5-9-2015

FPGA Based Binary Heap Implementation: With an Application to Web Based Anomaly Prioritization

Md Monjur Alam

FPGA BASED BINARY HEAP IMPLEMENTATION: WITH AN APPLICATION TO WEB BASED ANOMALY PRIORITIZATION

by

MD MONJUR ALAM

Under the Direction of Sushil K. Prasad, PhD

ABSTRACT

This thesis is devoted to the investigation of prioritization mechanism for web based anomaly detection. We propose a hardware realization of parallel binary heap as an application of web based anomaly prioritization. The heap is implemented in pipelined fashion in FPGA platform. The propose design takes $O(1)$ time for all operations by ensuring minimum waiting time between two consecutive operations. We present the various design issues and hardware complexity. We explicitly analyze the design trade-offs of the proposed priority queue implementations.

INDEX WORDS: Web Anomaly, FPGA, Priority Queue, Verilog

FPGA BASED BINARY HEAP IMPLEMENTATION: WITH AN APPLICATION TO WEB BASED ANOMALY PRIORITIZATION

by

MD MONJUR ALAM

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Sciences

Georgia State University

2015

FPGA BASED BINARY HEAP IMPLEMENTATION: WITH AN APPLICATION TO WEB BASED ANOMALY PRIORITIZATION

by

MD MONJUR ALAM

Committee Chair: Sushil K. Prasad

Committee: Xioajun Cao

Yanqing Zhang

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

May 2015

DEDICATION

This dissertation is dedicated to my mother, my wife and my son.

ACKNOWLEDGEMENTS

This dissertation work would not have been possible without the support of many people. I want to express my gratitude to my advisor Dr Sushil K. Prasad, for providing me an opportunity to work on this thesis. He has been guiding me through all the obstacles encountered in my research work and has been a constant source of motivation.

I must extend my thanks to all the committee members of this thesis, Dr. Xiaojun Cao and Dr. Yanqing Zhang, for there valuable suggestions to help in shaping this thesis.

There is a substantial contribution made by my wife Tazneem Alam to help me to finish this work. Apart from helping figure drawing, she has been the constant source of motivation in my ups and downs carrier. I must extend my thanks to her for providing healthy and tasty food through out my MS tenure.

I should not ignore the help of one innocence, my three year angel, Afnan Alam. While I am fatigue with work pressure, frustrated with the outcomes of research works; playing and giving accompany to this little baby boy alleviate my mental pain and these come to me as a tonic for energy and peace.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

- GSU - Georgia State University
- CS - Computer Science
- FPGA - Field Programmable Gate Array
- MS - Master of Science

PART 1

INTRODUCTION

Anomaly detection refers to the problem of finding patterns in data that do not conform to a well defined notion of normal behavior. We often refer these nonconforming patterns as anomalies or outliers [26]. Network based anomaly detection deals with score calculation and prepares a ranking for all packets based on that score. Due to high network congestion, it is incumbent to provide an efficient interface that can handle prioritization of packets based on the score assigned. As software based application inherently provides slower interface, the hardware based prioritization interface is necessary. Based on the priority, the interface will take some decisions (either pass or drop). For a high speed traffic, it is required to process these tasks in parallel.

Implementation of parallel priority queue will solve this requirement. A priority queue (PQ) is a data structure in which each element has a priority and a dequeue operation removes and returns the highest priority element in the queue. PQs are the most basic component for scheduling, mostly used in routers, event driven simulators [17], etc. There are several hardware based PQs implementations that are usually implemented by either ASIC chips [8,9,15] or FPGA [17-19]. But, all of them suffer some limitations and not applied to all applications.

1.1 Motivation of the Work

In the literature, several hardware-based priority queue architectures have been proposed [14,15]. All of these schemes have one or more shortcomings. The *Systolic Arrays* and *Shift Registers* based approaches [14,15], for example, are not scalable and require much hardware, more specifically, it require $O(n)$ comparators for n nodes. FPGA based pipelined heap is presented by Ioannou *et. al* [17]. This architecture is very much scalable and can run for 64K nodes without compromising performance. The major drawback of this design is that it takes at least 3 clock cycles to complete a single stage. More over, it never address the *hole* generated by parallel *delete* operation followed by an *insertion*. The calendar queues implemented by [8] can only accommodate a small fixed set of priority values since a large priority set would require extensive hard-ware support.

1.2 Objective and Design Issues

The objective of this work is to find a suitable design of parallel priority queue on FPGA platform to provide an efficient interface for the anomaly detector engine to handle packets prioritization very fast. We will store data based on its priority and this will be possible by incorporating parallel *addition* operation in binary heap. To access the highest priority data, we need to implement *delete* operation from the binary heap. Let us implement minimum (min) binary heap where root contains the maximum (max) priority element. As our intention is to provide efficient interface, the following design issues we should address while implementing it.

- To minimize waiting time for two consecutive operations.
- To minimize *hole* created by *deletion*.
- The design should be highly scalable and optimized.

1.3 Main Contribution

We have implemented a software based anomaly detection mechanism where a score is assigned to each packet. We apply Markov based model for score calculation. A FPGA based parallel binary heap is implemented for score prioritization. We present the various design issues and hardware complexity. The pipeline architecture ensures no waiting time for any operation except the *deletion* one which has to wait for a single cycle. Each of *insert* and *delete* operation takes $O(1)$ time. We also evaluate the design trade-offs of the proposed priority queue implementations. Our design takes care the *hole* created by *delete* operation. We minimize the *hole* at the time of *insertion*.

1.4 Organization of the Thesis

A Summary of the contents of the chapters to follow is given below:

Part 2: Contains an overview and the art of literature related to the work.

Part 3: Our proposed design including implementation result is presented here. We also describe different design trade-off in this part.

Part 4 : This part contains some concluding remarks and identifies some directions for future research.

PART 2

PRELIMINARY AND RELATED WORK

2.1 Web Based Anomaly

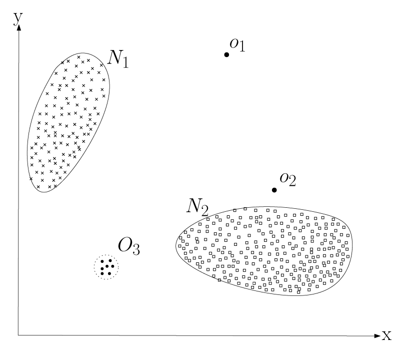

Figure (2.1) Illustrating anomalies in a two-dimensional data set [26].

Anomaly detection refers to the problem of finding patterns in data that do not conform to a well defined notion of normal behavior. We often refer these nonconforming patterns as anomalies or outliers. Fig. 2.1 depicts anomalies in a simple two-dimensional data set [26]. There are two normal regions N_1 and N_2 for the data since most observations reside in these regions. The points o_1 and o_2 and all the points in region O_3 are considered as anomalies as theses points are sufficiently far away from the two normal regions. We can consider net work packet in each region as data set. Each packet belongs to a particular set based on its score calculation.

2.1.1 Score Calculation

Among several methods, Markov model is one to calculate score for each packets [40]. The Markov model (MM) can be viewed as a probabilistic finite state automaton (PFSA) which generates sequences of symbols. The output of the Markov model consists of all paths from its start state to its terminal state. A probability value can be assigned to each

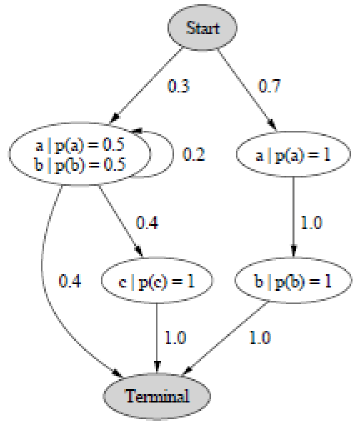

Figure (2.2) Markov Model Example [40].

output transition and the resultant score is calculated as the summation of all transition probability. For example, consider the non-deterministic finite automata (NFA) in Figure 2.2. To calculate the probability of the word 'ab', one has to sum the probabilities of the two possible paths (one that follows the left arrow and one that follows the right one). The start state emits no symbol and has a probability of 1. The result is

$$\begin{aligned} p(w) &= (1.0 * 0.3 * 0.5 * 0.2 * 0.5 * 0.4) + (1.0 * 0.7 * 1.0 * 1.0 * 1.0 * 1.0) \\ &= 0.706 \end{aligned} \tag{2.1}$$

2.1.2 Prioritization

Software based score prioritization of network packets are presented by Kruegel *et. al* [24]; where the packets with maximum score gets high priority to be processed next. Each time, score is calculated *on the fly* and it is compared with other set of precalculated scores. Effectively, there is a processing delay to come up with a decision. Moreover, processing parallel packet is not possible here, as the on the fly calculation here is highly serialized process.

2.2 Priority Queue

A priority queue is an abstract data structure that maintains a collection of elements with the following set of operations by a minimum priority queue Q:

- **Insert:** A number n_i is inserted into the set of candidate number N in Q, provided that the new list maintain the priority queue.

- **Delete:** Find out the minimum number in Q and delete that number from Q. Again, after deletion the property of priority queue should be kept unchanged.

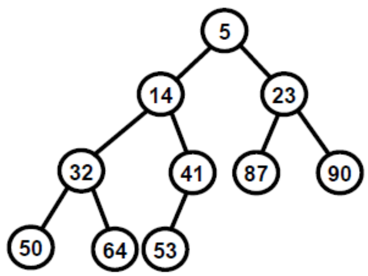

Figure (2.3) Binary Min Heap.

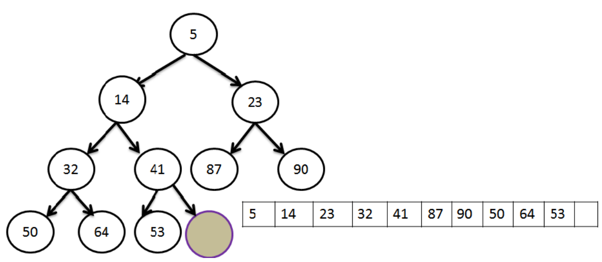

Figure (2.4) Array Representation of Binary Min Heap.

2.2.1 Priority Queue Implementation

Priority queue can be implemented by using binary heap data structure.

