Honors Thesis

Hardware Implementation of a Heap-Based Priority Queue

Ethan Miller

Advisor: Dr. John Nestor

May 2021

Table of Contents

Acknowledgements

I would like to thank Dr. John Nestor for his invaluable guidance and support on this thesis. His mentorship and time is greatly appreciated. I would also like to thank John Burk for his assistance with the block diagrams used in this thesis.

Abstract

A heap-based priority queue was designed and implemented in hardware in order to accelerate the storing and retrieval of routes during a VTR FPGA routing run. It is based on the use of block RAMs (BRAMs) to store data pointers and their associated priorities, while additional registers and data pipelines are used to maintain the heap property throughout a run. The use of BRAMs is highly desirable over register arrays as BRAMs are typically cheaper and more available on modern FPGAs. The priority heap circuit was implemented on a Xilinx Artix-7 100T FPGA, with associated PS/PL interface tested on a ZedBoard Zynq-7000 ARM/FPGA SoC Development Board. Performance was measured using the Xilinx Vivado Design Suite, with utilization metrics based on implementation on the Artix-7 100T. When implemented on an Artix-7 100T, a heap with maximum size 65,535 nodes completed an enqueue or dequeue every four clock cycles. At the clock frequency of 100Mhz, this results in an enqueue/dequeue time of 40 ns.

1. Introduction

Field Programmable Gate Arrays (FPGAs) are a class of integrated circuits designed to be reconfigurable by a user into custom digital logic designs. They are composed of arrays of programmable logic elements called Configurable Logic Blocks (CLBs), wires, and programmable switches which can be used to connect logic blocks together to form a complete digital design, as shown in Figure 1. Modern FPGAs also contain memory elements such as basic flip flops or registers and larger memory elements such as Block RAM (BRAM). FPGAs are often implemented along with a microprocessor or SoC (System on a Chip) in order to implement both hardware and software designs on a single chip. The cheap cost, wide availability, flexibility, and ease of deployment causes FPGAs to be a common choice for both hardware prototyping and chip design for which an ASIC (Application Specific Integrated Circuit) would not be economical or needed. Uses include digital image processing, network routers, crypto mining, and ASIC prototyping.

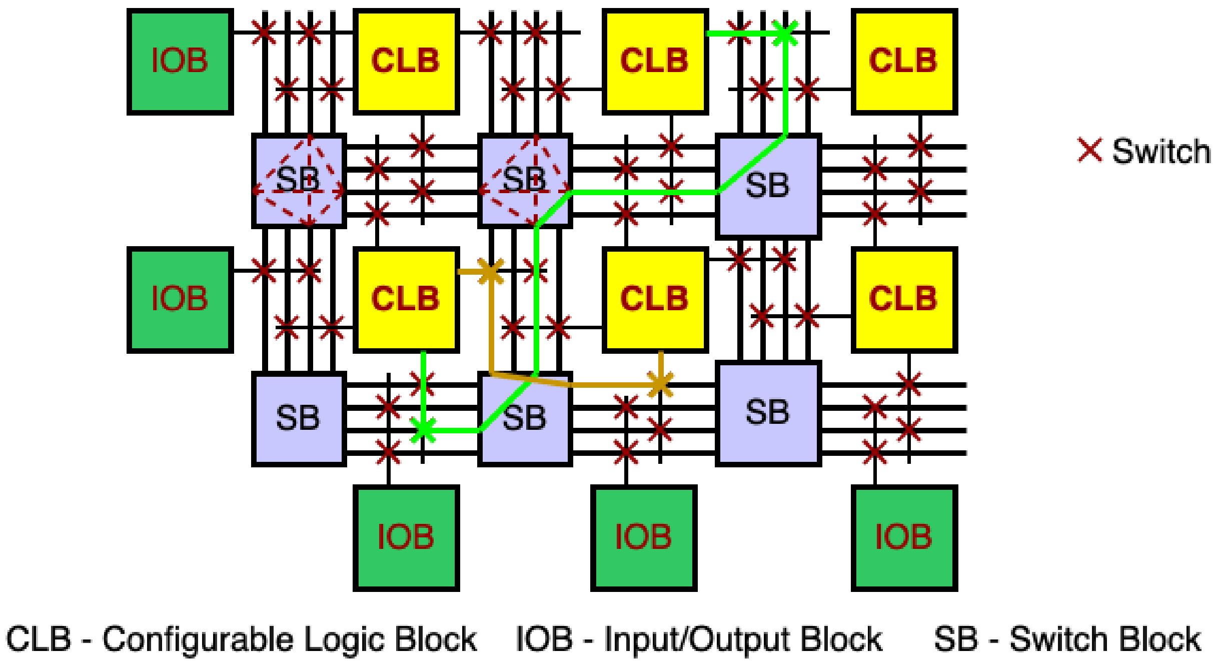


Figure 1 - Simplified Structure of an FPGA

FPGA designs are written using a Hardware Description Language (HDL) which specifies logic gates, memory elements, and interconnections in code, which are then transformed by computer-aided design (CAD) software into a configuration file called a bitstream file which can be used to configure a target FPGA chip. The process of transforming a given hardware description language to a bitstream takes place in four stages: synthesis, packing, placement, and routing, with optimization stages taking place in between. Synthesis is the process of transforming HDL code into the logic gates and wires which compose it. During packing, logic gates are organized together based on connectivity and assigned to logic blocks which can be placed on the FPGA. Placement places these logic blocks into the available physical space on the FPGA. The final stage, routing, connects all of the logic blocks together using programmable interconnect. This is the section of the FPGA CAD flow this thesis will focus on.

During the routing section of an FPGA CAD flow, potential routes are added to a priority queue based on their cost and speed many times for each net. These are then removed from the queue in order to find the most optimal route for each net. This operation is extremely computationally expensive, especially in software, where the cost of adding and removing elements to the queue increases as the queue increases in size, with operations on the queue consuming more than 30% of execution time during routing [3].

This thesis describes the design of a priority queue implemented in hardware on an FPGA which can be interfaced with existing open source FPGA routers to accelerate the routing process. The design utilizes the pipelined approach proposed by Bhagwan and Lin in [5] and several key properties of priority queues to be able to store and retrieve large amounts of entries sorted by priority very quickly. Unlike the design in [5] this design is implemented on an FPGA rather than an ASIC, and has changes made to facilitate operation with standard hardware blocks found on Xilinx FPGAs.

The remainder of this thesis is organized as follows. Chapter 2 provides background on the problem which this priority queue aims to solve, the architecture this design implements, and previous work and designs of hardware priority queues. Also discussed is the basics of FPGA routing, and the specific program which this thesis focuses on interfacing with. Chapter 3 provides technical detail about this specific hardware heap-based priority queue, including differences with previous designs, implementation details, general data flow, and interfaces. Chapter 4 provides FPGA resource utilization statistics and timing measurements of performance. Chapter 5 provides conclusions and ways in which the design can be improved.

2. Background

This section describes the FPGA routing process, various algorithms used in routing, the use of a priority queue in FPGA routing, and other designs that have implemented a priority queue in hardware.

2.1 FPGA Routing Algorithm

The routing process in an FPGA takes as input a *netlist* - a list of desired connections between CLB terminals or IO terminals known as *nets*. The routing process assigns each net to a set of wire segments and switches to form a complete connection. For example, Figure 1 shows the assignment of two nets to routing resources - one highlighted in green, and one highlighted in brown. As all nets in a given design must be routed or the design fails, and multiple nets often pass through the same areas, some form of intelligent routing is necessary to ensure a design is completely routed.

The complexity of FPGA routing and the large number of nets to be routed in complicated designs necessitates computer-aided design (CAD) software which can perform routing and optimization not performable by humans. To this end, a number of routing algorithms have been developed which aim to accelerate and optimize the routing process. This thesis will focus on the PathFinder algorithm and its reliance upon a priority queue structure.

2.1.1 The PathFinder Algorithm and Priority Queues

The PathFinder algorithm was developed by McMurchie and Ebeling in 1995 for the routing of FPGA resources [1] and remains in use to this day. The aim of PathFinder is to “balance the goals of performance and routability” through the use of an iterative algorithm which progressively approaches an optimal routing path. An optimal routing path is defined as one which successfully routes all nets while also minimizing delay to critical signals. As FPGAs have limited routing resources, nets must “compete” for available high demand routes. FPGA routing differs from more general purpose integrated circuit routing in that there is a finite amount of routing resources, and the positions of these resources are fixed.

