EE705 - vlsi design lab
Course Project

Bellman – Ford Algorithm

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 .

Implementation

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.

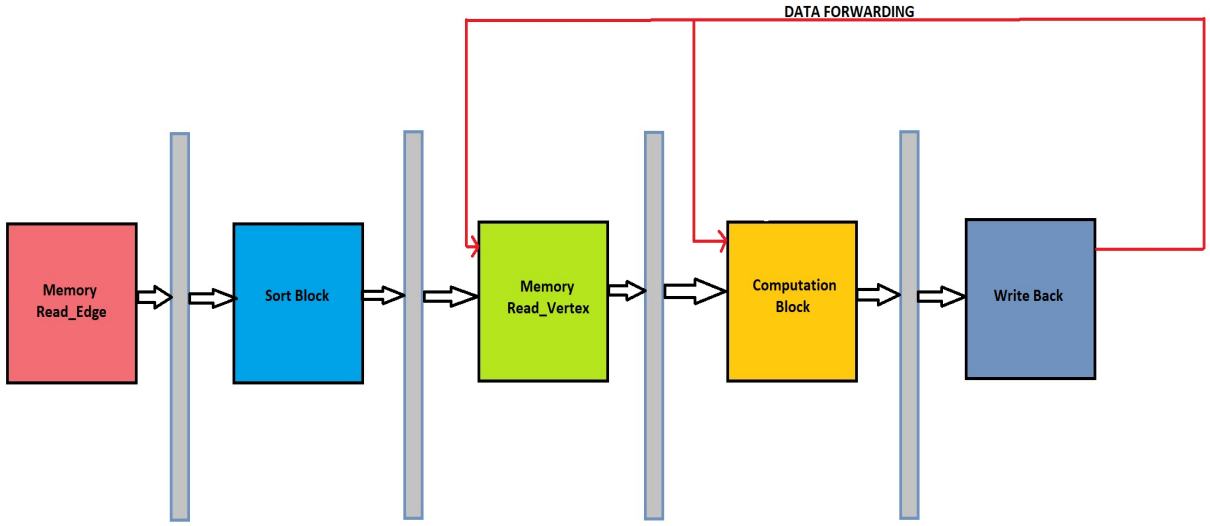

Figure 2: Implementation of Pipelined Bellman – Ford Algorithm.

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.

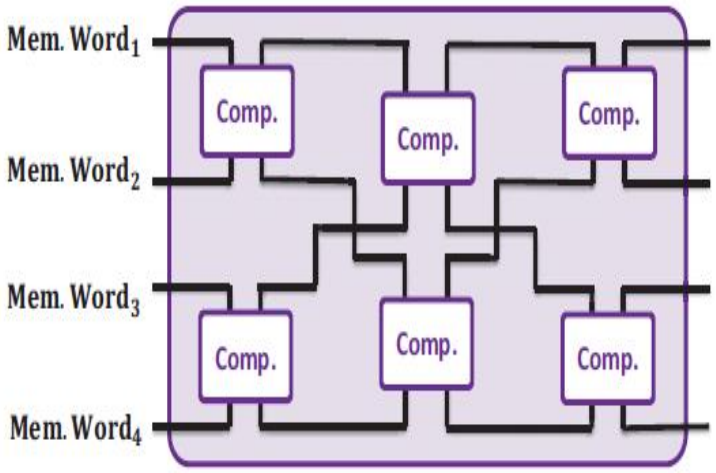

Figure 3: Sorting Block. (Source : [1])

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.

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.

Stage IV - Computation Block

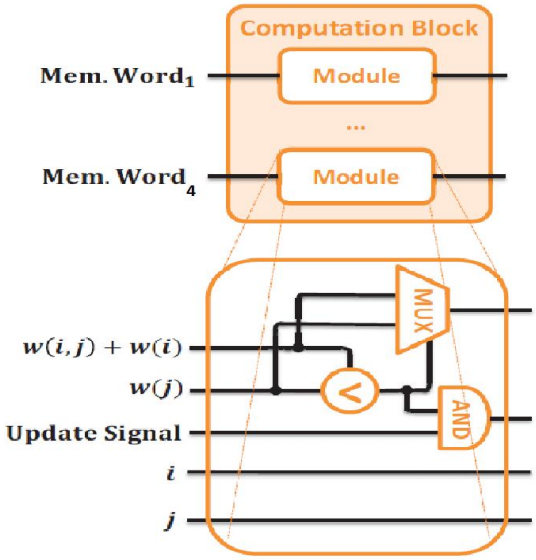

Figure 4: Computation Block. (Source : [1])

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.

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.

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.