Definition 2.2.1 *A min-heap is a binary tree H such that (i) the data contained in each node is less than (or equal to) the data in that nodes children and (ii) the binary tree is complete.*

Figure 2.3 shows the binary min heap (H). The root of H is $H[1]$, and given the index i of any node in H, the indices of its parent and children can be determined in the following way:

$$\begin{aligned} parent[i] &= \lfloor i/2 \rfloor \\ leftChild[i] &= 2i \\ rightChild[i] &= 2i+1 \end{aligned}$$

Figure 2.4 illustrates the array representation of binary heap. The insertion algorithm on the binary min heap H is as follow:

- Place the new element in the next available position (say i) in the H.
- Compare the new element $H[i]$ with its parent $H\lfloor i/2 \rfloor$. If $H[i] < H\lfloor i/2 \rfloor$, then swap it with its parent.
- Continue this process until either (i) the new elements parent is smaller than or equal to the new element, or (ii) the new element reaches the root ($H[1]$).

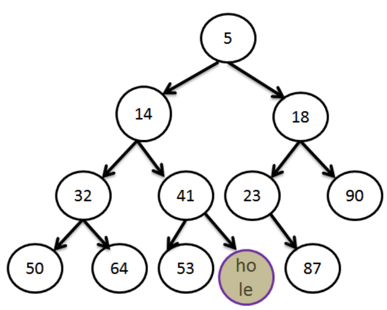

Figure (2.5) New heap structure after insertion 18.

Figure 2.5 shows the new heap structure after insertion of 18 at the heap presented in Figure 2.3.

The deletion algorithm is as follow:

- Return the root $H[1]$ element.

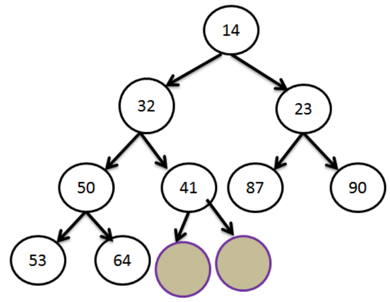

Figure (2.6) New heap structure after single deletion operation from the original heap shown at figure 2.3.

- Replace the root $H[1]$ by the last element at the last level (say $H[i]$).
- Compare root with its children and replace the root by its min child.
- Continue this replacement for each level by comparing $H[i]$ with $H[2i]$ and $H[2i+1]$, un till the parent become less than its children or it reaches to the leaf node.

Figure 2.6 depicts the heap structure of single deletion operation from the original heap shown at Figure 2.3. We can see that 5 was the root element at Figure 2.3. The updated Figure 2.6 depicts that the 5 is no anymore after the *deletion*. Moreover, heap is re-structured according to the *deletion* algorithm presented above.

2.3 Related Work

Many web anomaly detection techniques have been proposed which applied a set of training data to define a model of normal behaviour. It labelled any data as abnormal that is not included in this model [25,27,29,31,35-37]. Several variants of the basic technique have been proposed for network intrusion detection, and for anomaly detection in text data [23,34,39]. These approaches assume independence between the different attributes. Some approaches have been introduced that assume the conditional dependencies between the different attributes applying more complex Bayesian networks [28,33,38]. Rule-based anomaly detection techniques distinguish normal behavior of data instances from anomalies by learning rules. A test instance is termed as anomaly if it is not covered by any such rule. There are two steps for rule-based anomaly detection approach. First, rules are learned from the training data using a rule learning algorithm. A confidence value is associated with every rule. The second step is to search the rule that best captures the test data instance. The anomaly score of the test instance is calculated as the inverse of the confidence associated with the best rule. For example, a typical rule-based system is an expert system where the rules are generated by humans [26,30,32].

All of the approaches mentioned suffer from two basic problems:

1. There is no efficient implementation to deal with huge network congestion.

2. prioritization of network traffic is not maintained.

2.3.1 Anomaly Detection by Using Hardware

To resolve the first class of difficulty several authors [20,22] come up with hardware based solution. The intention is to provide very fast interface to process network data. To achieve this goal, Das *et. al.* [20,21] comes up with hardware based solution for anomaly detection. The work comprises of a new Feature Extraction Module (FEM) which summarizes the network behavior. It also incorporates an anomaly detection mechanism using Principal Component Analysis (PCA) as the outlier detection method. The authors of [22] propose a mechanism of feature extraction. The method is implemented on FPGA and it is suitable for large network with high data flow.

2.3.2 Parallel Priority Queue

Several authors have theoretically proved that parallel heap is an efficient data structure to implement priority queue. Prasad *et. al.* [1,4] theoretically illustrate this data structure to show $O(p)$ operations are required with $O(\log n)$ time for $p \leq n$, where n is the number of nodes and p is the number of processor used. The idea is designed for EREW PRAM shared memory model of computation. The many core architecture by [3] in GPGPU platform provides multi-fold speed up. Another theoretical approach [5] ensures $O(\log n)$ time processing time for n number of nodes. The implementation of this algorithm is expensive for multi-core architectures [6].

Hardware Based Priority Queue There have been several hardware based parallel priority queue implementations described in the art of literature [8-15]. Pipelined based ASIC implementations can reach $O(1)$ execution time [11,12]. Due to several limitations like cost and size, most of the ASIC implementations does not support a large number of nodes to be processed. These implementation are also limited to high scalability. In [13], the author claims the pipelined heap presented be the most efficient one. However, this implementation incurs high hardware cost. The design is not flexible, more specifically, it is designed with a fixed heap size. The *Systolic Arrays* and the *Shift Registers* [14,15] based hardware implementations are well known in the literature. The common drawback of these two implementation is using a large number of comparator ($O(n)$). The responsibility of comparators used here to compare nodes in different level with $O(1)$ step complexity. For the shift register [15] based implementations, when new data comes for processing, it is broadcasted to all levels. It requires a global communicator hardware which can connect with all level. The implementation based on *Systolic Arrays* [14] needs a bigger storage buffer to hold pre-processed data. These approaches are not scalable and require much hardware, more specifically, it require $O(n)$ comparators for n nodes. To overcome the hardware complexity, a recursive processor is implemented by [16]; where a drastic hardware is reduced by compromising execution timing cost. Bhagwan and Lin [9] designed a physical heap such a way that commands can be pipeline between different levels of heap. The authors in the paper [8] give some pragmatic solution of so called *fanout* problem mentioned in [10]. The design presented in [41] is very efficient in terms of hardware complexity. But, as the design is implemented by using hardware-software co-design, it is very slow in execution ($O(\log n)$).

For the FPGA based priority queue implementation, Kuacharoen *et. al* [19] implemented the logic presented in [10] by incorporating some extra features to ensure the design to be acted as a task scheduler in real time. The major limitation of this paper is that it deals with very small number of nodes. A hybrid priority queue is implemented by [18] and it ensures high scalability and high throughput. FPGA based pipelined heap is presented by Ioannou *et. al* [17]. This architecture is very much scalable and can run for 64K nodes without compromising performance. The major drawback of this design is that it takes at least 3 clock cycles to complete a single stage. More over, it never address the *hole* generated by parallel *delete* operation followed by an *insertion*.

PART 3

FPGA BASED PARALLEL HEAP

Like an array representation, heap can be represented by hardware register or FPGA latch. Each level of the heap can be virtually represented by each latch. The size of the latch at each level can be represented as $2^{\beta-1}$, where β is the level assuming that root is the level 1. Figure 3.1 shows the different latches do represent the different levels. Here, root node can be stored by L_1, the next level with two elements can be stored in L_2 and the last level with 3 elements can be stored in L_4, although the last level can have max 8 elements.

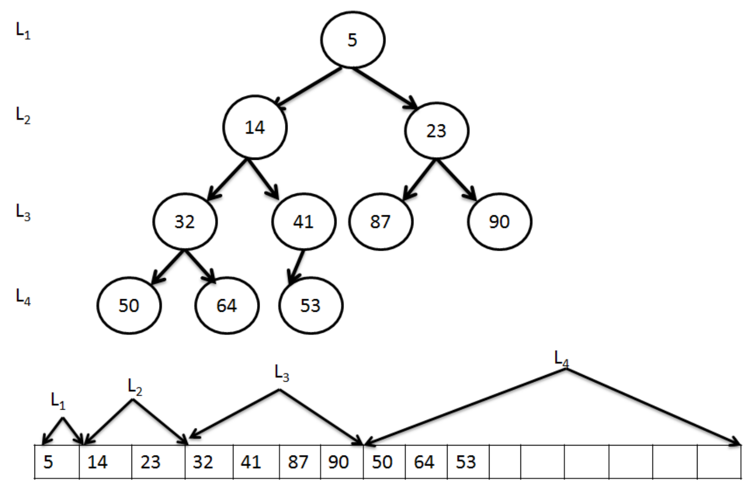

Figure (3.1) Storage in FPGA of deferent nodes in binary heap

3.1 Insert Operation

We have already discuss the insert operation which is intimated from the last available node of the heap. This bottom up approach restrict the other operations like delete, replace, etc. to perform in parallel. As deletion means the least element to be deleted and the least element always resides at root in case of min heap; deletion operation should wait till the root is updated by the insert operation. If we insert element 3 in the heap mentioned at

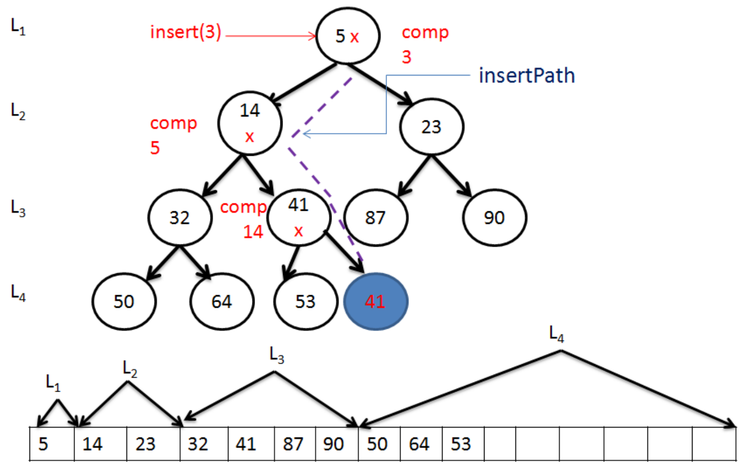

Figure (3.2) Insertion path

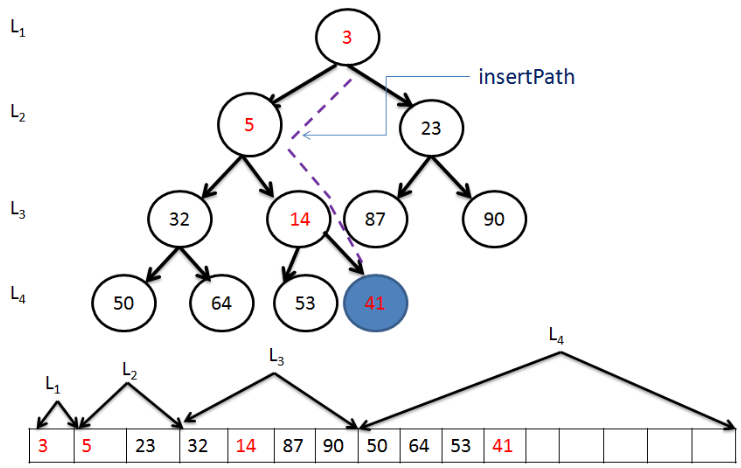

Figure (3.3) Contain of latch (L) after insertion completed

Figure 3.1, followed by delete one element from heap then what will happen? Let us assume nodes at each level get updated by a single clock cycles. That means, in worst case, total 4 clock cycles are required to complete the insert operation in this situation. So, delete operation either has to wait for 4 clock cycles or it will wrongly delete the root, which is 5. So, it is incumbent to insert from root and go down. But, we need to know the path for the new inserted element, otherwise the tree will not be complete binary tree. We have adopted a nice algorithm presented by Vipin *et. al* [7] in our design. The algorithm is as follow:

- Let k is the last available node at where new element to be inserted. Let j be the first node of the last level. Then binary representation of $k - j$ will give you the path.