FPGA routing algorithms model the routing problem using a routing resource graph in which nodes represent terminals and wire resources. At their most basic level they attempt to route nets using a shortest path algorithm (such as Dijkstra’s algorithm) with basic obstacle avoidance; once a routing resource has been used by a given shortest route, it cannot be used again by another route between terminals. This approach will result in unroutable nets on complex designs, so routing algorithms typically utilize an approach called rip up and retry in which sections of the route which are found to be suboptimal are removed from the routing path and routing is performed again to route all nets successfully. The problem with this naive approach, and the main improvement of PathFinder over previous algorithms is that the success of the rip up and retry can be dependent on the order in which nets are routed, as resources which are more important to later nets can be claimed by lower priority nets early in the process.

The PathFinder algorithm deals with this issue through the use of assigning costs to each node, defined as the cost of routing a given net through that node. The cost of a given node, c_n is based on the delay, d_n, of utilizing that node, h_n, the historical congestion on the node and p_n, or the number of other nets which utilize the node, or $c_n = (d_n + h_n) * p_n$. The advantage of this approach can be seen in Figure 1.

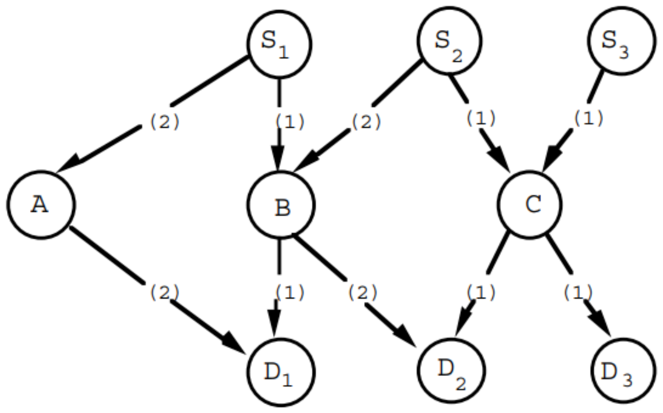
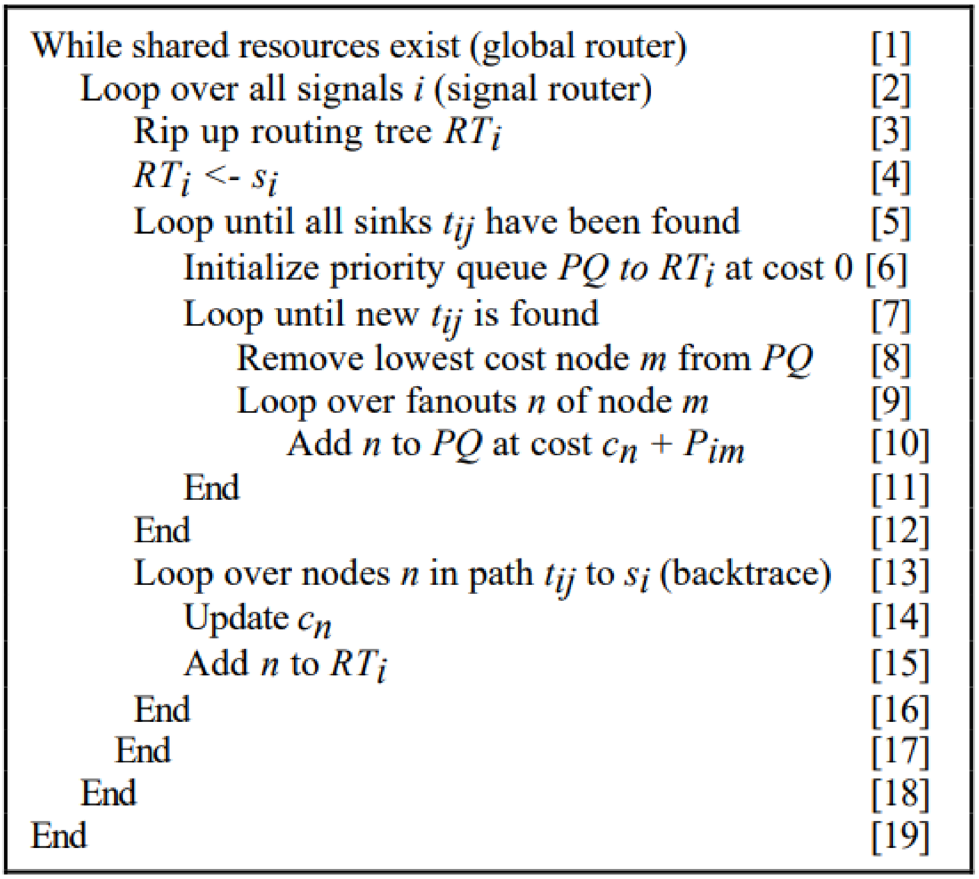


Figure 1: Example route with second order congestion [1]

In this example, three signals must be routed from S_n to D_n respectively. A naive routing approach, i.e. one that does not include historical congestion, would attempt to route signal 1 through node B, as it is the shortest path to the signal destination. Upon routing signal 2 through node C, it would be discovered that signal 3 could not be routed at all. In order for the naive algorithm to correctly route all three nodes, signals 1 and 2 would have to be rerouted, however the order in which they are rerouted becomes crucial. If signal 1 is rerouted first, it will continue utilizing node B until signal 2 is routed first, if at all. Thus the importance of historical congestion becomes clear. Every routing attempt in which a node is shared, in this case nodes B and C, the cost of routing through that node will increase. Eventually, it will become an overall cheaper option to route signal 1 through node A rather than node B even if node B is closer to signal 1 origin as the algorithm recognizes the shared need for node B.

The PathFinder algorithm consists of two main components: the negotiated congestion router and the priority queue. The negotiated congestion router deals with the routing of nets and the assigning of costs to each node in the route. As costs are assigned to nodes, it becomes necessary to store these routed nodes, ordered by their costs. Enter the priority queue. A priority queue is a data structure in which data is stored alongside a given priority and retrieved in order corresponding to that priority. This means that higher priority items will be pushed to the front of the queue to be removed earlier upon a dequeue. The necessity of such a structure can be seen in Figure 2.



```
While shared resources exist (global router)                    [1]
  Loop over all signals $i$ (signal router)                     [2]
    Rip up routing tree $RT_i$                                  [3]
    $RT_i$ <- $s_i$                                             [4]
    Loop until all sinks $t_{ij}$ have been found               [5]
      Initialize priority queue $PQ$ to $RT_i$ at cost 0        [6]
      Loop until new $t_{ij}$ is found                          [7]
        Remove lowest cost node $m$ from $PQ$                   [8]
        Loop over fanouts $n$ of node $m$                       [9]
          Add $n$ to $PQ$ at cost $c_n + P_{im}$                [10]
      End                                                       [11]
    End                                                         [12]
    Loop over nodes $n$ in path $t_{ij}$ to $s_i$ (backtrace)   [13]
      Update $c_n$                                              [14]
      Add $n$ to $RT_i$                                         [15]
    End                                                         [16]
   End                                                          [17]
  End                                                           [18]
End                                                             [19]
```

Figure 2: Negotiated Congestion Router Algorithm [1]

Much of this algorithm lies outside the scope of this thesis, however there are several lines of interest. Lines 8-10 and 13-16 concern removing the lowest cost node from the priority queue, looping through every connected node, and adding those connected nodes back to the queue with an updated cost. As these operations on the priority queue occur for all routes and multiple paths for each route, enqueues and dequeues are frequent and can become costly for large designs.

2.2 VTR

VTR (Verilog to Routing) is an open-source CAD flow for FPGAs developed at the University of Toronto and contributed to by various universities and tech companies [2]. It provides a complete workflow through which digital circuit designs written in the HDL Verilog can be synthesized, optimized, packed, placed, and routed onto a given architecture. It consists of four major software components: ODIN II, the HDL synthesizer, ABC, a logic optimizer, VPR, the pack, place, and router, and FASM, which produces a bitstream file based on the output of the previous tools. This thesis focuses on the acceleration of the routing stage of VPR, specifically interactions with the priority queue.

