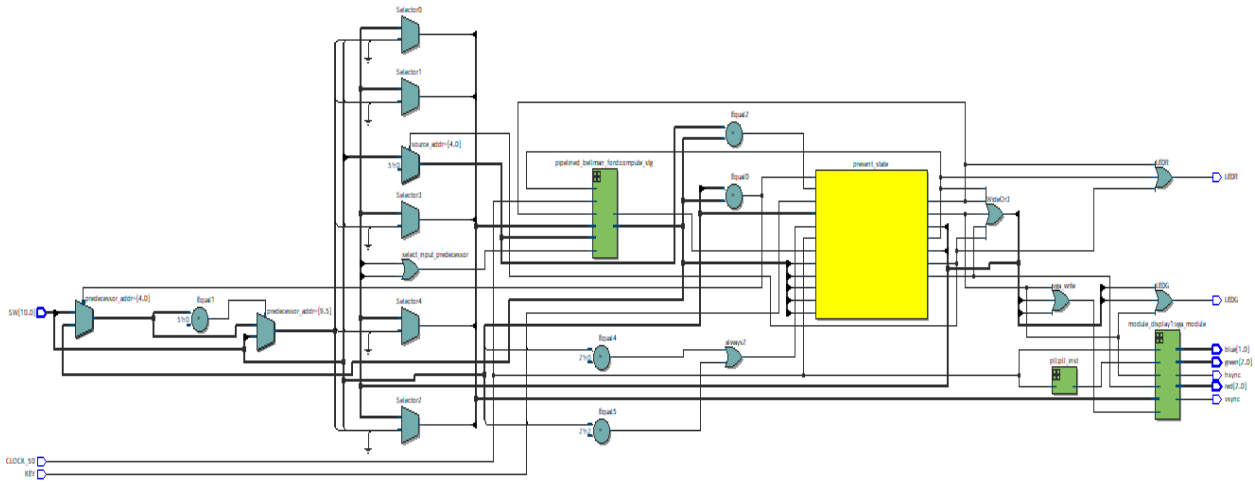
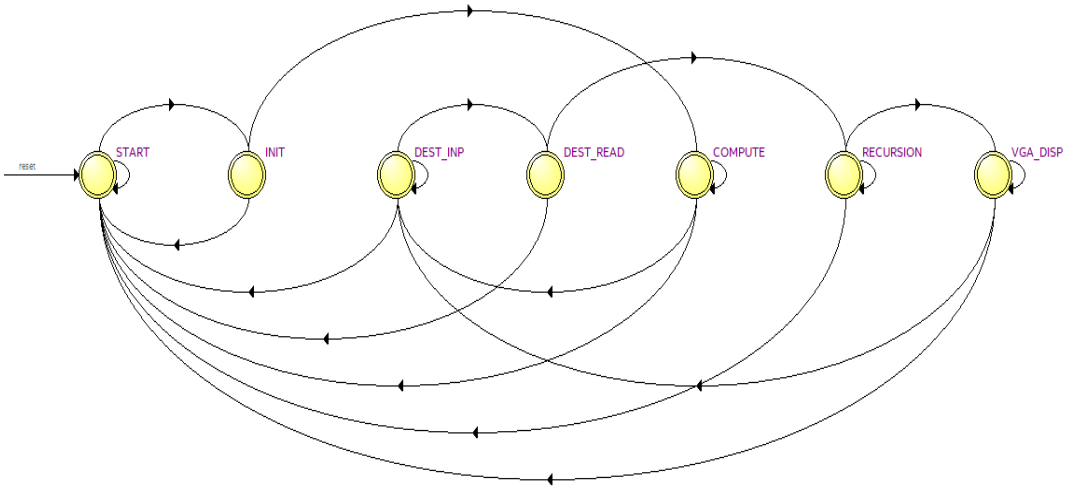


Figure 16: RTL View of the system



	Source State	Destination State	Condition
1	COMPUTE	START	(KEY)
2	COMPUTE	DEST_INP	(KEY).(pipelined_bellman_ford:compute_stg)
3	COMPUTE	COMPUTE	(!pipelined_bellman_ford:compute_stg).(KEY)
4	DEST_INP	START	(KEY)

Figure 17: State diagram shown in State Machine Viewer

3.2 Modelsim simulation test results

3.2.1 RTL simulation

Test cases:

1. Source address = Node-1 , Destination address = Node-7
Predecessors obtained:7 → 6 → 3 → 1

2. User is in VGA_DISP state and enters new destination address. Source address is kept unchanged.

Source address = Node-1 , Destination address = Node-12

Predecessors obtained : $12 \rightarrow 11 \rightarrow 10 \rightarrow 8 \rightarrow 5 \rightarrow 1$

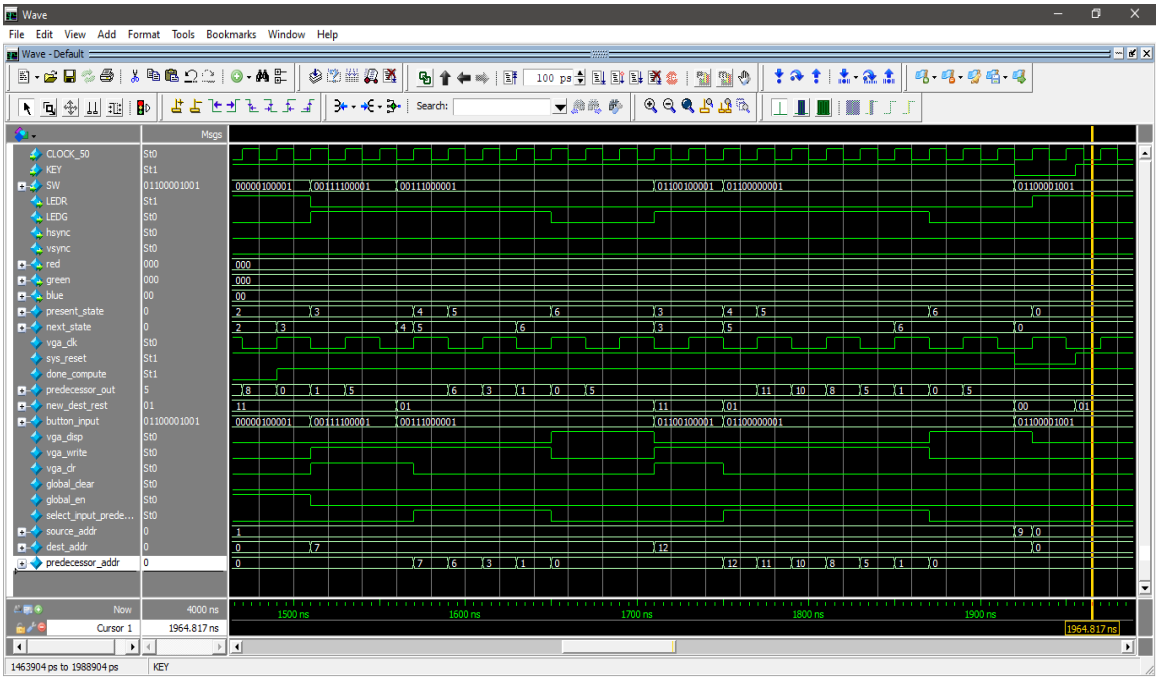


Figure 18: RTL simulation using Modelsim: Test cases - 1 & 2

3. User restarts the process to apply different source address. Source and destination nodes are such that there is no path from source to destination

Source address = Node-9, destination address = Node-2

There are no intermediate predecessor nodes between the selected nodes.Hence, predecessor field will show destination node and source node. In VGA display, only the source node and destination node will be highlighted in the VGA display, no path will be shown.