- Let $k - j = B$, which binary representation is $b_{\beta-1}b_{\beta-2} \cdots b_2 b_1$. Starting from root, scan each bit of B starting from $b_{\beta-1}$;
  - if $b_i == 0$ ($i \in \{\beta - 1, \beta - 2, \cdots, 2, 1\}$), then go to left
  - else go right

The Figure 3.2 shows the insertion path for *new* element to be inserted. For the new element insertion, node at 11 should be filled up. The first node of the last level is at index 8. So, 11-8 = 3, which can be represented as 011. So, starting from root, the path should be $root \rightarrow left \rightarrow right \rightarrow right$ and this can be demonstrated by the Figure 3.2. After the insertion completion, the contain of the nodes along with the value of latch is presented by the Figure 3.3.

3.2 Delete Operation

There is one conventional approach to delete element from heap. As root resides the min element, deletion always happen from root and the last element is replaced to root. There are two difficulties here:

1. For sequential operation, it works perfectly file. For, parallel execution of insert/del, *hole* can be created here. The situation happen after any *insert* followed by *delete* operation.

   From the Figure 3.4 we can illustrate this scenario clearly. Let at t_1, the operation insert with element 100 is encountered and it is denoted by $insert(100)$. Obviously, the element will be inserted at the last node of last level which is 12. Let, after one clock cycle of *insert*, *delete* is encountered (say at t_2). At, that time, *insert* was modifying at L_2. So, due to *delete*, hole will be created at node 10th as shown in Figure 3.4. Eventually, when $insert(100)$ will finish, the element 100 will occupy at the position of $H[12]$, but, $H[11]$ will become empty. This situation is illustrated by Figure 3.5.

   Let us assume that *insert* instruction comes at time t_i and *delete* instruction comes at t_j, where $i, j = 1, 2, 3, \cdots$ and $j > i$. Let, operation of either *insert* or *delete* takes

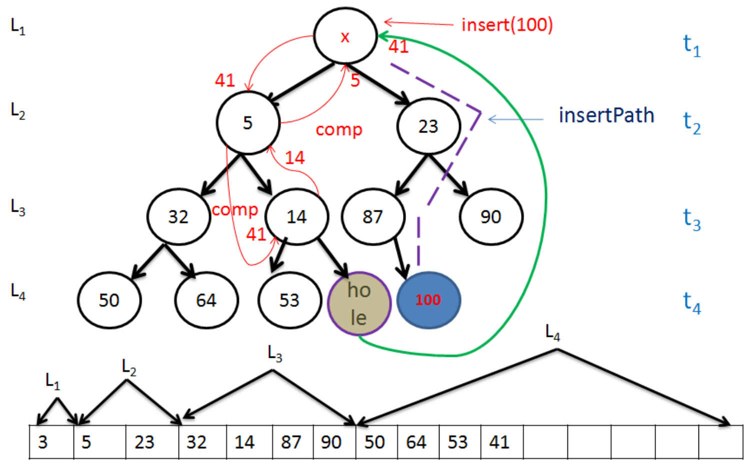

Figure (3.4) Hole is the resultant for parallel operation of insert-delete

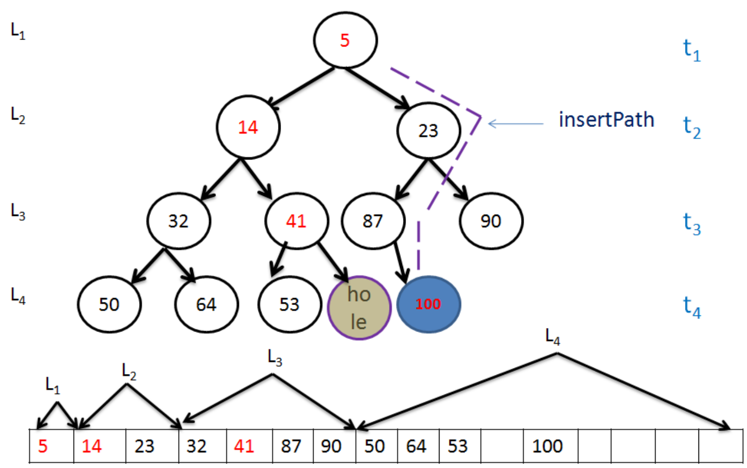

Figure (3.5) Contain of latch (L) after parallel operation of insert-delete

one clock cycle at any level to complete tasks at that level. It is obvious that, only single node gets modified (if any) for all levels. In general, for any $insert - delete$ combination, *hole* will be created if $(t_j - t_i) < \beta$, where β is the depth of heap.

2. While you replace root by last element of heap, it requires extra clock cycle. Moreover, we need to compare three elements, root and its two children or any node and its children. For hardware perspective, it is cost efficient to compare two elements rather than to compare three elements. More over, it incurs the path delay longer.

So, we should intentionally avoid the root replacement by last element. Let us delete root first and keep it as it is. Fill the root with its least child and follow the algorithm. In this case, we can save one cycle and hardware cost, more specifically, can minimize the path delay. Now, our aim is to minimize *hole* by adding logic.

3.3 Insert-Deletion Logic Implementation

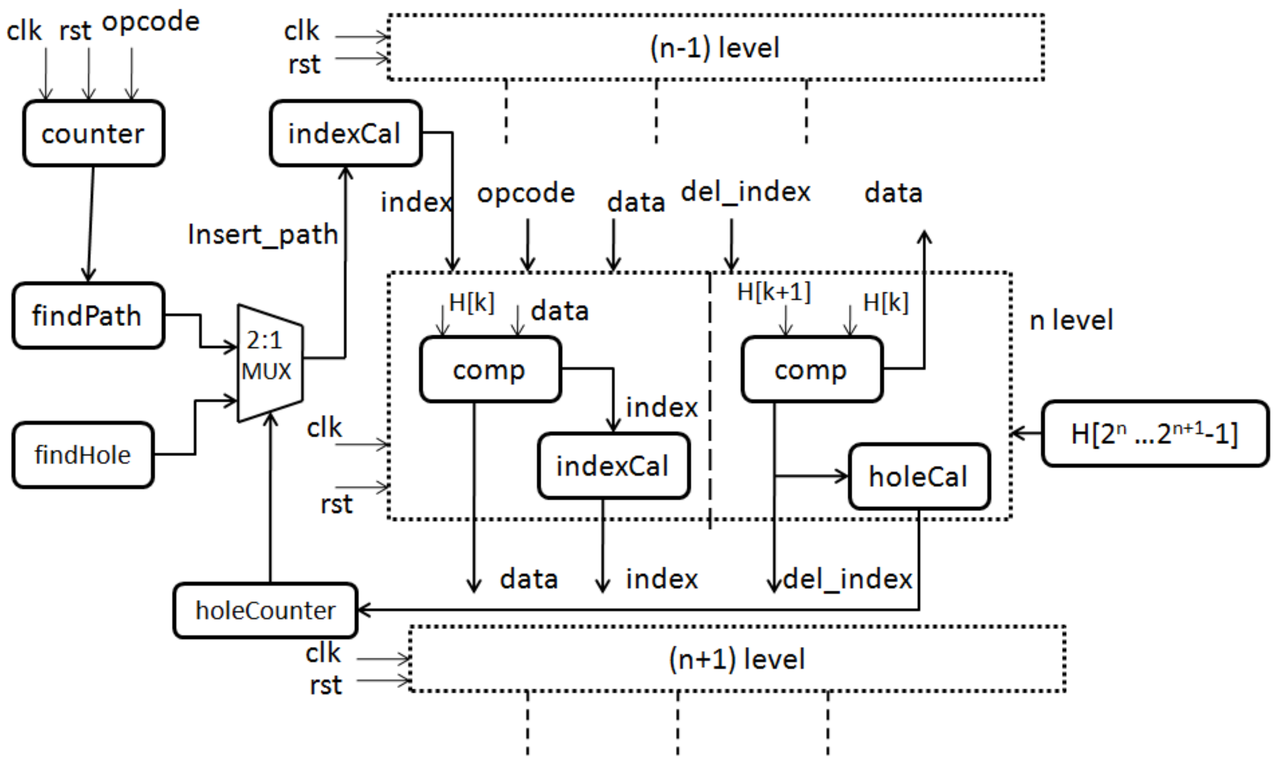

Figure (3.6) Top Level Architecture of insert-delete

Figure 3.6 illustrates the top level architecture of *insertion-delete* operation. The *counter* is used to maintain the total number of element present in the heap. It is incremented by one for *insert* operation and decremented by one for *deletion operation*. The *indexCal* block is used to find the insertion path. We have modified the existing path finding algorithm by [7]. We first consider the *holeReg* to obtain insertion path. The *holeReg* contains the *holes* created at *deletion* operation. We maintain a *holeCounter* to identify a valid *hole*. Based on the *index*, the heap node is accessed and the node is compared with the present data. Based on the comparison, either the node is updated by present data and the node is passed to the next level as present data, or the node become unchanged and the present data is passed to the next level.