2.2.1 VPR

VPR (Verilog Place and Route) is the section of VTR that performs the packing, placing, and routing of a given design onto an FPGA. As this thesis is concerned with acceleration of the routing stage, that will be the focus of this section. In its current implementation, VPR utilizes a software priority queue based on a heap to maintain node cost during a route. A heap is a data structure well suited for implementing a priority queue. It is based on a binary tree structure embedded in a one-dimensional array.

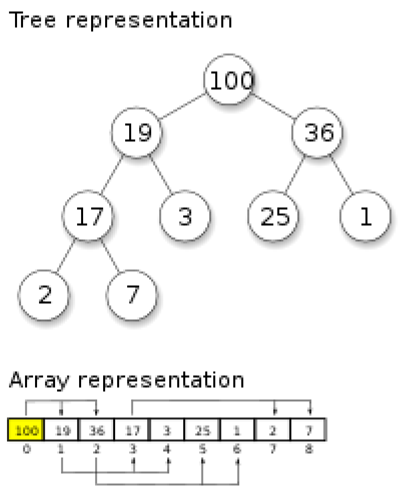


Figure 3: Tree and Array Representation of a Heap [6]

For any given node in the tree, that node has a higher priority value assigned to it than either of its children. Thus the heap is always partially sorted and the highest priority values can be extracted at the root node, or the node at the top of the heap. In VPR's implementation, the heap is ordered based on a key which represents the cost of utilization of a given node, with the cheapest elements at the top of the heap.

apex2.blif				
max_size	num_inserts	num_back_inserts	num_get_head	num_empties
2353	197279	797	45037	798
9385	9291070	918080	2318424	55119
1901	1328977	336404	334121	25276
6967	1684106	398535	420596	29128
11266	7192909	627468	1969790	40579
12028	2243701	481011	573711	34197
15988	2896955	476890	822722	33311
12144	5112516	691964	1294563	42338
8894	3848208	604328	945695	40429

Table 1: Operations on the heap

In order to characterize the properties of the heap utilized by VPR, the VPR source code was modified to record a number of statistics related to the heap. These statistics were collected and collated by a bash script which produced the above table. Table 1 shows the maximum size and number of various operations on the heap. Each row of this table is a separate run of VPR with different amounts of routing resources allocated to the run. As more resources are allocated, the maximum size of the heap increases as there are more possible paths for any given net to run through. The size of the heap grows to nearly sixteen thousand elements at its largest size, and millions of inserts and retrievals are performed every run. For a software heap-based priority queue, inserts and extracts run in $O(\log n)$ time, making them very efficient compared to simple list based priority queues [7]. However, in comparison to hardware, memory accesses are very slow in modern CPUs versus same-chip memory access on FPGAs. In [3], Rios finds that operations on the heap take 25 to 30 percent of the router's overall run time. Clearly there is opportunity for optimization, in the form of a hardware based priority queue.

2.3 Previous Work With Hardware Priority Queues

Given the high number of accesses to a priority queue during an FPGA routing run, hardware acceleration is desirable, and several different approaches have been attempted by researchers. In this section, various forms of hardware priority queues will be discussed, as well as the architecture that this thesis implements.

2.3.1 Shift Register Queue

The shift register priority queue is a queue based on a shift register hardware structure. A shift register is a series of registers in which the data is shifted from the output of one register to the input of the next register on each clock cycle in which an enable signal is present. In a similar way, in the shift register priority queue, data flows through the queue like it flows through a shift register. Every node in this design has access to its own register and its neighbors' registers. When a new entry is to be enqueued, it is sent to every block. If the priority to be written is lower than the entry in a given node, that node will instead take the value of its neighbor to the right. If the priority is higher, the node will stay with its current value. The block with lower priority currently stored and higher priority to the right will receive the new entry. On a dequeue, all data is shifted to the left by one, and the lowest priority possible is written into the rightmost node.

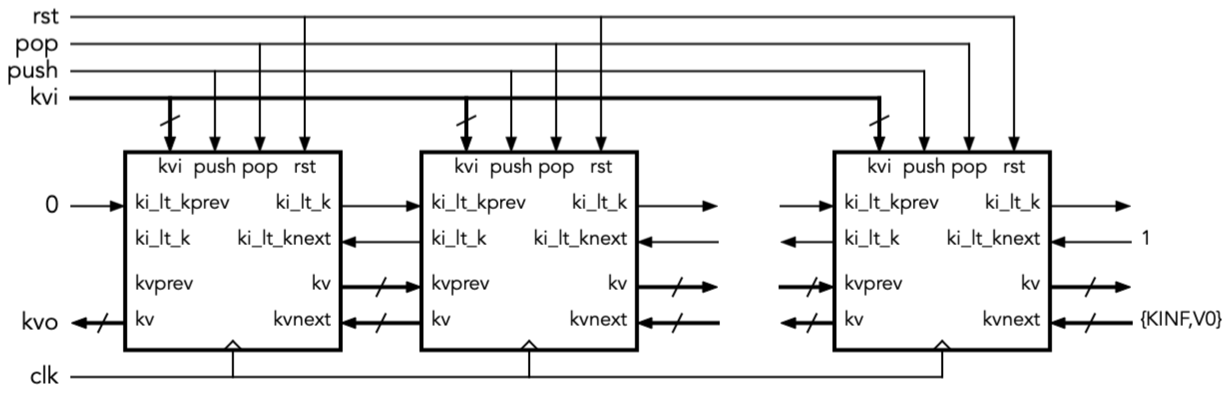


Figure 4: Shift Register Priority Queue Block Diagram

This priority queue architecture is very expandable, as each node in the shift register is the exact same as all the rest. It is also a relatively simple design, which makes implementation much easier. The two main issues which make this design unfeasible for the purposes of this thesis are performance and hardware overhead. The performance of this design is limited by the fact that any new values need to be received by every block in the queue, which leads to large delays as the new value propagates through the queue. In addition, this design is register based, which renders it incapable of storing large amounts of data due to the relatively small amounts of registers present on FPGAs, and requires extra comparator hardware for every element added to the queue.

2.3.2 Systolic Priority Queue

The systolic priority queue, originally designed by Lieserson at Carnegie Mellon University, is a priority queue based on the systolic motion of the heart [4]. The basis of a systolic system is a connected network of digital logic blocks, called processors, that pass data amongst themselves rhythmically, similarly to how the heart passes blood throughout the body. Each of these processors has connections to its neighbors and is capable of performing operations on the data it passes depending on the function of the systolic system. A systolic priority queue is then a systolic system which maintains the properties of a priority heap. A diagram of a simple systolic priority queue is presented in Figure 5.

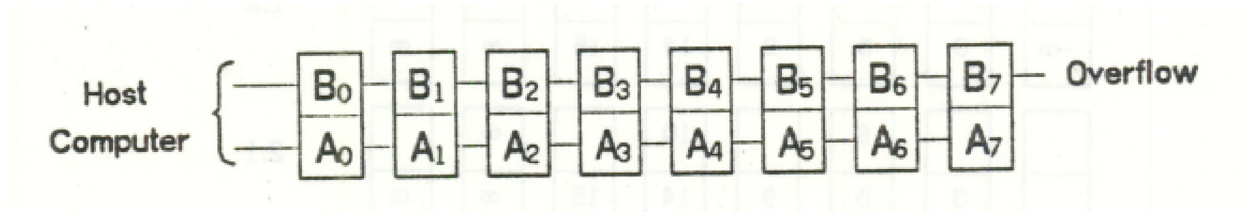


Figure 5: Simple Systolic Priority Queue (from [4])

The systolic priority queue consists of two arrays of registers, the A and B registers. The A registers contain the sorted data, with the smallest key values present in the leftmost registers. The B registers contain elements that have yet to be inserted into their place in the queue. Registers can be initialized to the largest possible key value, allowing for insertion of smaller key values. Input and output happens at the host computer at the left side. Elements which overflow from the registers are discarded on the right side of the diagram. Each clock cycle, the odd and even numbered processors activate alternatively, pulsing data down the line to higher numbered registers. On a clock cycle, whichever half of registers are active perform the following tasks. First, the B register loads value to its left into the register. Next, the processor arranges the elements present in its own registers and the A register of its left neighbor in such a way that $A_{i-1} \leq A_i \leq B_i$. In this way, the larger keys float progressively rightward towards the overflow while lower keys stay to the left to be dequeued. An enqueue is as simple as writing a key to the B_0 register while a dequeue replaces the value in A_0 with the largest possible key.