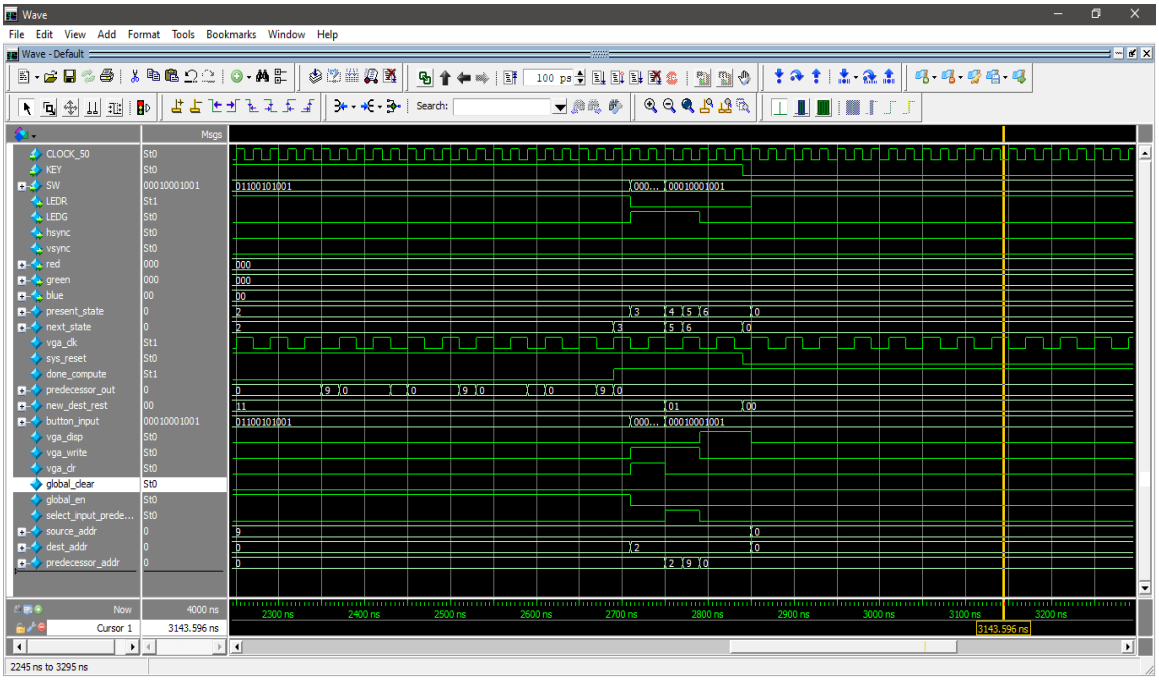


Figure 19: RTL simulation using Modelsim: Test case-3

3.2.2 Gate level simulation

From the timing analysis, maximum clock frequency that can be used in the system is found to be 62.75MHz. Gate level simulation is carried out using 50MHz clock.