Deletion : We maintain *del_index* to find the last deleted node. For example, initially, *del_index* becomes 1 which means root is deleted. The comparator finds the min element between $H[del\_index * 2]$ and $H[del\_index * 2 + 1]$ and that min gets replace to $H[del\_index]$. Now, *del_index* gets modified with the index of min element. Again the comparator finds the min of the ancestors of the new index and replace the node of new index with that of min one. Each time *holeCal* finds if there is a valid child for *del_index*. If there is no valid child, then *holeCounter* is incremented by 1 and *holeReg* is updated with *del_index*. By this way, we maintain *hole*.

Algorithm 1 Algorithm for $Insert - Delete(data, opcode)$

```
if (opcode == 1) then
    counter = counter +1;
    if (holeCounter > 0) then
        insert_path = findPath(counter, holeCounter)
    end if
    for (0 to number of level) do
        index = indexCal(insert_path)
        if (data < H[index] then
            H[index] = data
            data = H[index]
        else
            data = data
        end if
    end for
else
    Remove H[1]
    while (leftChild[del_index] ≠ NULL&rightChild[del_index] ≠ NULL) do
        if (leftChild[del_index] < rightChild[del_index]) then
            H[del_index] = leftChild[del_index]
            del_index = del_index * 2
        else
            H[del_index] = rightChild[del_index]
            del_index = del_index * 2 + 1
        end if
    end while
    hole_counter = hole_counter + 1
    hole_reg[hole_counter] = del_index
end if
```

The *insert-delete* parallel algorithm is presented at Algorithm 1. We use 2:1 multiplexer to select the path based on the value of *holeCount*. The logic for *findPath* is illustrated at Algorithm 2. The *indexCal* block is implemented based on the value of *findPath* and the logic is illustrated at Algorithm 5. To calculate the first node of last level is noting but the mathematical expression of $2^{\beta-1}$ where β is the level of heap. There is some difficulty to realize this expression in hardware. We express this logic by Algorithm 3.

Algorithm 2 Algorithm for $findPath(counter, holeCounter)$

if $(holeCounter > 0)$ **then**
  return $findHole(holeCounter)$
else
  $leaf\_node = find\_1st\_node\_last\_level(counter)$
  return $(counter - leaf\_node$
end if

Algorithm 3 Algorithm for $find\_1st\_node\_last\_level(counter)$

for $(i = 0$; $2^i < counter$; i = i+1) **do**
  $leaf\_node$ = i+1
end for
return $leaf\_node$

Algorithm 4 Algorithm for $findHole(hole\_counter)$

return $holeReg[holeCounter]$

We have used global clock(clk) and global reset(rst) signal for the each logic block except the combinational logic parts. The clk and rst signals are not mentioned at each figure due to place limitation. The function of *findHole* is basically an implementation of stack register and its return value is presented at Algorithm 4.

Algorithm 5 Algorithm for $indexCal(insert\_path)$

for ($i = 0$ **to** $insert\_path$ bits) **do**
  if (bit $==$ 0) **then**
    $index_i = 2 {*} index_(i-1)$
  else
    $index_i = 2 {*} index_(i-1) + 1$
  end if
end for

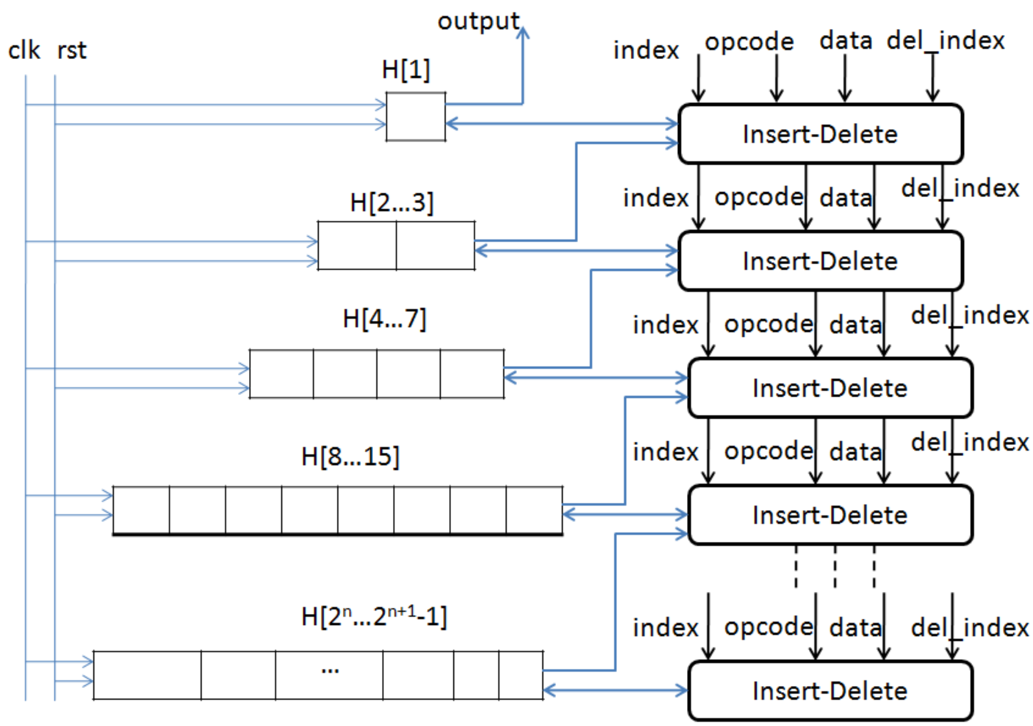

Figure (3.7) Pipeline Design Overview

3.4 Pipeline Design

To achieve high throughput we need to start one operation before completing the previous operation. So, many operation can be in progress in the tree. To achieve so, we consider our design to take a single clock cycle to perform each stage. For any stage, only one operation (*insert, delete*) can be execute at ant time t. That is why we need all operation should be started from the top (root) of the tree and proceed towards the bottom (leaf).

Figure 3.7 illustrates the basic pipeline architecture of our binary heap. Each level perform *insertion or deletion* based on the signal *opcode*. Each level takes three clock cycles to perform all operations. Each level sends *data* and *opcode* to the next level to perform. There is a global clock and global reset attached to each stage. All the level contains the same logic hardware except the first level.

3.4.1 Optimization Technique

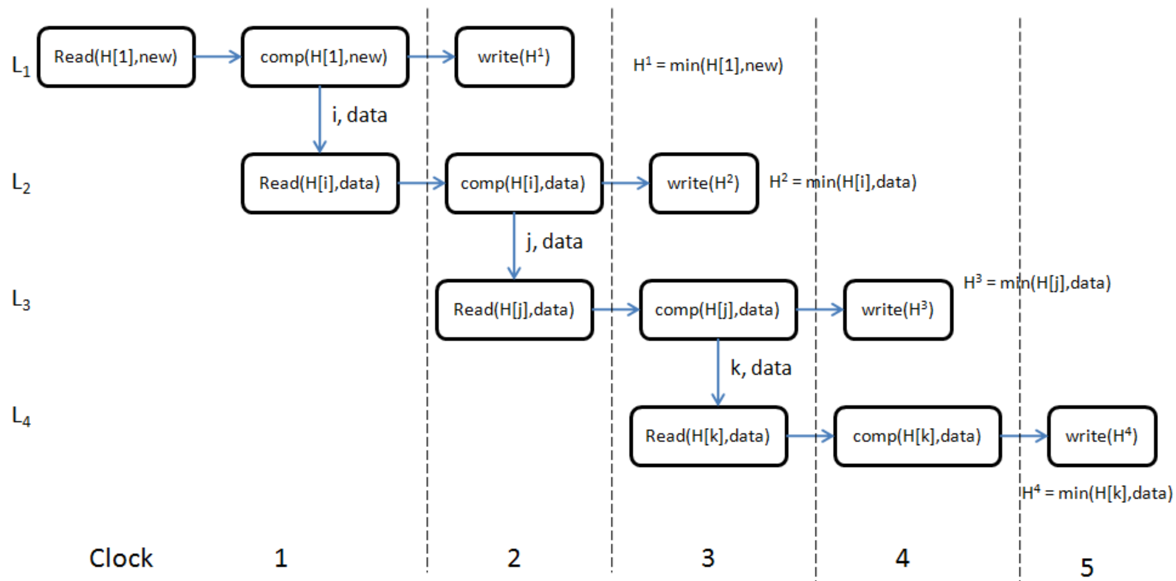

Figure (3.8) Parallel insert operation: illustrates operations at each level at each clock

We need to know the operations at each level at each clock cycle to provide more optimization. We make each individual operations like *Read, Write* and *compare (comp)* to complete in separate single clock cycle. Each level has to perform these three basic operations resulting three clock cycles in total. We pre-compute data for a level such a way that there are maximum overlap between consecutive two levels in case of *insertion*. For any level β, if *Read* operation executes at t time, then it executes *comp* operation at $t+1$. The *comp* generates the next *index* to be read by the next level. So, $\beta+1^{st}$ level perform *Read* at $t+1$ time. Now, β level performs write operation at $t+2$, while $\beta+1^{st}$ level finish *comp* and generates the index to be read by the $\beta+2^{nd}$ level. At time $t+3$, the $\beta+1^{st}$ level will perform *Write* operation while the $\beta+2^{nd}$ level will complete the *comp* operation and will make *index* available for the $\beta+3^{rd}$ level. By this ways, we find there are two operations overlaps between two consecutive levels in 3 cycles. Effectively, it results of writing at each clock cycle after initial latency of two clock cycle at the first level. The Figure 3.8 illustrates this situation. We can see that, while level L_2 perform *comp* at clock 2, then level L_3 performs the *Read*. The level L_2 completes *Write* at clock 3, while level L_3 completes the *comp* followed by *Write* at clock 4. We make *comp* operation by β and *Read* operation by $\beta+1$ at same clock cycle t by using the concept of different edge of clock. Level L_2, for example, performs *comp* at positive edge of clock 2 and level L_3 performs *Read* at negative edge of clock 2.