The systolic priority queue is easy to implement and exceptionally quick, with enqueue and dequeue happening in two clock cycles. Additionally, it avoids the main drawback of the shift-register implementation: the need to send every new entry to every element in the queue. The systolic priority queue takes advantage of the property of priority queues that, at any given time, only the first element of the queue must be correct. The head of the systolic priority queue always has the highest priority of any element, and the remainder of the elements can be sorted in the background, as it does not affect enqueues or dequeues. Despite this, it suffers from two major issues which render it infeasible for being utilized in the context of FPGA routing. The first is the necessity of both A and B registers, which means the total storage of the queue is only half of the memory capacity of the FPGA upon which it is implemented. The second is that a processor is needed for every element, which means required hardware cost scales linearly with extensions to the queue, which is not feasible for the large queue necessary for routing.

2.3.3 QuickQ

Proposed by Rios in 2007 for the explicit purpose of accelerating FPGA routing, the QuickQ is based on the BRAM blocks available in commercial FPGAs instead of electing to use distributed memory such as register memory like the systolic queue [3]. The QuickQ consists of a set of nodes of equal size containing a dual port BRAM, control logic, and multiplexers to route data.

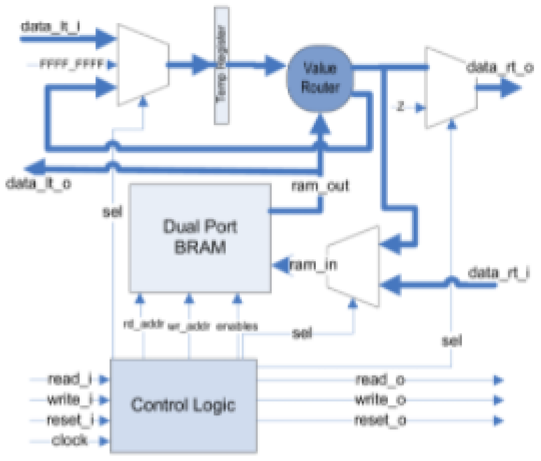


Figure 6: QuickQ Node (from [3])

Enqueue works by inserting items into an already sorted queue by passing the value down the list until its place in the list is found. This takes place within each successive node with the value either finding its place within a node or being passed to the next node for insertion. Once a value finds the correct place within the list, the value at that position is displaced to the next BRAM entry and so on until every element has been shifted down one in the queue. Overflowing items are discarded.

Rios's results are based on an FPGA implementation with width 32, node depth 16, and total nodes to 40, giving the priority queue a total element storage capacity of 640, the node depth times total nodes. In this configuration, a read or write to the queue takes 16 to 18 clock cycles, or roughly the node depth of the design. This leads to a constant time for operations on the queue, a highly desirable property as operations on a software queue scale as the queue grows longer.

The QuickQ priority queue has the advantage of constant time operations on the queue and, being BRAM based, is able to store a greater number of elements when compared to the previous systolic queue, however it is still far below the number of elements a software heap would be capable of handling during a given run. In addition, as additional nodes are added to the QuickQ, the amount of hardware overhead increases linearly as element depth increases (in increments of the BRAM depth). While not as severe as the systolic queue hardware overhead as additional hardware is only necessitated as new levels are added, the design can be improved upon.

2.3.4 Pipelined Heap

The Pipelined heap, or P-heap, was proposed by Bhagwan and Lin for the express purpose of overcoming the memory and hardware scaling issues plaguing other priority queue architectures [5]. Although originally developed for implementation on an ASIC, the design can be ported to and implemented on an FPGA as described in the following sections of this thesis. The basic structure of the P-heap is based on the organization of a normal heap, a binary tree embedded in a one dimensional array, and consists of two elements: the token array, and the binary array. The binary array is the array in which elements and their associated data are stored, while the token array is an extra array with one element per level of the tree in which information about the operations currently being performed at that level are stored. Each element of the binary array consists of an active flag, which represents whether or not the node is currently storing a valid priority, the priority value currently being held at the node, if one is present, and the capacity of the node, or the number of inactive nodes in the subtree rooted at the node. The token array elements consist of the operation which is currently being executed on that level of the p-heap, the value which is to be inserted into that level, and the index of a node at which an operation is occurring.

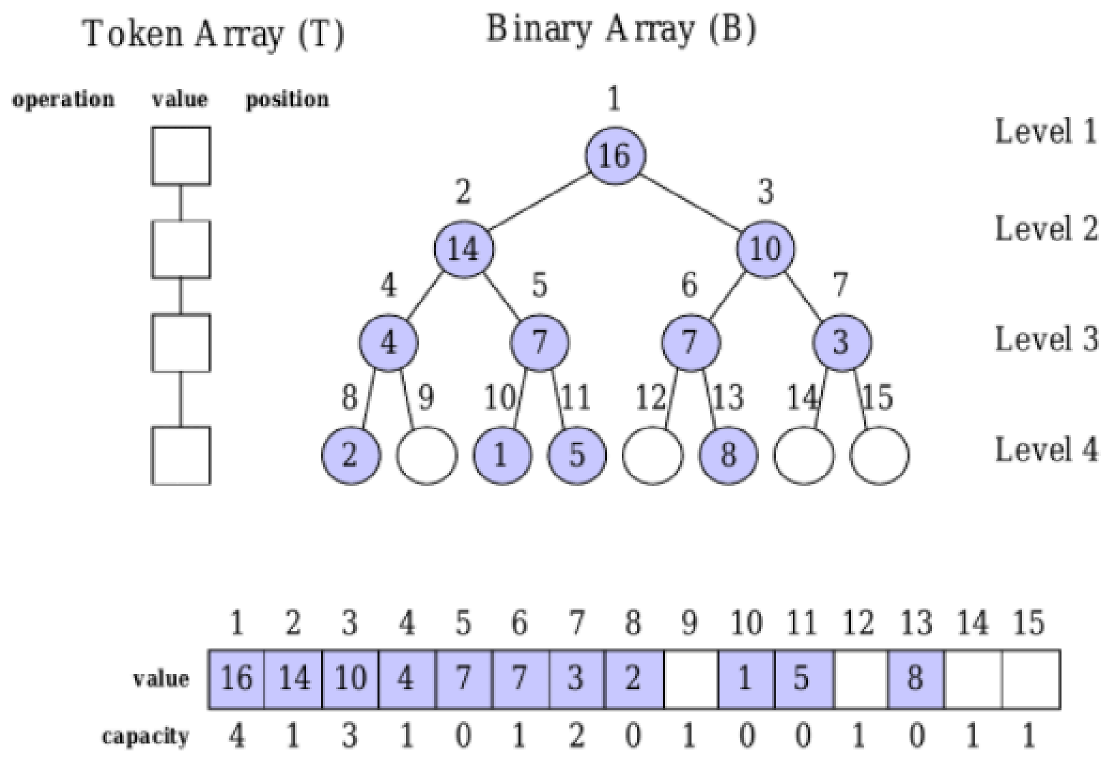


Figure 7: The P-heap data structure [5]

The P-heap satisfies the general heap property, as every active node has an equal or higher priority than any active children. If a node is inactive, its children must also be inactive.

Enqueue and dequeue operations on the P-heap operate on two scopes, the global scope and the local scope. The local operation performs either the enqueue or dequeue operation on the given level while the global operation observes all levels and shifts operations down the queue when they require shifting or removing them from the token array when completed. In this manner, operations shift down the heap level by level, with each level being capable of performing its own operation, thus the name Pipelined heap. Each operation, enqueue or dequeue can be performed in as much time as it takes to perform the operation on one level, as only the result of the operation on the top level matters to the external interface; the rest of the operation is free to complete on the rest of the heap in the background. The specific algorithm to perform these actions will be discussed in the next section, as well as modifications to facilitate implementation in hardware.

The pipelined heap presents a number of advantages over the other priority queue structures. Firstly, the use of BRAM instead of registers is crucial to being able to store the necessary number of elements. Secondly, as additional hardware is only required when increasing levels, and the memory size increases exponentially per level, with each level containing twice as many elements as the previous level, the hardware overhead increases logarithmically with respect to queue length. Finally, operations on the heap execute in constant time, with the execution time equalling the time necessary to perform an operation on the first level.