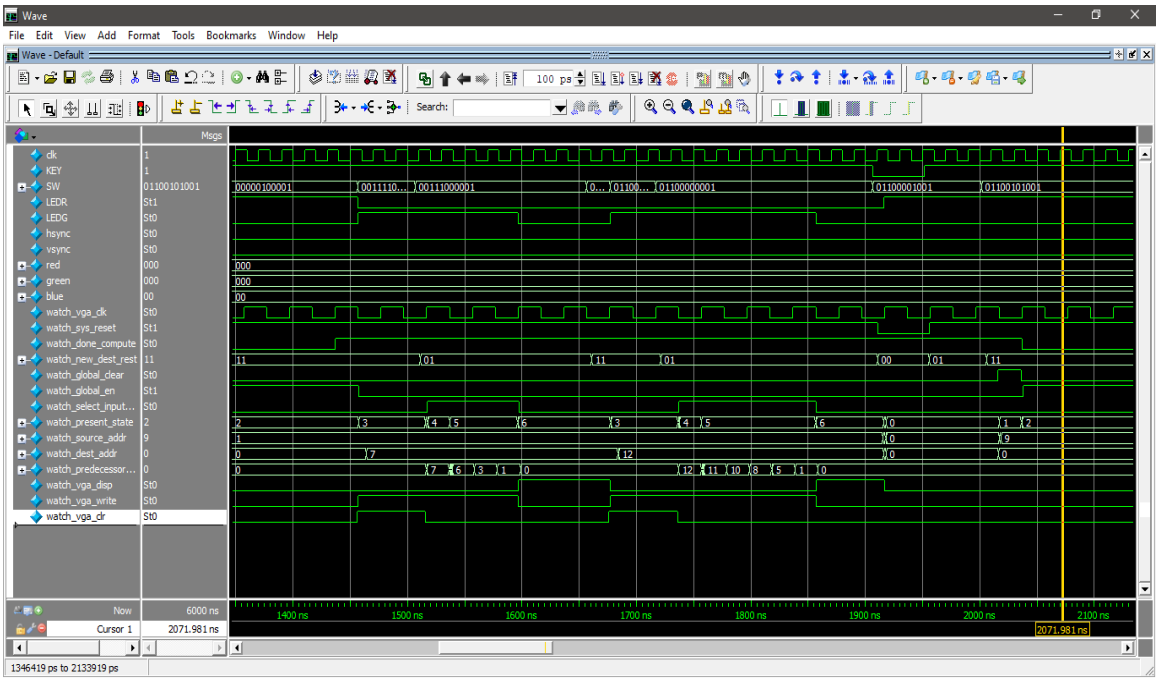


Figure 20: Gate level simulation : Test cases - 1 & 2

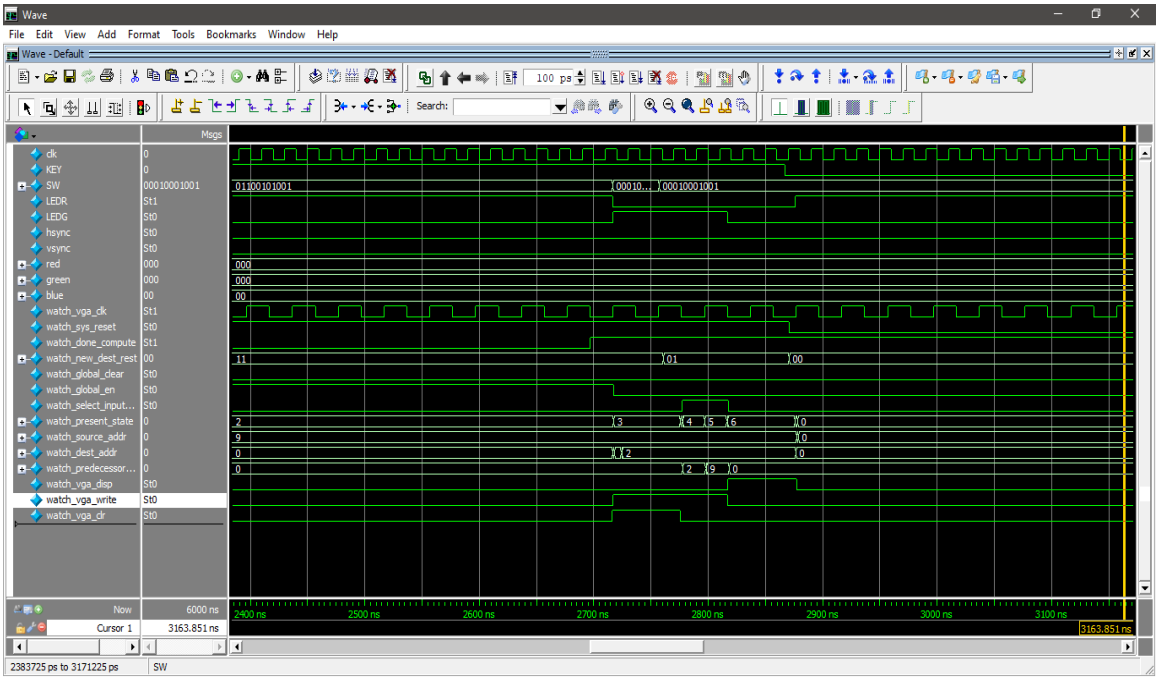


Figure 21: Gate level simulation : Test case-3

3.3 Timing analysis of the system

Clock Status Summary

<<Filter>>

	Target	Clock	Type	Status
1	CLOCK_50	CLOCK_50	Base	Constrained
2	pll_inst|altpll_co...erated|pll1|clk[0]	pll_inst|altpll_co...erated|pll1|clk[0]	Generated	Constrained

Figure 22: Clock status summary

Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name
1	39.71 MHz	39.71 MHz	pll_inst|altpll_component|auto_generated|pll1|clk[0]
2	62.75 MHz	62.75 MHz	CLOCK_50