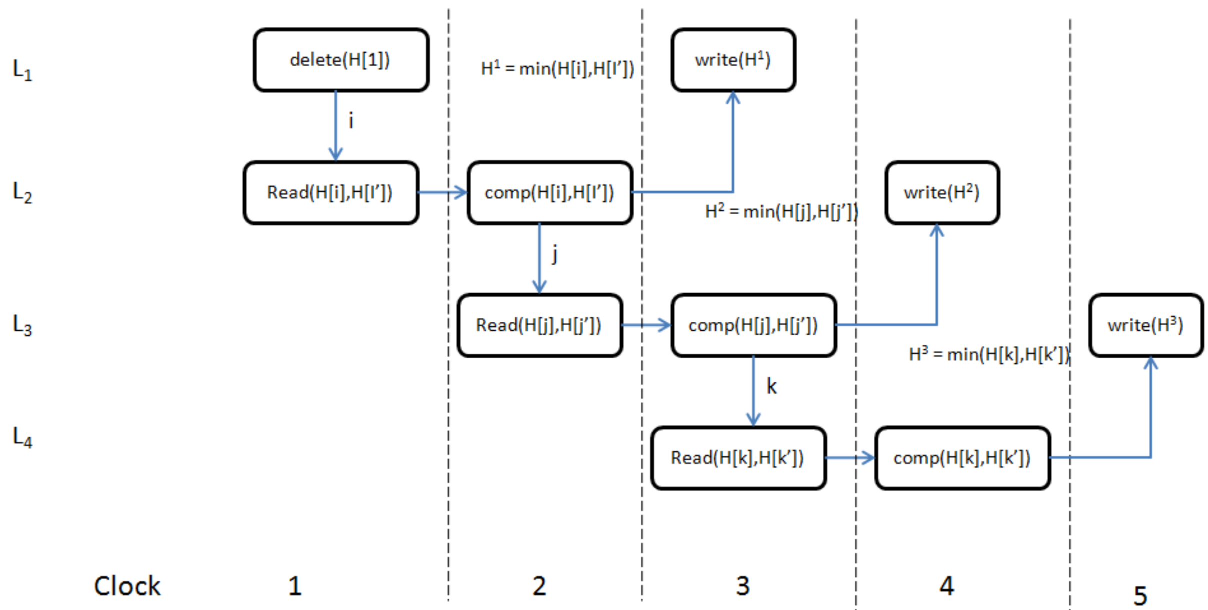

Figure (3.9) Parallel delete operation: illustrates operations at each level at each clock.

Hardware Sharing Unlike, the *insert*, the *delete* operation of any level waits for data from its next level. As the min element of a certain levels go up to the upper level, the data will be available to write after performing the *comp* operation of that level. In general, if *Read* operation executes at t time by level β, then it executes *comp* operation at $t+1$ (except root level). As the *comp* generates the next *index* to be read by the next level. So, $\beta+1^{st}$ level perform *Read* at $t+1$ time. But, the level β can not perform *Write* operation at $t+2$, because the data from $\beta+1^{st}$ level will be written at the level β; and the resultant of *comp*

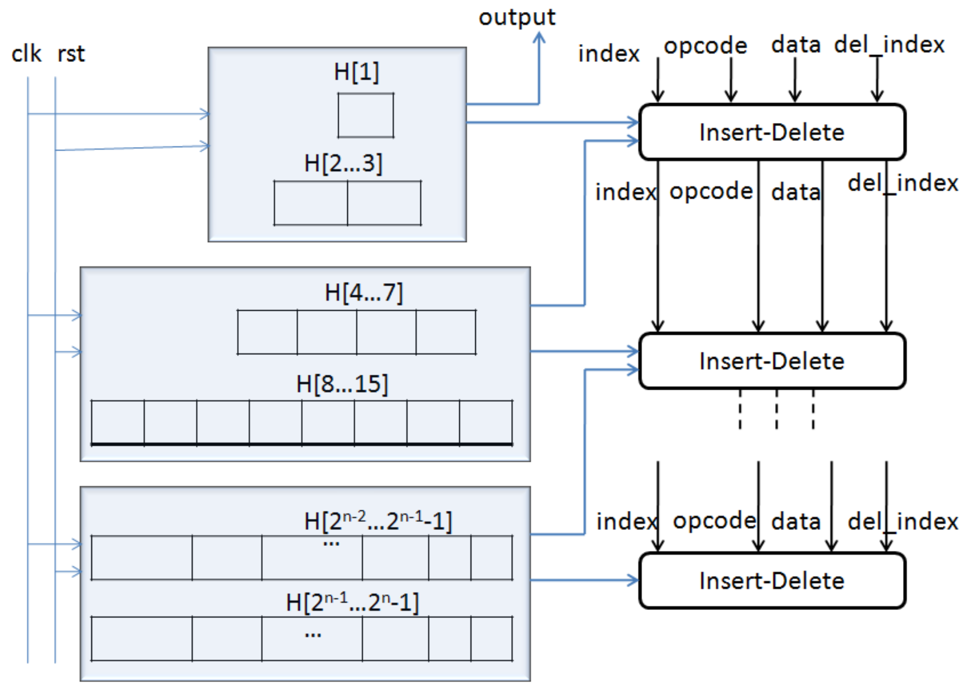

Figure (3.10) Sharing *Insert-Delete* hardware resulting reducing combinational logic by half

by $\beta + 1^{st}$ level will be available after $t + 2$; that means the level β can perform *Write* only at time $t+3$. At $t+2$ the level β becomes idle. For each level, we can see that there is such idle state. For example, while level L_2 perform *comp* at clock 2, then level L_3 performs the *Read* (Figure 3.9). Level L_2 becomes idle at clock 3 while L_3 performs *comp* at that time. Eventually, the level L_2 performs *Write* at clock 4 after the data available by the level L_3 performs. From the Figure 3.9, we can see that at clock 3 the data from L_2 is written at level L_1. That means, the level L_2 suffers at a temporary *hole* at clock 3. This *hole* at level L_2 is compensated while the level L_3 write at L_2 at clock 4. But, the the level L_3 suffers from temporary *hole*. While a level has temporary *hole*, the level is in inactive state; that means there could not be any operation to be performed at that level at that time. In general, for any time t, the β level can not be completed if $\beta + 1$ level can not finish the task of *comp* at $t + 1$. That means we can share hardware between the levels β and $\beta + 1$.

Figure 3.10 illustrates the hardware sharing where a common *Insert-Delete* block is used for two consecutive levels.

Table (3.1) Variation of frequency, execution time and throughput with number of level

Number of Level (β)	Frequency (f) (MHz)	Execution Time (ns)	Throughput (τ) (GB/Sec)
4	318.8	9.41	1.27
8	232.8	12.88	1.85
10	212	14.15	2.12
12	210	14.25	2.52
16	207.2	14.5	3.31
20	173.4	17.3	3.46
24	171.6	17.48	4.10
28	157.45	19.05	4.39
32	143.69	20.87	4.57

3.5 Implementation Result

The proposed design has been simulated by ISim for implementation on Xilinx Sparttan6 XC6SLX4 hardware platform.

(τ) is calculated as:

$$\tau = \frac{\omega \times f}{\chi} \tag{3.1}$$

where ω is the bit length, f is the clock frequency and χ is the number of clock cycle required to compute *insert-delete*. We obtain maximum clock frequency of 207.21 MHz with minimum clock period of 4.82 nano second (ns).

Table 3.1 demonstrates the performance result obtained from simulation. The execution time per level is calculated as:

$$t = \frac{3}{f} \tag{3.2}$$

where β is the number of level and f is the frequency. We use the number of level (β) and bit length (ω) interchangeably. Number of elements in the heap will be $2^{\omega} - 1 = 2^{\beta} - 1$.

Form the table, we found that the obtained clock frequency is not constant, it is inversely proportion to the bit length (β). We obtain maximum frequency = 318.8 MHz for $\beta = 4$, and minimum frequency 143.69 MHz $\beta = 32$. The parameter, execution time is directly proportion to frequency and inversely proportion to β. For example, it takes $\frac{3}{143.69} = 20.87$ ns when $\beta = 32$. Because, it takes 3 cycles (worst case) each stage to complete the task. The relation of throughput is a little bit complex. We can see that, it is directly proportion to frequency which is inversely proportion to β. But, it is directly proportion to β it self. As we have design a fully pipelined architecture, the output can be obtained in each clock cycles as shown at Figure 3.9. We obtain throughput, for example, $143.69 \times 32 = 4.59$ GB/Sec when $\beta = 32$.

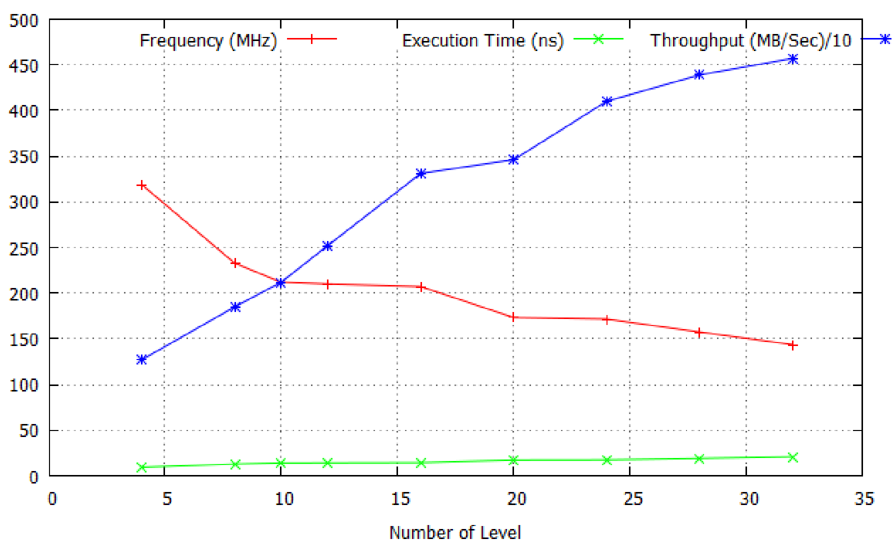

Figure (3.11) Different performance matrices

Figure 3.11 illustrates the graphical presentation of different efficiency parameter with variation of β. From the figure, it is clear that throughput increases even though frequency decreases with the increase of β.

Table (3.2) Performance comparison and hardware complexity.