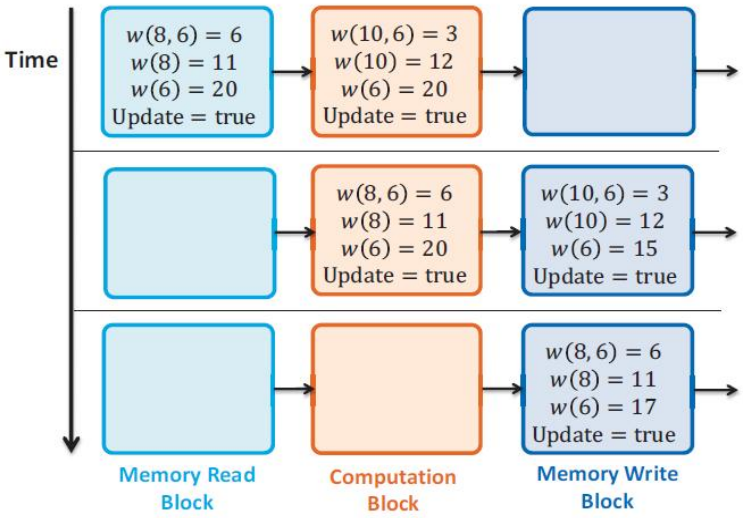

Figure 5: Example of Data Hazard. (Source : [1])

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].

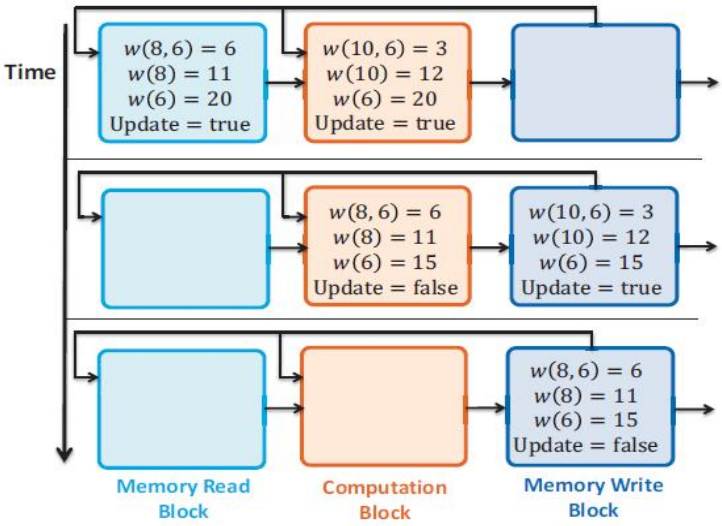

Figure 6: Implementation of Data Forwarding. (Source : [1])