Figure 23: Fmax summary

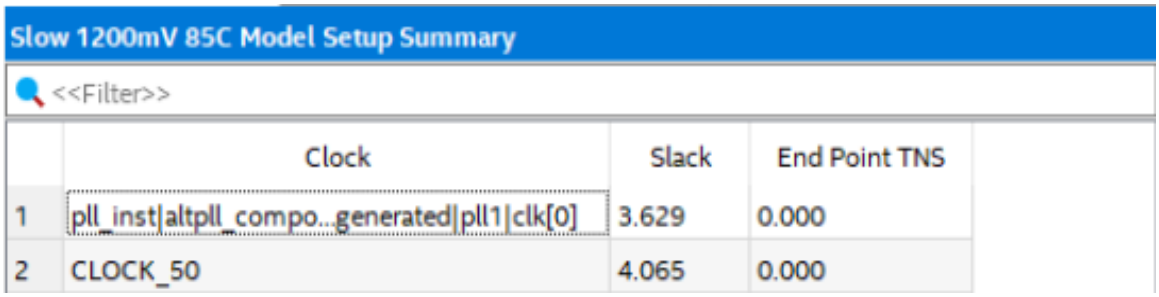


Figure 24: Setup summary

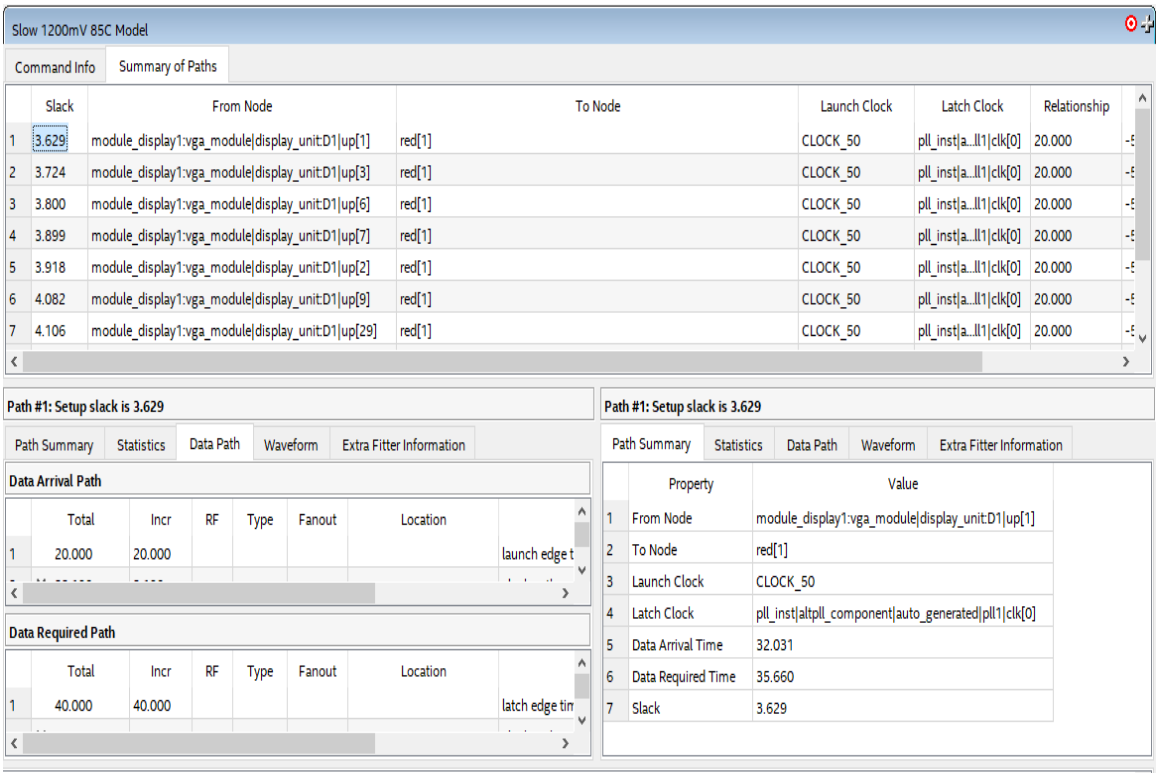


Figure 25: Setup summary (Detailed)