Design	κ	Flip-flop (F)	SRAM	LUT	f (f) (MHz)	τ (GB/Sec)	Time (t)	Complete Tree ?
[14]	2^{β}	$2^{\beta+1}$	0	8560	-	-	$O(1)$	Yes
[41]	$2\times\beta$	$2^{\beta+1}$	0	1411	-	-	$O(\log n)$	Yes
[17]	$2\times\beta$	$2\times\beta$	$2\times\beta$	-	180	6.4	$O(1)$	No
[9]	$2\times\beta$	$2\times\beta$	$2\times\beta$	-	35.56	10	$O(1)$	No
Our	$\frac{\beta}{2}$	β	β	**1970**	**143.69**	**4.57**	$O(1)$	**Yes**

3.5.1 Hardware Cost

We can visualize hardware cost with some parameters like [17] :

$$C = \beta \times (\kappa + F) + 2^{\beta} \times M$$

where C is the cost for β levels. κ is the numbers of comparators used, F is the number of *flip-flop* for each level and M represents the memory bits. For accessing memory bit, we use static RAM (SRAM). Xilinx provides 2x512 SRAM . So, effectively, we can simulate 2^{34} nodes. As we have addressed two levels of optimization like :*hole* minimization and *hardware sharing*; our design results very much cost effective comparing to the traditional designs [9,14,17]. We used, for example, 1970 number of Look-up tables (LUTs), 2870 number of slices with 800 *flip-flop* register to simulate 2^{32} number of nodes.

Table 3.2 demonstrates comparative analysis of our proposed design with existing ones. As different designs address different issues and implemented in different platform, it will be not fair to have direct comparison. We could see that, our design performs worst comparing to [41] in terms of total number of LUT used. But, as the design of [41] is implemented by using hardware-software co-design, it is very slow in execution ($O(\log n)$. Our design is very much comparable to [9,17]. The design of [9] ensures high throughput with low clock frequency by using cell sizes of 424 bits. Unlike [9,17], our design stands at moderate value of throughput and frequency by ensuring balanced complete binary tree.

PART 4

CONCLUSIONS AND FUTURE WORK

We implement a web based anomaly detection device. The anomaly is detected based on score calculation. The incoming network packets are captured and parsed the packets. The entire anomaly detection engine is based on software. Only the hardware part is the prioritization of anomalous packets.

We propose a hardware realization of parallel binary heap as an application of web based anomaly prioritization. The heap is implemented in pipelined fashion in FPGA platform. The propose design takes $O(1)$ time for all operations by ensuring minimum waiting time between two consecutive operations. We present the various design issues and hardware complexity. We explicitly analyze the design trade-offs of the proposed priority queue implementations.

4.1 Future Scope of Work

The work presented in this thesis leaves several directions for future research. We present some of these ideas here.

- The interface we provide is essentially two parts: one is software part and the other is hardware one. The software part is responsible to parsing network packets and find some score based on some models. The hardware part provides an interface to make priority for the detected anomalous packets. It would be great idea if we can implement the detection part in hardware. In that case, we can achieve high throughput.

- We present the binary heap where each node has maximum two children. In many cases, each node may have n number of items [4]. In that case, each node of the heap will have n sorted data (except the last node). Each time of *insert* or *delete*; we need to assure the heap construction along with the sorted list of each node. There could be a lot of scope to have parallel operation, but it would be little complex in terms of FPGA implementation.

REFERENCES

[1] S. Prasad, Efficient parallel algorithms and data structures for discreteevent simulation, PhD Dissertation, 1990.

[2] Sushil Prasad and I. Sagar Sawant 1995. Parallel Heap: A Practical Priority Queue for Fine-to-Medium-Grained Applications on Small Multiprocessors. Proceedings of 7th IEEE Symposium on Parallel and Distributed Processing (SPDP) 1995.

[3] Xi He, Dinesh Agarwal and Sushil K. Prasad: "Design and implementation of a parallel priority queue on many-core architectures", HiPC 2012, pp. 1-10.

[4] N. Deo and S. Prasad, Parallel heap: An optimal parallel priority queue, The Journal of Supercomputing, vol. 6, no. 1, pp. 87-98, 1992.

[5] G. S. Brodal, J. L. Tradff, and C. D. Zaroliagis, A parallel priority queue with constant time operations, Journal of Parallel and Distributed Computing, vol. 49, no. 1, pp. 4-21, 1998.

[6] A. V. Gerbessiotis and C. J. Siniolakis, Architecture independent parallel selection with applications to parallel priority queues, Theoretical Computer Science, vol. 301, no. 1 Vol 3, pp. 119-142, 2003.

[7] V. Nageshwara Rao, Vipin Kumar: "Concurrent Access of Priority Queues", IEEE Trans. Computers 37(12): 1657-1665 (1988)

[8] S.-W. Moon, J. Rexford, and K. G. Shin, Scalable hardware priority queue architectures for high-speed packet switches, IEEETC: IEEE Transactions on Computers, vol. 49, 2000.

[9] R. Bhagwan and B. Lin, Fast and scalable priority queue architecture for high-speed network switches, in INFOCOM, 2000, pp. 538-547.

[10] H. J. Chao and N. Uzun, A VLSI sequencer chip for ATM traffic shaper and queue manager, IEEE Journal of Solid-State Circuits, vol. 27, no. 11, pp. 1634-1642, November 1992.

[11] K. Mclaughlin, S. Sezer, H. Blume, X. Yang, F. Kupzog, and T. G. Noll, A scalable packet sorting circuit for high-speed wfq packet scheduling, IEEE Transactions on Very Large Scale Integration Systems, vol. 16, pp. 781-791, 2008.

[12] H. Wang and B. Lin, Succinct priority indexing structures for the management of large priority queues, in Quality of Service, 2009. IWQoS. 17th International Workshop on, july 2009, pp. 1-5.

[13] X. Zhuang and S. Pande, A scalable priority queue architecture for high speed network processing, in INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings, april 2006, pp. 1-12.

[14] S.-W. Moon, K. Shin, and J. Rexford, Scalable hardware priority queue architectures for high-speed packet switches, in Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE, jun 1997, pp. 203-212.

[15] R. Chandra and O. Sinnen, Improving application performance with hardware data structures, in Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, april 2010, pp. 1-4.

[16] Yehuda Afek, Anat Bremler-Barr, Liron Schiff: Recursive design of hardware priority queues. Computer Networks 66: 52-67 (2014)

[17] A. Ioannou and M. G. Katevenis, Pipelined heap (priority queue) management for advanced scheduling in high-speed networks, IEEE/ACM Transactions on Networking (ToN), vol. 15, no. 2, pp. 450-461, 2007.

[18] Muhuan Huang, Kevin Lim, Jason Cong: A scalable, high-performance customized priority queue. FPL 2014, pp. 1-4

[19] P. Kuacharoen, M. Shalan, and V. J. Mooney, A configurable hardware scheduler for real-time sys- tems, in Engineering of Reconfigurable Systems and Algorithms, T. P. Plaks, Ed. CSREA Press, 2003, pp. 95-101.

[20] Abhishek Das, David Nguyen, Joseph Zambreno, Gokhan Memik, Alok N. Choudhary: An FPGA-Based Network Intrusion Detection Architecture. IEEE Transactions on Information Forensics and Security 3(1): 118-132 (2008)

[21] Abhishek Das, Sanchit Misra, Sumeet Joshi, Joseph Zambreno, Gokhan Memik, Alok N. Choudhary: An Efficient FPGA Implementation of Principle Component Analysis based Network Intrusion Detection System. DATE 2008: 1160-1165

[22] Sailesh Pati, Ramanathan Narayanan, Gokhan Memik, Alok N. Choudhary, Joseph Zambreno: Design and Implementation of an FPGA Architecture for High-Speed Network Feature Extraction. FPT 2007: 49-56

[23] M. Abadeh, J. Habibi, Z. Barzegar, and M. Sergi, "A parallel genetic local search algorithm for intrusion detection in computer networks," Eng. Appl. of AI, Vol. 20, No. 8, pp. 1058-1069, 2007.

[24] Christopher Krgel, Giovanni Vigna: "Anomaly detection of web-based attacks". ACM Conference on Computer and Communications Security 2003: 251-261

[25] J. Cannady, "Artificial neural networks for misuse detection," in Proc. of the National Information Systems Security Conference, Arlington, VA, 1998, pp. 443-456.

[26] Varun Chandola, Arindam Banerjee, and Vipin Kumar, "Anomaly detection: A survey," ACM Comput. Surv. Vol.41, No. 3, pp. 1-58, 2009.

[27] C. Krugel, G. Vigna, and W. Robertson, "A multi-model approach to the detection of web-based attacks," Computer Networks, Vol. 48, No. 5, pp. 717-738, 2005.

[28] K. Das, and J. Schneider, "Detecting anomalous records in categorical datasets," in Proc. of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, USA, 2007, pp. 220-229.

[29] H. Debar, M. Becker, and D. Siboni, "A neural network component for an intrusion detection system," in IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, 1992, pp. 240-250.

[30] D. Denning, "An intrusion-detection model," IEEE Trans. on Software Engineering, Vol. 13, No. 2, pp. 222-232, 1987.

[31] F. Valeur, D. Mutz, and G. Vigna, "A Learning-Based Approach to the Detection of SQL Attacks, DIMVA, 2005, pp. 123-140.

[32] J. Hochberg, K. Jackson, C. Stallings, and J. Mcclary, "An automated system for detecting network intrusion and misuse," Computers & Security, Vol.12, No. 3, pp. 235-248, 1993.

[33] D. Janakiram, V. Reddy, and A. Kumar, "Outlier detection in wireless sensor networks using Bayesian belief networks," in Proc. of the 1st International Conference on Communication System Software and Middleware, 2006, pp. 1-6.

[34] M. Kiani, A. Clark, and G. Mohay, "Evaluation of Anomaly Based Character Distribution Models in the Detection of SQL Injection Attacks," ARES 2008, pp. 47-55.

[35] S. Lee, and D. Heinbuch, "Training a neural-network based intrusion detector to recognize novel attacks," IEEE Transactions on Systems, Man & Cybernetics, Vol. 31, No. 4, pp. 294-299, 2001.

[36] S. Mukkamala, G. Janoski, and A. Sung, "Intrusion detection using neural networks and support vector machines," in 2002 International Joint Conference on Neural Networks (IJCNN), Honolulu, HI, USA, 2002, pp. 1702-1707.

[37] K. Rieck, and P. Laskov, "Language models for detection of unknown attacks in network traffic," Journal in Computer Virology, Vol. 2, No. 4, pp. 243-256, 2007.

[38] W. Robertson, G. Vigna, C. Krugel, R. Kemmerer, "Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks, NDSS, 2006.

[39] G. Vigna, F. Valeur, D. Balzarotti, W. Robertson, C. Kruegel, and E. Kirda, "Reducing errors in the anomaly-based detection of web-based attacks through the combined analysis of web requests and SQL queries,", Journal of Computer Security, Vol.17, No. 3, pp. 305-329, 2009.