This particular implementation of the priority heap, being designed for an ASIC, is not completely suited to direct implementation to an FPGA and must be altered to work on that hardware. The major difference is the ability to tailor memory blocks present on the ASIC to function exactly as the algorithm needs them, while the BRAM blocks present on an FPGA are limited in their ability unless compromises are made.

3. Pipelined Heap Implementation

The Heap-Based Priority Queue, the topic of this thesis, is designed to take the place of a software heap in an FPGA CAD routing run. This section explains the design of the queue and the operation of the overarching design and each related submodule.

3.1 Design Requirements

In order for a heap-based priority queue to be suitable for operation in an FPGA CAD routing run, it should fill several requirements. The first is that storage should occur in block

RAM, not register arrays, to maximize the amount of storage available on an FPGA. The second is that hardware overhead should not increase prohibitively as size increases, or that the limiting factor should be the amount of BRAMs available on the board, not LUT/register availability. The third is that it should be able to store more than 26,000 elements, which was the maximum size of the priority queue in the VPR benchmark circuit discussed above.

3.2 Module Descriptions

This section of the thesis deals with implementation of the FPGA pipelined heap, differences from Bhagwan and Lin's ASIC implementation, and data flow through the heap. Complete SystemVerilog code can be found at https://github.com/ilthraim/pheap.

3.2.1 The Pheap Module

The pheap module is the main module of the pipelined heap project. It handles input and output to external modules via ready/valid interfaces and generates and controls all levels of the p-heap. Individual levels are generated algorithmically with their associated data width in a loop and connected to their block rams and additional hardware to shift data between levels. This is where the first key difference is between this thesis and the original P-heap's design. In the original design, the token array was used to maintain operations being done, the position in a given level they are to be done, and the value to be written. In this design, the operation section and the values passed between levels of the token array are maintained, but the start and ending positions of a level are directly shifted from one level to another. As wires instantiated inside of a SystemVerilog generate loop are unable to be accessed out of that loop's scope, a module was created with the explicit purpose to shift these positions and values between levels based on a control signal.

The pheap is ready to accept data when the first level has fully completed its operation and the second level is either done or ready to be shifted to the next level. This is to prevent concurrent memory access issues from taking place between two levels and the corresponding BRAM. Valid data is asserted on the output when the first level has successfully completed a dequeue. On every rising clock edge, the pheap module samples every level of the heap and, based on that level's done status, either shifts the operation to the next level and signals the next level to begin execution or registers that the level has finished the task.

Done tag	Meaning		Opcode tag	Meaning
DONE	The level has completed operations on the BRAM and no additional operations on lower levels are required		FREE	The level is currently free for an operation to be loaded
NEXT_ LEVEL	The level has completed operations on the BRAM, and the operation must be passed to the next level for completion		LEQ	The level is currently enqueuing an element to its level
WAIT	The level has begun operations on the BRAM and is not free to receive new instructions		DEQ	The level is currently removing an element from its level

Figure 8: Level Status Flags vs. Flag Meanings

A SystemVerilog generate loop is used to create every level up to the maximum level specified by the LEVEL parameter. As the first level only contains one element and is the only level from which data is removed, it is altered slightly from the remaining levels.

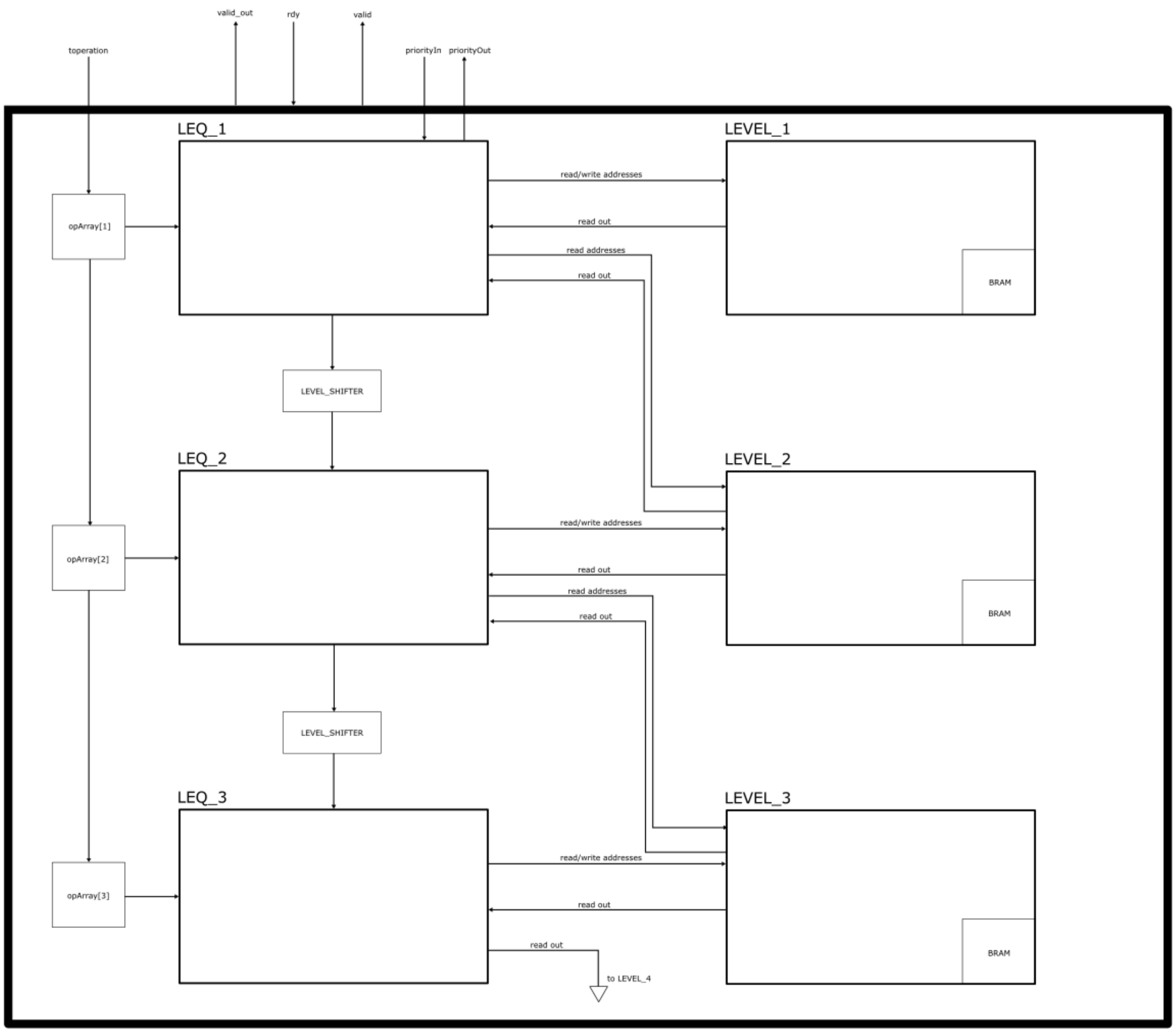


Figure 9: Block Diagram of pheap Module

3.2.2 The pheapTypes file

The pheapTypes file contains a number of parameters, enumerated types, and structs which can be utilized throughout the rest of the project for convenience. This includes the done tags and opcodes found in Figure 8, the width of a priority value, the total number of levels in the heap, and structs to represent both a pheap entry and every level of the opArray, which stores each level's current operation and priority value. Structs in SystemVerilog are a collection of different elements which can be referenced directly by their names or by a bit selection of the struct type. All structs in this design are declared as packed structs to ensure all data belonging to a given struct is stored together. Figure 10 shows how the structs which hold a pheap entry and a level of the opArray are layed out.