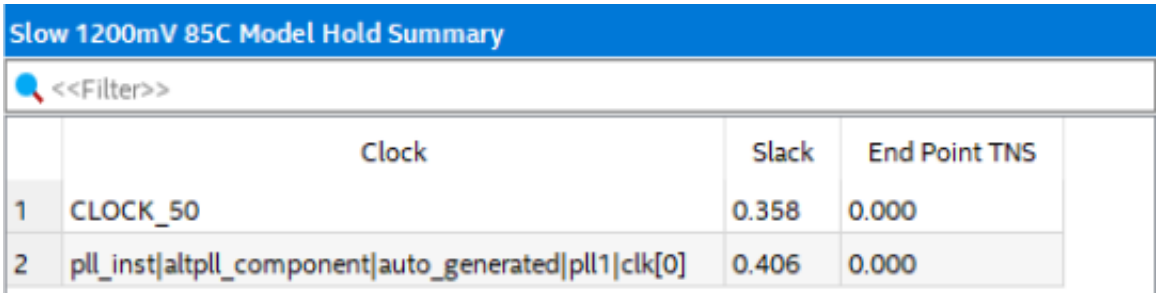


Figure 26: Hold summary

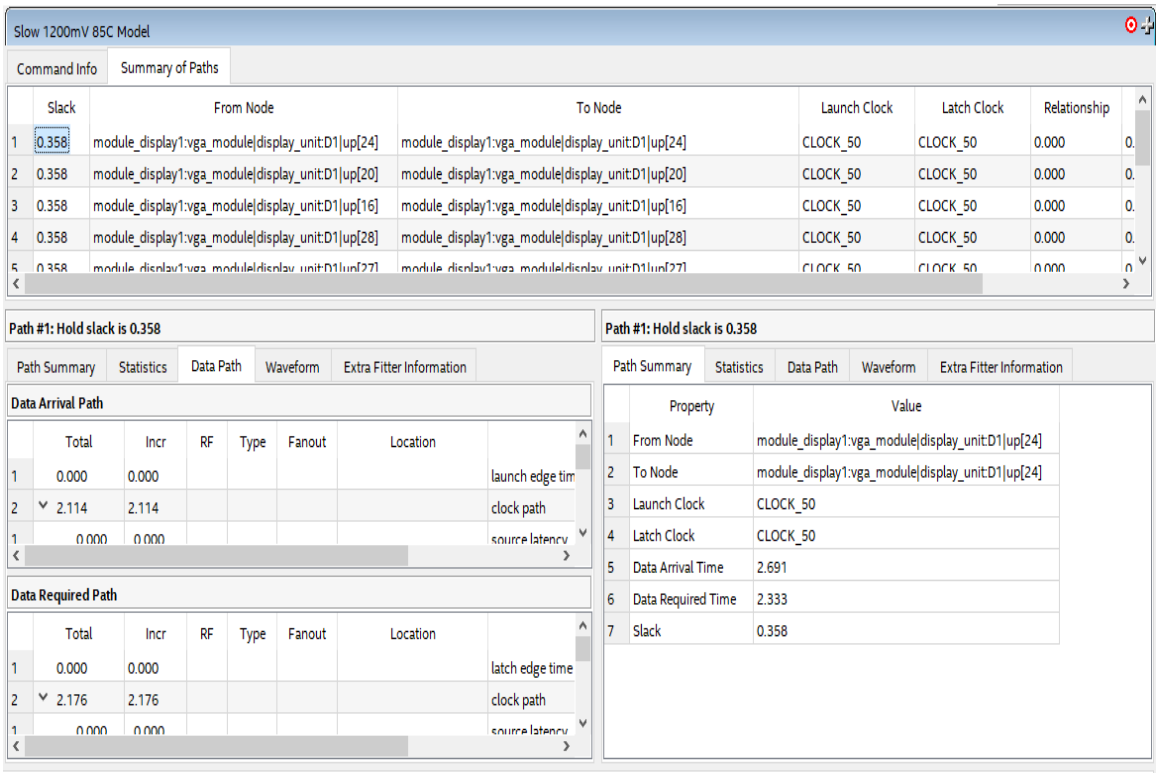


Figure 27: Hold summary (detailed)