[40] L. Rabiner, "A tutorial on hidden Markov model and selected applications in speech recognition", in Proc. of IEEE, Vol. 77, No. 2, pp. 257-286, 1989.

[41] Chetan Kumar, Sudhanshu Vyas, Ron K. Cytron, Christopher D. Gill, Joseph Zambreno, Phillip H. Jones: "Hardware-software architecture for priority queue management in real-time and embedded systems". IJES 6(4): pp. 319-334 (2014)

Appendix A

SOURCE CODE

The following RTL generate insert-delete logic for root and i^{th} level:

```
// this module contains the logic for priority queue with
// min heap algorithm.
// assumptions: input is 32 bits wide ,
// storage element is 32 bits wide and 32 depth
// heap_count max value is 32 ( 32 bits wide)

`define WIDTH 32

module prio_q_heap_algo ( clk , rst_n , inp_data , opcode , heap_count ,
    heap_root , num_heap_lvl , last_data ,
  wire1 , wire2 , wire3 );



input                    clk ;
input                    rst_n ;
input [`WIDTH-1:0]       inp_data ;
input                    opcode ;

output [`WIDTH-1:0]      heap_root;
// output [`WIDTH-1:0]      heap_count; // This code is for testing
// output [`WIDTH-1:0]      num_heap_lvl; // This code is for testing
// output [`WIDTH-1:0]      last_data;    // This code is for testing

// output [`WIDTH-1:0]      wire1;
// output [`WIDTH-1:0]      wire2;
// output [`WIDTH-1:0]      wire3;
```

```
wire      addition , deletion ;
wire    [`WIDTH-1:0]     heap_root;
wire    [`WIDTH-1:0]     last_data;
wire    [`WIDTH-1:0]     felem_vl;

reg  [`WIDTH-1:0]      num_heap_lvl;
reg     [`WIDTH-1:0]     heap_count;
reg     [`WIDTH-1:0]     stored_data [`WIDTH-1:0];
reg     [`WIDTH-1:0]     hole [`WIDTH-1:0];

reg [`WIDTH-1:0] wire1, wire2, wire3, wire4;
reg [`WIDTH-1:0] index1 = 1, index2, index3, index4;
reg [`WIDTH-1:0] del_index1, del_index2, del_index3, del_index4;
reg [`WIDTH-1:0] tmp_data;

parameter    FULL        = `WIDTH'h1F;
parameter    EMPTY       = `WIDTH'h0;

assign addition          = opcode;  // 1 for addition
assign deletion          = ~opcode; // 0 for deletion

assign full              = (sptr == FULL);
assign empty             = (sptr == EMPTY);

assign heap_root         = stored_data[1];
assign last_data         = stored_data[heap_count];

integer i, hole_count = 0;
integer del_index = 0;
integer index = 0;

//All nodes are stored with empty values
```

```
always@(posedge rst_n)
begin
         if(!rst_n)
          begin
          for(i = 0; i < 32; i = i + 1) begin
                stored_data[i] = 'WIDTH'h0;
                                        hole[i] = 'WIDTH'h0;
                            end
     end
end



// Below always block counts the Incoming heaps/inputs
always @ ( posedge clk or negedge rst_n)
   begin : heap_counter
        if(!rst_n)
           heap_count <=  'WIDTH'h0;
       else if (addition )
           heap_count <=  heap_count + 1'b1;
           else if (deletion)
           heap_count <=  heap_count - 1'b1;
       else
           heap_count <=  heap_count;
   end

         assign insert_path = find_path(heap_count, hole_counter, hole_reg);



         always @ ( posedge clk or negedge rst_n)
begin : storage_element
         if(!rst_n)
                  begin
                            stored_data[1] <= 'WIDTH'h0;
                  end
```

```
else if (addition) // opcode = 1
        begin
            if(heap_count == 5'h1) // only one data
                stored_data[index1] = inp_data; //assign to
                    root

            else
            begin
                if(inp_data < stored_data[index1])
                begin
                    wire1 = stored_data[index1]; // root
                        goes to next level
                    stored_data[index1] = inp_data;  //
                        root is replace
                end
                else
                    wire1 = inp_data; // data goes to next
                        level
            end
        end
else if (deletion) //opcode == 0
        begin
            stored_data[del_index] = 5'h0;
            if(heap_count == 5'h2) // only 2 elements
            begin
                stored_data[del_index] = stored_data[2];
                stored_data[2] = 5'h0;
                hole_count = hole_count + 1'b1; // hole_count
                    incremented
                hole[hole_count] = 5'h2; // the address of
                    hole
            end
            else
            begin
```

```
                    if(stored_data[2] < stored_data[3]) // More
                        than two elements
                    begin
                            stored_data[del_index] = stored_data
                                [2];
                            stored_data[2] = 5'h0;
                            del_index2 = 2; // changing del_index
                    end
                    else
                    begin
                            stored_data[1] = stored_data[3];
                            stored_data[3] = 5'h0;
                            del_index2 = 3; // changing del_index
                    end
                end
                if(stored_data[del_index2*2] == 5'h0 || stored_data[
                    del_index2*2 + 1] == 5'h0) // finding hole
                begin
                    hole[hole_count] = del_index2;
                    hole_count = hole_count + 1'b1;
                end
            end

end

//2nd level
always @ (posedge clk or posedge wire1 or del_index2)
begin


 if (addition)
     begin
   if(tmp[num_heap_lvl -1] == 1'b0) // left branch
                begin
```

```
                    if(wire1 < stored_data[2*index1]) // replace
                      current node
                         begin
                              wire2 = stored_data[2*index1];
                              stored_data[2*index1] = wire1;

                         end
                    else // not replace
                         begin
                              if(heap_count > 5'h3)
                                   wire2 = wire1;
                              else
                                   stored_data[index1*1]
                                     = wire1;



                         end
                    index2 = 2*index1; // left child
               end
          else // right branch
               begin
                    if(wire1 < stored_data[2*1+1]) // replace
                      current node
                         begin
                              wire2 = stored_data[2*1+1];
                              stored_data[2*1+1] = wire1;
                                //

                         end
                    else
                         begin
                              if(heap_count > 5'h3) // next
                                level exists
                                   wire2 = wire1;
```

```
                                        else
                                                stored_data[2*1+1] =
                                                    wire1;
                                    end
                              index2 =  2*index1 +1; // right child
                        end
            end

else if (deletion )
                  begin
                  //      stored_data[1] = 5'h0;
                        if(heap_count <= del_index2*2) // no data in next
                            level
                        begin
                              stored_data[del_index2] = stored_data[
                                  del_index2*2];
                              hole_count = hole_count + 1'b1;
                              hole_reg[hole_count] = del_index2*2
                        end
                        else
                        begin
                              if(stored_data[del_index2*2] < stored_data[
                                  del_index2*2+1])
                              begin
                                    stored_data[del_index2] = stored_data[
                                        del_index2*2]; // parent is
                                        replaced
                                    del_index3 = del_index2*2; // new
                                        index calculated
                                    stored_data[del_index2*2] = 5'h0; //
                                        present one becomes empty
                              end
                              else
                              begin
```

```
                                        stored_data[del_index2] = stored_data[
                                            del_index2*2 +1]; // parent is
                                            replaced
                                        stored_data[del_index2*2 +1] = 5'h0;
                                            // present one becomes empty
                                        del_index3 = del_index2*2+1; // new
                                            index calculated
                              end
                    end
                    if(stored_data[del_index3*2] == 5'h0 || stored_data[
                        del_index3*2+1] == 5'h0) begin // to check leaf
                        node or not
                              hole[hole_count] = del_index3; // hole is
                                  created here
                              hole_count = hole_count + 1'b1;
                    end
          end
end

//3rd level
always @ ( posedge clk or posedge wire2)
begin

 if (addition)
       begin
   if(tmp[num_heap_lvl -2] == 1'b0) // left child
                    begin
                              if(wire2 < stored_data[index2*2])
                                        begin
                                                  wire3 = stored_data[index2*2];
                                                       // go to next level
                                                  stored_data[index2*2] = wire2;
                                                        // replace
```

```
                        end
                    else
                        begin
                            if(heap_count > 5'h7) // next
                              level
                                wire3 = wire2;
                            else
                                stored_data[index2*2]
                                  = wire2    ;
                        end
                    index3 = index2*2; // left child
                end
            else  // for right child path
                begin
                    if(wire2 < stored_data[index2*2+1])
                        begin
                            wire3 = stored_data[index2
                              *2+1]; // current node
                              data goes to next level
                            stored_data[index2*2+1] =
                              wire2;  // replace current
                               node

                        end
                    else
                        begin
                            if(heap_count > 5'h7)
                                wire3 = wire2;
                            else
                                stored_data[index2
                                  *2+1] = wire2 ;
                        end
                end
                index3 = index2*2+1; // right child
```

```
        end

else if (deletion )
                begin
                //      stored_data[1] = 5'h0;
                        if(heap_count <= del_index3*2) // no data in next
                            level
                        begin
                                stored_data[del_index3] = stored_data[
                                    del_index3*2];
                                hole_count = hole_count + 1'b1;
                                hole_reg[hole_count] = del_index3*2
                        end
                        else
                        begin
                                if(stored_data[del_index3*2] < stored_data[
                                    del_index3*2+1])
                                begin
                                        stored_data[del_index3] = stored_data[
                                            del_index3*2]; // parent is
                                            replaced
                                        del_index4 = del_index3*2; // new
                                            index calculated
                                        stored_data[del_index3*2] = 5'h0; //
                                            present one becomes empty
                                end
                                else
                                begin
                                        stored_data[del_index3] = stored_data[
                                            del_index3*2 +1]; // parent is
                                            replaced
                                        stored_data[del_index3*2 +1] = 5'h0;
                                            // present one becomes empty
                                        del_index4 = del_index3*2+1; // new
```

```
                                                    index calculated
                                    end
                            end
                            if(stored_data[del_index4*2] == 5'h0 || stored_data[
                                del_index4*2+1] == 5'h0) begin // to check leaf
                                node or not
                                    hole[hole_count] = del_index4; // hole is
                                        created here
                                    hole_count = hole_count + 1'b1;
                            end
                    end
    end



endmodule
```

