**QuickQ Design Notes**

Last Revision: March 18, 2024

## Introduction

This document outlines an implementation of the QuickQ priority queue proposed by Joseph Rios for implementation in FPGAs. The QuickQ implements a priority queue using a set of node modules as shown below. Each node keeps a subset of the queue contents sorted in order of increasing priority. Nodes communicate with each other to shift data right on enqueue operations and left on dequeue operations.



Figure 1 – Connection of qq\_node modules

Each **qq\_node** module maintains a sorted list of enqueued items in a Block RAM with a parameterized depth. During an enqueue operation, the new value data\_i is inserted into the first node where its value exceeds the value already stored; displaced items are then shifted to the right. During a dequeue operation, each value in each node is shifted left by one position.

The full and empty flags indicate conditions for each node. When connected together, the overall queue is considered full when the rightmost node is full, while the queue is considered empty when the leftmost node is empty.

## Top-Level Organization (qq\_node)

Figure 2 shows the organization of the **qq\_node** module, consisting of a datapath and a control unit.

During an enqueue operation, input data is initially stored in the **temp** register. A control unit reads values from the BRAM starting at location 0 using an address generated by a counter (see Figure 3 for timing). Because the BRAM has a synchronous read operation, the address to be read is set before the clock cycle in which the actual read takes place. A comparator is used to determine whether the value read from the BRAM is greater than the **temp** register. When this occurs it is swapped and the displaced value is moved to the right, displacing additional values until either a location is found that is empty (indicated by containing the MAX\_KEY value) or that the last location of the BRAM is read. In the latter case, the value in the **temp** register is enqueued in the **qq\_node** module to the right of the current node.

During a dequeue operation, the first element of the BRAM is replaced with the second element, the second element is replaced with the third element, and so on until either a MAX\_KEY value is read (indicating that an empty value has been read) or the last location in the BRAM has been read (see Figure 4 for timing). In this latter case, the last location in the BRAM is replaced with data from the right neighbor of the node; if there is no right neighbor the data\_rt\_i input is tied to MAX\_VALUE so an empty value will be written.



Figure 2 – qq\_node Organization



Figure 3 – Enqueue Timing in an Empty Node



Figure 4 – Enqueue Timing in an Almost-Full Node



Figure 5 – Enqueue Timing in a Full Node



Figure 4 – Dequeue Timing in an Almost-Full Node



Figure 5 – Dequeue Timing in a Full Node

## Controller Organization (qq\_control)

Figure 6 shows the controller organization. It consists of a FSM, and address counter that produces addresses for the write port of the BRAM, and an incrementer that produces addresses for the read port of the BRAM as shown in the table.



Figure 6 – Controller Organization

## Controller Function (qq\_fsm)

Figure 7 shows an ASM Diagram of the controller function. ASM Diagrams use a flowchart notation to describe the function of a state machine. Each state in the diagram consists of a rectangle “state box” that is labeled with the state name and contains any outputs that are set unconditionally during the state. Truncated-diamond “decision boxes” specify transitions, while oval “conditional boxes” specify outputs that are set in a state based on the conditions in the preceding decision box.

The state machine includes an initialization sequence that sets the BRAM to contain all empty values.



Figure 7 – ASM Diagram of Control FSM

## Appendix – Replace (Enqueue-Dequeue) Operation

A replace operation removes the first item in the priority queue and searches for the new item “fits” between the existing items in the queue. It operates in a fashion similar to dequeue in that it starts by moving items to the left until key of the new item is greater than the item being moved. At that point the new item is placed into the “hole” left by the moved item. If the end of a node is reached, the last item in the node is replaced by the first item in its right neighbor and the replace operation is then applied to the right neighbor.



Figure 8 – Replace Timing within a Node

Figure 9 – Replace Timing (forwarding to Right Neighbor)

Figure 10 shows the organization of the QuickQ modfied to support Replace operations. A replace operation simultaneously removes the heat of the priority queue and inserts a new element into the queue. It operates in a way similar to a dequeue operation. Starting with a write address wraddr=0, it compares the item to be inserted with the current element. If the new item is greater than the current element, then the new element is written in its place and the operation is complete. Otherwise the current element is overwritten with its right neighbor, the write address is incremented, and the process repeats. When the write address reaches DEPTH-1, a separate state reads the value from the node’s right neighbor and compares it to the new element. If the new element is less than the element read from the neighbor, then the new element is written into the last element of the node and the operation is complete. Otherwise, the new element is passed to the right neighbor and replace operation is started on the right neighbor.



Figure 10 – Modified qq\_node Organization

Figure 11 – Modfied ASM Chart (to be added)