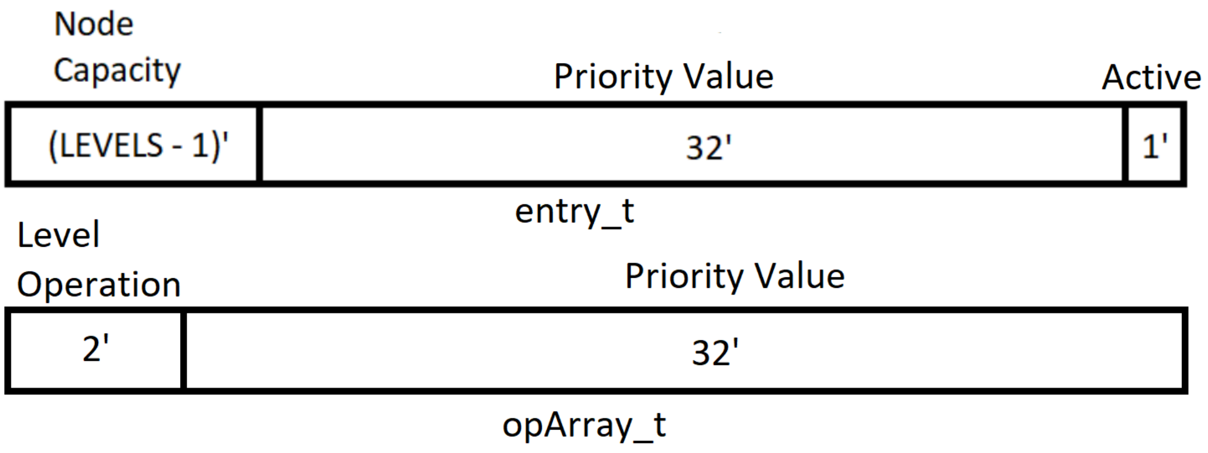


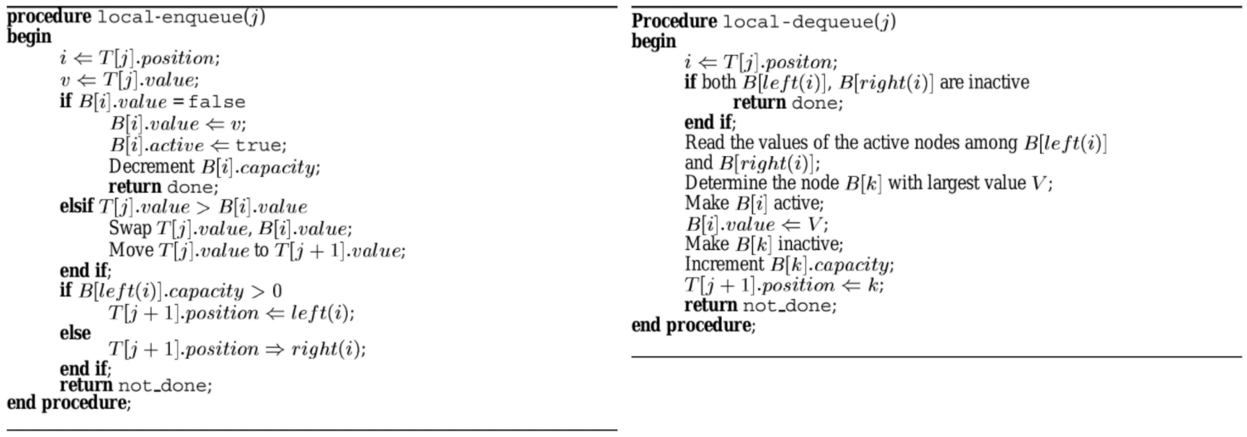
Figure 10: Contents of entry_t (Heap Node) and opArray_t (Operation Array)

3.2.2 The LEQ Module

The LEQ module performs operations on its level as specified by the top level pheap module. It takes as input a start flag, node index to start at, the priority value from the module above it, and the operation to perform. It provides as output a completed flag detailed in the section above, the position the next level should begin operation on, and the priority value that results from the operation on the current level.

The algorithm implemented in this section is largely the same as Bhagwan and Lin’s local enqueue/local dequeue algorithms with several modifications to render them implementable in hardware. The LEQ module consists of a state machine with two states, READ_MEM and SET_OUT. This is necessary because an element must be read from the BRAM and then written back with some changes made. The BRAM blocks require one clock cycle to read an address and one clock cycle to write data to an address. Since nodes may require reading/writing to the same address, this must be done on two clock cycles, even though the BRAM could support a concurrent read/write. In the READ_MEM state, the values of the current node to be operated on and its left and right children are read to the LEQ module. These values have three fields, as seen in Figure 10.

In the SET_OUT state, execution follows either the local enqueue or local dequeue algorithm from above depending on the operation currently being performed.



```
procedure local-enqueue($j$)
begin
    $i \Leftarrow T[j].position$;
    $v \Leftarrow T[j].value$;
    if $B[i].value$ = false
        $B[i].value \Leftarrow v$;
        $B[i].active \Leftarrow$ true;
        Decrement $B[i].capacity$;
        return done;
    elsif $T[j].value > B[i].value$
        Swap $T[j].value$, $B[i].value$;
        Move $T[j].value$ to $T[j+1].value$;
    end if;
    if $B[left(i)].capacity > 0$
        $T[j+1].position \Leftarrow left(i)$;
    else
        $T[j+1].position \Rightarrow right(i)$;
    end if;
    return not_done;
end procedure;
```

```
Procedure local-dequeue($j$)
begin
    $i \Leftarrow T[j].positon$;
    if both $B[left(i)]$, $B[right(i)]$ are inactive
        return done;
    end if;
    Read the values of the active nodes among $B[left(i)]$
    and $B[right(i)]$;
    Determine the node $B[k]$ with largest value $V$;
    Make $B[i]$ active;
    $B[i].value \Leftarrow V$;
    Make $B[k]$ inactive;
    Increment $B[k].capacity$;
    $T[j+1].position \Leftarrow k$;
    return not_done;
end procedure;
```

Figure 11: Local Enqueue and Local Dequeue Algorithms [5]

To perform an enqueue, the current node's, referred to as node *i*, active flag is checked. If it is inactive, the element to be enqueued is written to *i* with a capacity of one less than *i*'s current capacity, and set to active. The LEQ then signals the enqueue operation is complete. If *i* is active and has a lower priority than the value to be queued, the module stores the current node in *i*, decrements the capacity of *i*, and passes *i*'s previous value to the next level. If the priority in *i* is greater than the priority to be written, the priority to be written is passed down to the next level and *i*'s capacity is decremented. The position for the next level to begin operation is either *i*'s left or right child node; if both children have capacity available, the next level will begin with whichever child has the lower priority, otherwise beginning with the left child if there is capacity remaining in that subtree, or the right if not.. The capacity of the right child is never checked. In this way, if the entire heap is filled, incoming values will pass through the heap and eventually be discarded through the lowermost right child. Note the extra check to priority values when compared to the implementation in [5]. This is an attempt to balance the heap such that higher priority values are not exclusively passed down the right side of the tree. In addition, the capacity of *i* is always decremented, unless it is zero. This is because the element will be inserted somewhere in the subtree rooted at *i*, thus the capacity will at some point in execution decrease by one.

To perform a dequeue, the active flags of both children of *i* are checked. If they are both inactive, the process is done, and *i* is removed from the heap, marked inactive, and increases in capacity by one. If there is an active child, the highest priority child is inserted into *i*, the capacity of *i* is increased, and the dequeue moves to that position on the next level. Note it is not necessary to check which child is active, as the inactive child, if present, will have a priority of zero, over which a non-zero priority will take preference. The implementation as seen in [5] is infeasible here, specifically where the child is marked inactive from the current level. As the BRAMs are dual-port, and two reads are necessary in order to access both children of *i*, it would be impossible to then also write to one of the children. The dequeue will only ever make a node inactive on the level it is currently operating.

As LEQ modules require access to the BRAM on the same level and below, they have four connections to BRAM modules: a separate read and write port to the BRAM on the current level, and two concurrent read ports to the BRAM on the level below, to view the children of the node currently being operated upon. Figure 12 shows simplified connections between LEQ modules and level memories. When LEQ2 is inactive, LEQ1 has full access to both LEVEL1 and LEVEL2. When LEQ1 has completed operation, LEQ2 will assert active and have control over LEVEL2.