Detailed Pipelined Architecture

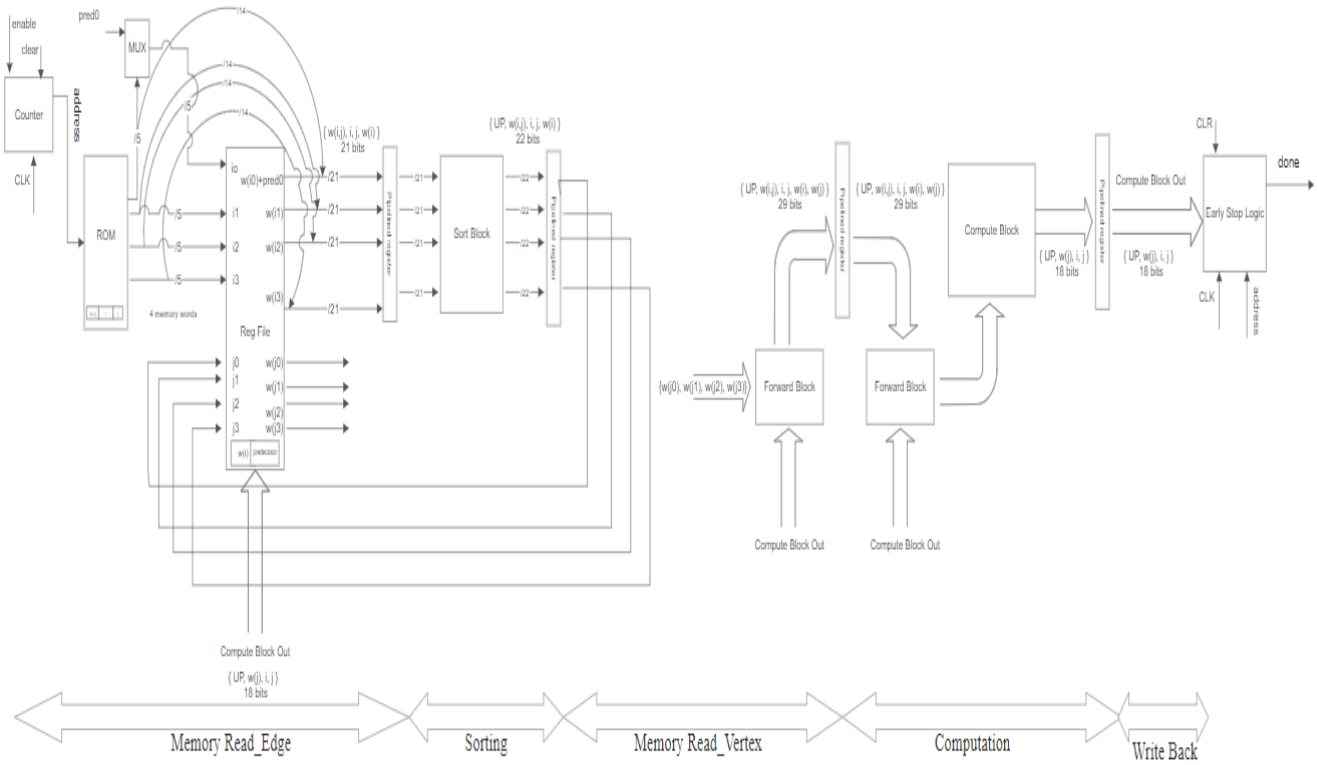

Figure 7: Detailed pipelined architecture

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)

Sample Graph and Expected Results

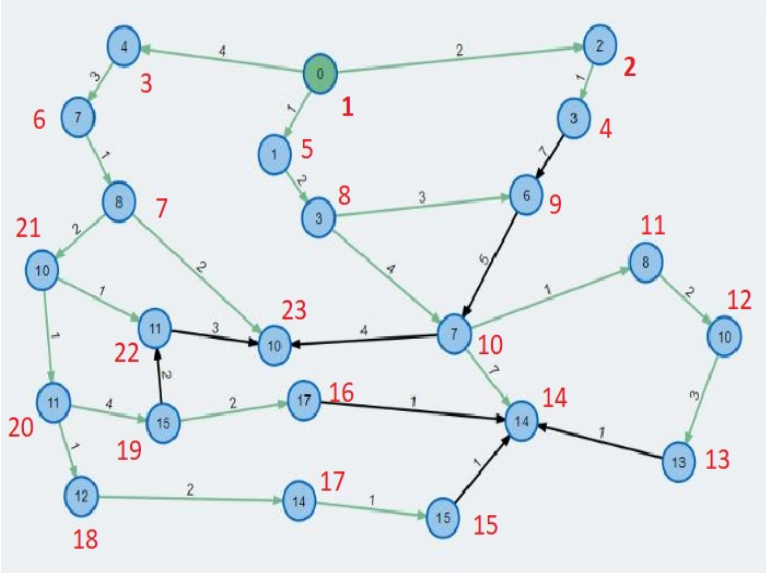

Figure 8: Sample graph with bellman ford algorithm solution (Source : [2])

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.

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.

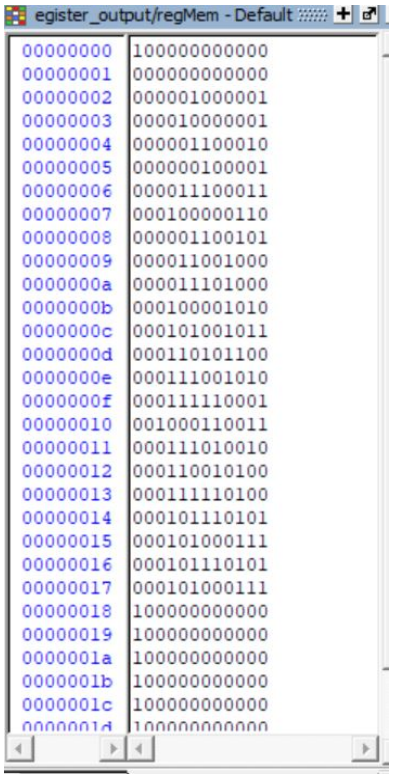

Figure 9: Memory List state of Reg File upon completion of Bellman Ford algorithm

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	000100000100	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.