The following code find the insertion path:

```
// this module contains the logic for finding insertion path

`define WIDTH 5

module find_path ( heap_count, hole_counter, hole_reg, insert_path );


        input               heap_count ;
        input    hole_counter;
        input    [`WIDTH-1:0]      hole_reg [`WIDTH-1:0]; //Array of static RAM

        output   [`WIDTH-1:0]      insert_path;




        reg                  [`WIDTH-1:0]     insert_path ;

        wire     [`WIDTH-1:0]     felem_vl;
        wire                 [`WIDTH-1:0]      tmp
   wire             [`WIDTH-1:0]      num_heap_lvl;


        assign num_heap_lvl  = heap_count <= 5'h1 ? 5'h1 : clogb2(heap_count)
            ;  //To find depth
        assign felem_vl       = 2**num_heap_lvl; // To find 1st elment of lst
           level
        assign tmp = heap_count - felem_vl; // Path in binary

        always@(heap_count)
        begin
                if(hole_counter > 0)
```

```
                        insert_path = hole_reg[hole_counter]; //address at
                            hole register
                else
                        insert_path = tmp;      // last available node
        end


/* This fubction find the number of level based on
        number of elements available in heap
*/
        function integer clogb2;
                input [`WIDTH-1:0] value;
                integer         i;
                begin
                        clogb2 = 0;
                        for(i = 0; 2**i < value; i = i + 1)
                                clogb2 = i + 1;
                end
        endfunction


endmodule
```

Appendix B

SIMULATION

We use ISim simulator tool to verify the behaviorial model of our design. The test bench is generated with clock period of 20 ns. **The following code is for test bench:**

```
`define WIDTH 5
`define TOP prio_q_heap_algo_tb

module prio_q_heap_algo_tb ();

reg                          CLK , RST_N;
reg        [`WIDTH-1:0]      INP_DATA;
reg                          OPCODE;

wire       [`WIDTH-1:0]      HEAP_COUNT;
wire       [`WIDTH-1:0]      last_data;
wire       [`WIDTH-1:0]      HEAP_ROOT;
wire       [`WIDTH-1:0]      NUM_HEAP_LVL;

wire       [`WIDTH-1:0]      wire1;
wire       [`WIDTH-1:0]      wire2;
wire       [`WIDTH-1:0]      wire3;


prio_q_heap_algo prio_q_inst (.clk(CLK) , .rst_n(RST_N) , .inp_data(INP_DATA)
    , .opcode(OPCODE) ,
.heap_count(HEAP_COUNT) , .heap_root(HEAP_ROOT) , .num_heap_lvl(NUM_HEAP_LVL),
     .last_data(last_data), .wire1(wire1)
, .wire2(wire2) , .wire3(wire3));
```

```
initial
  begin
//    $recordfile("prio_q_waves");
//    $recordvars('TOP);
  end

initial
  begin
    CLK          = 1'b0;
    RST_N        = 1'b1;
    #20
    RST_N        = 1'b0;
    #60
    RST_N        = 1'b1;
    #1000
    $finish;
  end

initial
  begin
   #70
                @(posedge CLK)
OPCODE  = 1'b1;
        @(posedge CLK)
        OPCODE  = 1'b1;

    INP_DATA     = 'WIDTH'h7;
         @(posedge CLK)
    INP_DATA     = 'WIDTH'h6;
    OPCODE       = 1'b1;
   @(posedge CLK)
    INP_DATA     = 'WIDTH'h11;
    OPCODE       = 1'b1;
```

```
        //  @(posedge CLK)
        //  OPCODE       = 1'b0;
@(posedge CLK)
    INP_DATA     = `WIDTH'h5;
    OPCODE       = 1'b1;
   @(posedge CLK)
    INP_DATA     = `WIDTH'h8;
    OPCODE       = 1'b1;
   @(posedge CLK)
    INP_DATA     = `WIDTH'h3;
    OPCODE       = 1'b1;
   @(posedge CLK)
    INP_DATA     = `WIDTH'h10;
    OPCODE       = 1'b0;
   @(posedge CLK)
    INP_DATA     = `WIDTH'h17;
    OPCODE       = 1'b1;
   @(posedge CLK)
    INP_DATA     = `WIDTH'h18;
    OPCODE       = 1'b1;
   @(posedge CLK)
    INP_DATA     = `WIDTH'h2;
    OPCODE       = 1'b1;
   @(posedge CLK)
    INP_DATA     = `WIDTH'h12;
    OPCODE       = 1'b1;
   @(posedge CLK)
    INP_DATA     = `WIDTH'h31;
    OPCODE       = 1'b1;
   @(posedge CLK)
    INP_DATA     = `WIDTH'h30;
    OPCODE       = 1'b1;
   @(posedge CLK)
    INP_DATA     = `WIDTH'h1;
```

```
    OPCODE        = 1'b1;
   @(posedge CLK)
    INP_DATA      = `WIDTH'h14;
    OPCODE        = 1'b1;
   @(posedge CLK)
    INP_DATA      = `WIDTH'h7;
    OPCODE        = 1'b1;
   @(posedge CLK)
    INP_DATA      = `WIDTH'h24;
    OPCODE        = 1'b1;
   @(posedge CLK)
    INP_DATA      = `WIDTH'h13;
    OPCODE        = 1'b1;
   @(posedge CLK)
    INP_DATA      = `WIDTH'h0;
    OPCODE        = 1'b1;
   @(posedge CLK)
    INP_DATA      = `WIDTH'h24;
    OPCODE        = 1'b1;
   @(posedge CLK)
    OPCODE        = 1'b0;


  end

always
 #20 CLK = ~CLK;

endmodule
```

Figure B.1 demonstrates the out put for different level. We have tested it for five levels. Figure B.2 shows the synthesizing top level design.

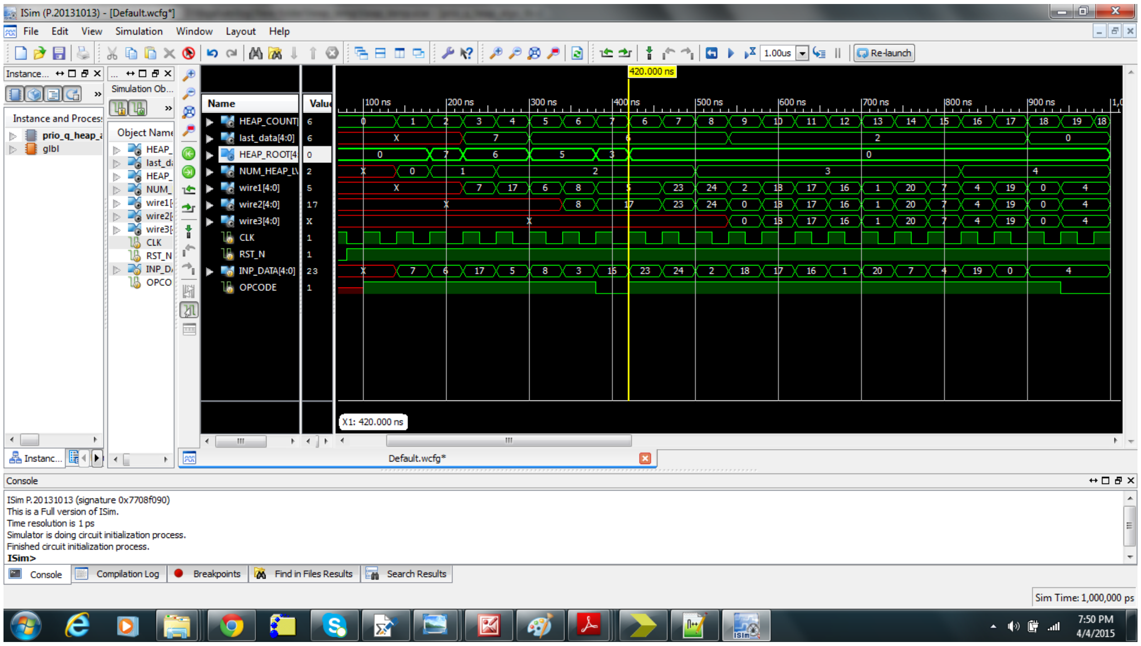

Figure (B.1) Print screen of simulation out put

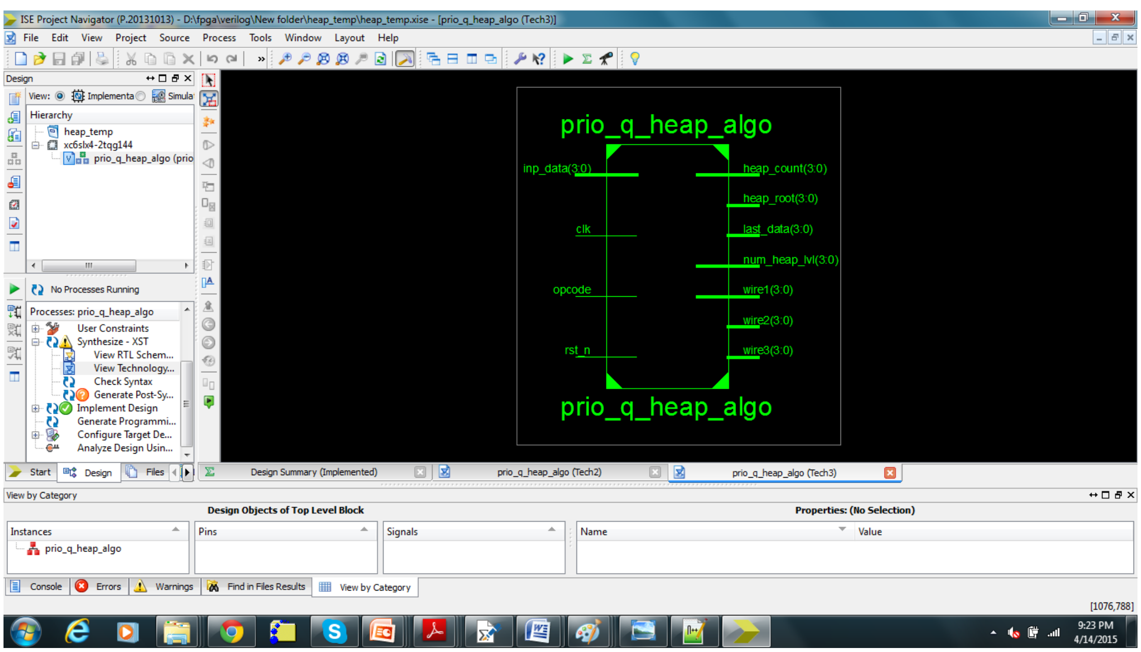

Figure (B.2) Print screen of top level design