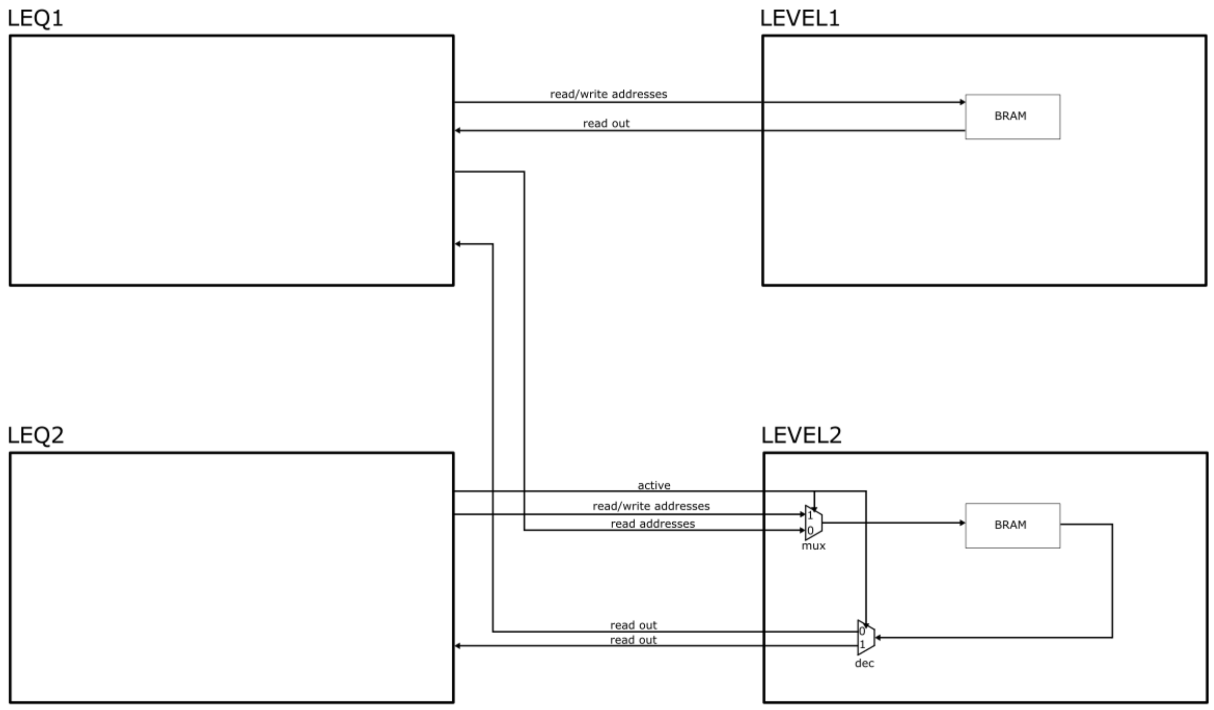


Figure 12: Simplified Connection Between LEQ Modules and BRAMs

3.2.3 The Level module

The level module is, at its most basic, a wrapper for the levelRam module. Its purpose is to route read and write addresses and enables to and from the LEQ on the level above or the same heap level depending on which is currently active and requires access to the BRAM. When the LEQ module on the same level is active, it asserts the topActive signal, which allows it to read and write to the current node of interest. Otherwise, when topActive is low, the level assumes the LEQ above it is active, and passes both read addresses to the BRAM.

3.2.4 The Level Ram module

The levelRam module contains an inferred BRAM parameterized by width and depth based on the specification of the pheap. Each BRAM has width equal to the bit width of a node, and depth of $2^{LEVEL - 1}$: level 2 has depth 2, level 3 has depth 4, and so on. Level 1, only containing one node, is stored in a register, not a BRAM. As the node bit width in this implementation is larger than 36 bits, the maximum available bit width of a BRAM on an ARTIX-7 100T FPGA, a 36 bit and 18 bit BRAM are connected to allow a full node to be stored. Initial heap values are also initialized in this module, with each node having zero priority, the maximum possible capacity for a node at the level, and an inactive flag. These values are stored in a data file which is loaded into the BRAM upon FPGA configuration.

3.2.5 The Level Shifter module

The level_shifter module takes as input the ending position of the LEQ module on the level above it and outputs that position to the starting position of the LEQ module on its level, based on a signal from the pheap controller module.

4. Results

The heap-based priority queue was designed in Verilog. It was synthesized for and implemented using a Xilinx Artix-7 100T FPGA on a Nexys A7 development board. Testing was done with various sizes of the heap to tabulate FPGA resource usage as the heap grows.

4.1 System Timing

The heap can enqueue or dequeue an element once every four clock cycles, giving a constant execution time per operation of 4 times the clock period, in this case 100Mhz, or 40 nanoseconds per enqueue/dequeue. This can be seen in the simulation waveform captured in Figure 13, which is the result of enqueuing 16 random 32 bit values to a queue with 4 levels, size 15. Note the rdy signal is asserted every four clock cycles, signalling the queue is ready to accept additional data from the outside interface. Compared to the design in [3], which required up to 46 clock cycles for node addition or subtraction at the same clock speed, this an acceleration of over ten times. The ASIC design in [5] executes in 28.12 nanoseconds for an implementation with 32 bits, however this is only marginally faster, and this design can be implemented on an easily available FPGA.

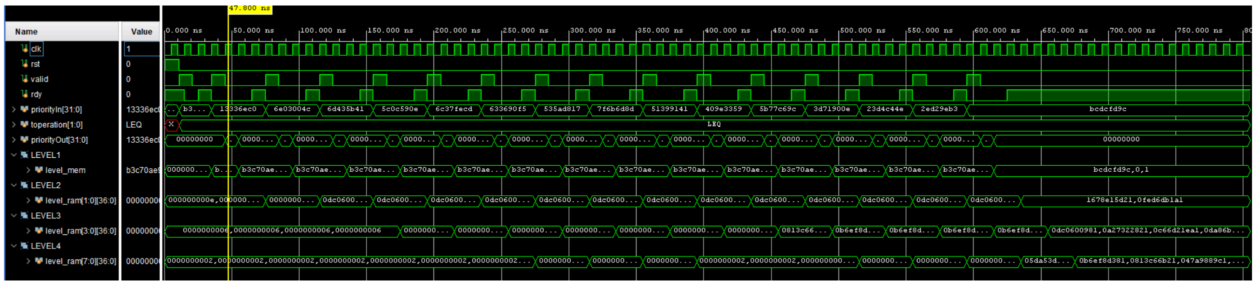


Figure 13: Simulation Waveform in Vivado

4.2 System Scalability

Levels	Heap Capacity	BRAM Utilization (#, %)	LUT Utilization (#, %)	Register Utilization (#, %)
1	1	0, 0.00	56, 0.09	69, 0.05
2	3	1, 0.74	288, 0.45	138, 0.11
3	7	2, 1.48	568, 0.9	208, 0.16
4	15	4.5, 3.33	860, 1.36	279, 0.22
5	31	6, 4.44	1159, 1.83	351, 0.28
6	63	7.5, 5.56	1477, 2.33	424, 0.33
7	127	9, 6.67	1809, 2.85	498, 0.39
8	255	10.5, 7.78	2135, 3.37	573, 0.45
9	511	12, 8.89	2475, 3.90	649, 0.51
10	1023	13.5, 10.00	2865, 4.52	726, 0.57
11	2047	15, 11.11	3210, 5.06	804, 0.63
12	4095	17.5, 12.96	3703, 5.84	883, 0.70
13	8191	23.5, 17.41	4120, 6.5	963, 0.76
14	16383	35.5, 26.3	4547, 7.17	1044, 0.82
15	32767	59.5, 44.07	5012, 7.91	1126, 0.89
16	65535	109.5, 81.11	5481, 8.65	1209, 0.95
17	131071	FAIL	FAIL	FAIL

Table 2: FPGA Resource Utilization vs. Heap Size

To test resource utilization vs heap size, a pipelined heap of different number of levels was synthesized in Xilinx Vivado, with priority size 32 bits, the width of an unsigned integer. As shown in Table 2, the BRAM utilization of the FPGA scales linearly for the first three levels, then a jump to level four, linearly again to level eleven, at which point it begins to increase exponentially. The first jump at four occurs as one 36Kb BRAM is no longer wide enough to store an element as the node capacity field grows. The jump at level twelve occurs when the 36Kb BRAM is no longer deep enough to store the entirety of a level, and additional BRAMs must be multiplexed together in order to store a level. The 16th level alone utilizes 49 of the available 36Kb BRAMs on the FPGA. With a maximum depth of sixteen levels, the total capacity of the heap is 2^{16} - 1, or 65,535. This is far larger than the QuickQ's size of 720 items, and large enough for all but the most resource intensive routing benchmark included with VPR.