Timing Analysis

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

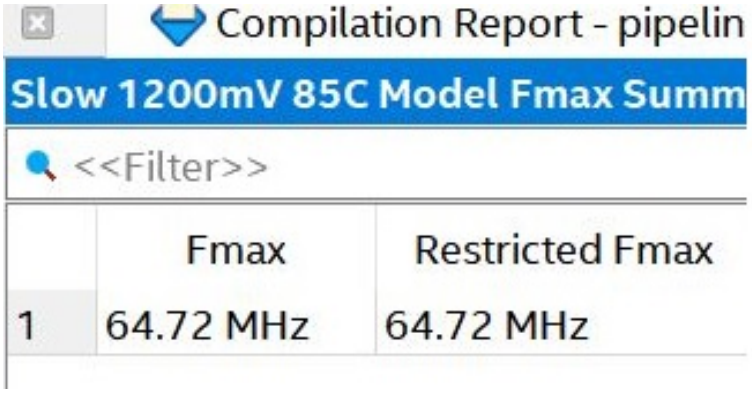

Figure 10: fmax summary

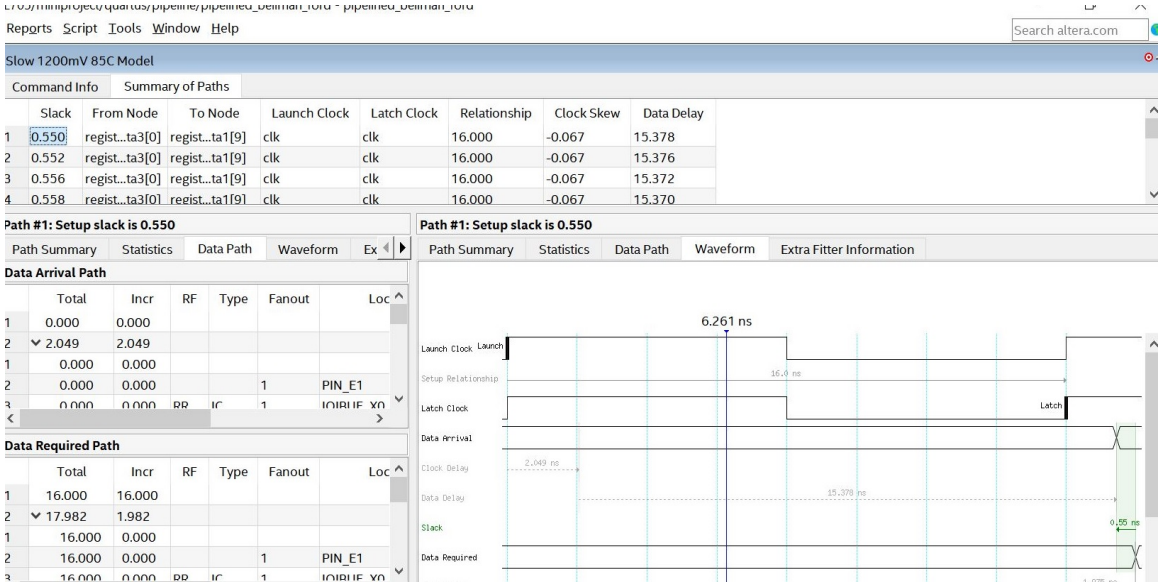

Figure 11: Setup Time Summary

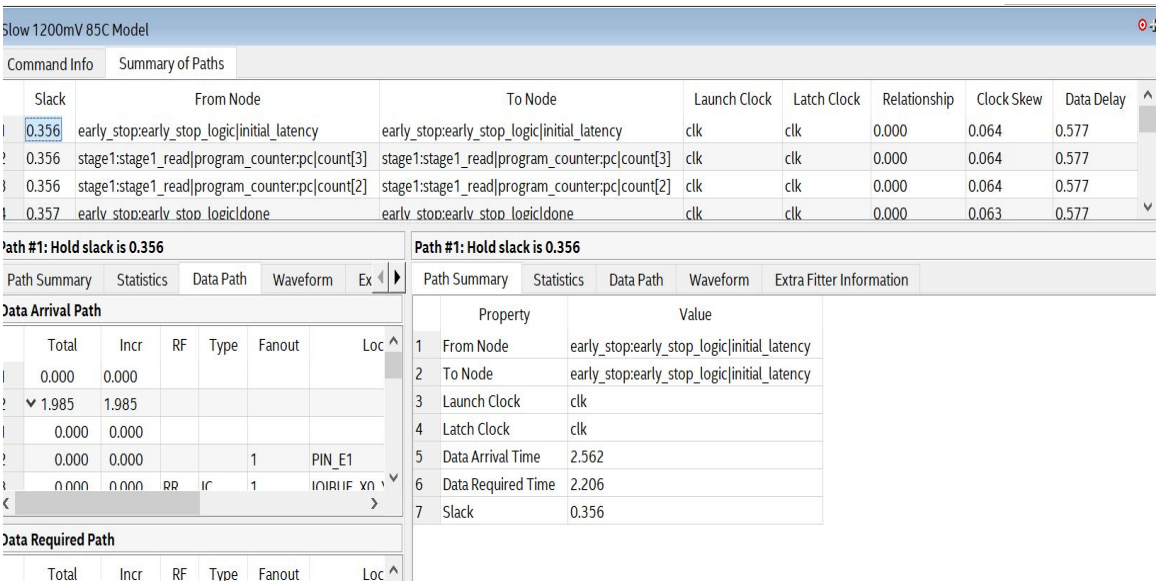

Figure 12: Hold Time Summary