3.4 Results from live testing

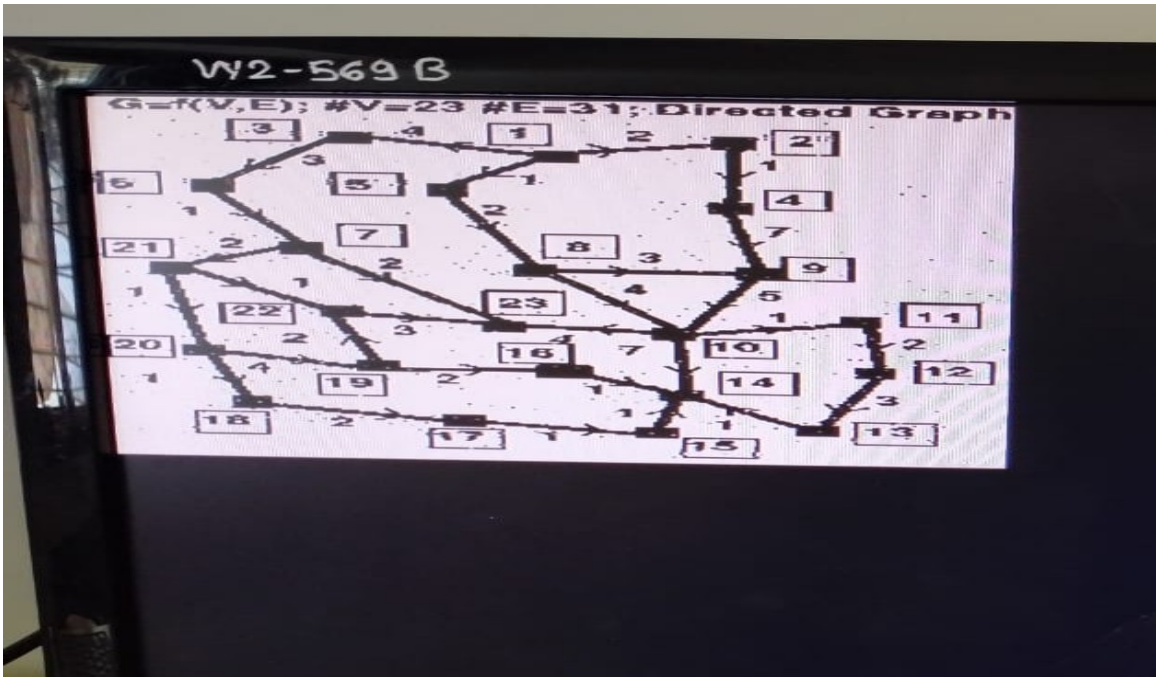


Figure 28: Graph displayed before setting source and destination vertex

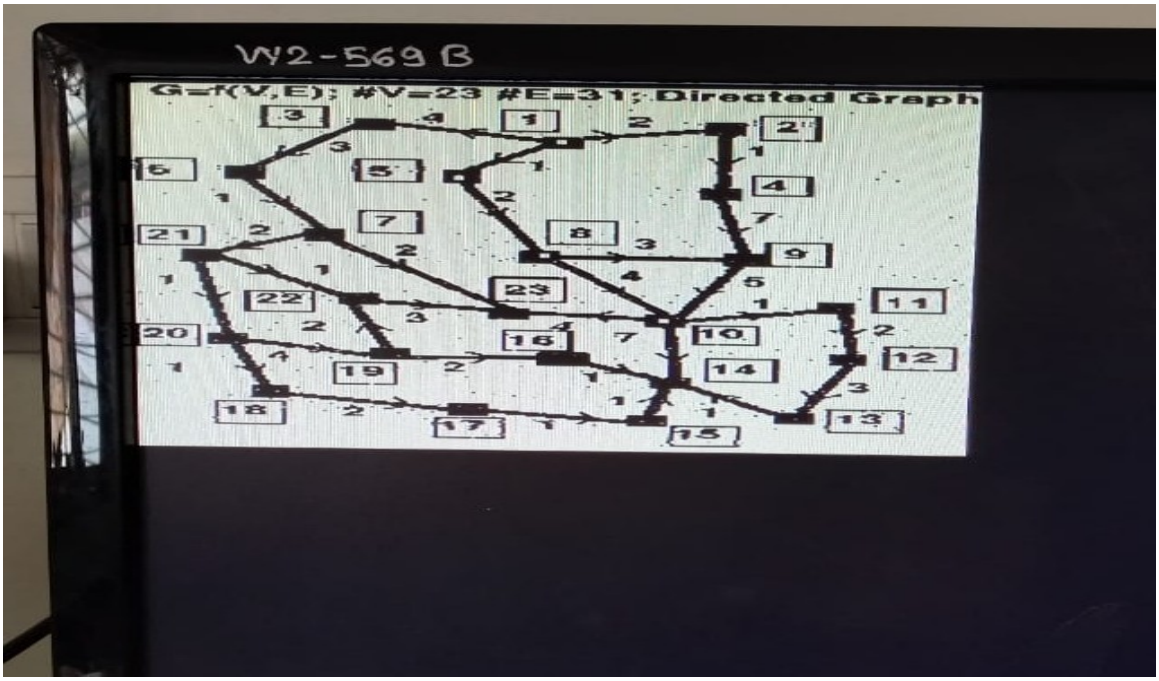


Figure 29: Graph showing shortest path from node 1 to node 10

References

[1] S. Zhou et.al, "Accelerating Large-Scale Single-Source Shortest Path on FPGA", 2015 IEEE IPDPSW.

[2] Bellman Ford Shortest Path Solver [Online], Accessed 08 May.
https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford/index_en.html

[3] VGA Controller Tutorial [Online]
https://www.youtube.com/watch?v=wzhDRIX2Ors