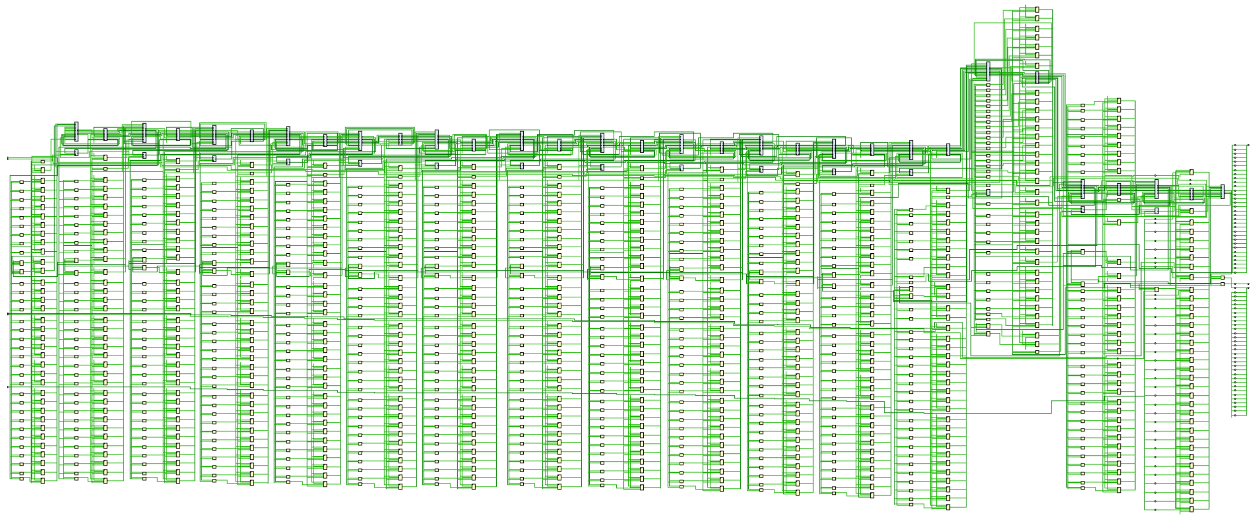


Figure 14: Schematic of Synthesized 16 Level Design

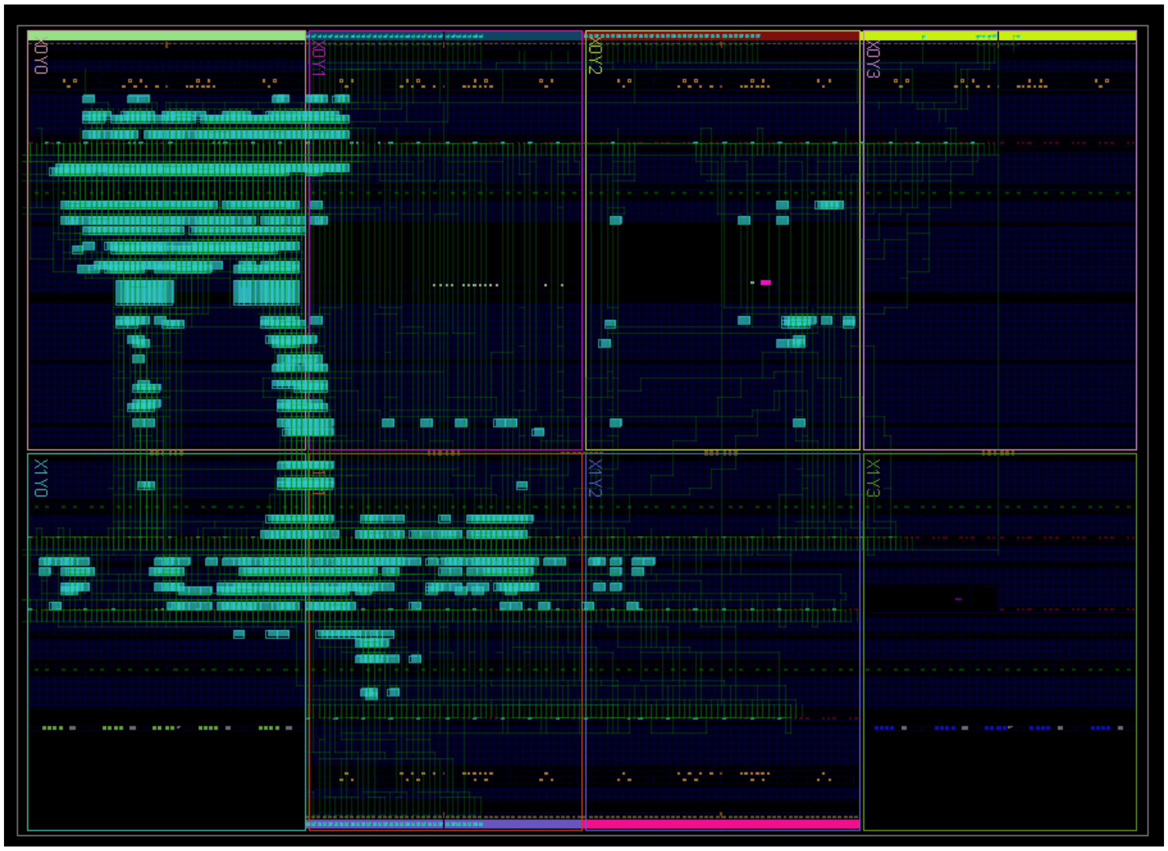


Figure 15: Implemented Pheap Design

5. Conclusions

A hardware heap-based priority queue was designed, implemented on an Artix-7 FPGA, and tested for the purpose of accelerating the FPGA routing process. The hardware heap-based priority queue is based on previous work by Bhagwan and Lin [5], with the main difference being implementation on an FPGA rather than an ASIC. This queue design, referred to as a pipelined heap, consists of a priority heap controller, with a separate controller for each level, and one or more BRAMs to serve as the level's storage element. In this way, insertions can be completed in constant time, as external interfaces only rely on the results of operations performed on the top level. Additionally, as elements are stored in BRAM rather than registers, the number of nodes which can be stored is greatly expanded over other priority queue designs such as systolic or shift register based designs.

In an implementation on an Artix-7 FPGA, a heap-based priority queue was initialized with a priority bit width of 32 and a maximum heap size of 65,535. With this implementation, an enqueue or dequeue could be completed once every four clock cycles, or every 40 ns on a 100Mhz clock. This performance exceeds previous FPGA-based designs in both heap capacity and speed.

Most future improvements should revolve around a more efficient utilization of the available BRAMs on the FPGA. For example, lower levels could be stored in registers, rather than BRAM, specifically for levels 2-10 as they do not make full use of the BRAMs available space. In addition, the growing size of the capacity field for each node requires more BRAMs as it becomes significantly larger as the width of a RAM is no longer sufficient. A way to reduce the impact of the capacity field would greatly increase the maximum capacity of the pheap. Another way in which this design could be improved would be implementing a fast way to reset the heap to default values (i.e. zero priority, maximum capacity and inactive flag). This is a feature necessary for interfacing with VPR, which resets the heap frequently.

References

[1] L. McMurchie and C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs," *Third International ACM Symposium on Field-Programmable Gate Arrays*, Napa Valley, CA, USA, 1995, pp. 111-117, doi: 10.1109/FPGA.1995.242049.

[2] K. E. Murray et al, “VTR 8: High-performance CAD and Customizable FPGA Architecture Modelling,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 13, no. 2, May 2020.

[3] J. Rios, “An efficient FPGA priority queue implementation with application to the routing problem,” *UCSC Technical report uscs-crl-07-01*, May 9, 2007.

[4] C. Leiserson, “Systolic Priority Queues,” *Tech Report CMU-CS-79-115*, April 1979.

[5] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies* (Cat. No.00CH37064), Tel Aviv, Israel, 2000, pp. 538-547 vol.2, doi: 10.1109/INFCOM.2000.832227.

[6] “Heap (data structure),” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Heap_(data_structure).

[7] T. Corman, C. Leiserson, R. Rivest, and C. Stein, Introduction to Algorithms, 2nd ed. MIT Press / McGraw-Hill, 2001.

[8] Sung-Whan Moon, J. Rexford and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Transactions on Computers*, vol. 49, no. 11, pp. 1215-1227, Nov. 2000, doi: 10.1109/12.895938.