A. Arbiter implementation

In order to use the priority scores determined by our scheme to allocate the data transfer budget to the highest scoring logical qubits, we need a way to identify the k logical qubits with the highest priority scores out of the n that are competing to be decoded. An arbiter circuit that implements the top-k argmax of the n priority scores based on arithmetic comparison of binary priority scores and multiplexing of qubit indexes would incur a significant area cost for the budget current integration density of superconducting cells allows, especially as the number of logical qubits n and number of supported priority levels increases. Fortunately, designs like this one that consist of comparisons can be implemented efficiently using temporal computing, a logic scheme where information is represented as the time of arrival of signals, allowing priority scores of any number of bits to be supported by a single wire. Specifically, we design our arbiter using the paradigm of race logic [?], for which a mapping of it's primitives to superconducting cells was provided in [?].

For the comparator module we base the arbiter on, we make use of three race logic primitives. The first two primitives are First Arrival (FA) and Last Arrival (LA), which perform the equivalent of min and max operations of the temporally encoded values. FA will fire upon the first pulse it receives in one of it's input ports and reset upon receiving a second pulse on it's other input port, whereas LA both fires and resets upon receiving the second pulse. In SFQ they are implemented with an inverted Muller C cell and a regular C cell respectively. We have tuned these cells to have very similar delays, so differences in their propagation speed do not lead to the need for synchronizing signals inside the arbiter. The third primitive is Inhibit (INH), that has two input ports a and b and propagates a pulse received on b only if it arrives before a pulse on a does. Typically Inhibit is implemented using a clocked inverter cell, however because we need only a boolean signal of whether b arrived before a and not a temporal signal of when b arrived for our purposes, we substitute it for the cheaper DRO cell, and assign a to the DRO's clock port and b to it's data input port. In the case a and b arrive very close to each other, they will both represent the same priority score and their comparison will be a tie, in which case either the DRO firing or not are both valid for our purposes. With these primitives we can design the comparator module of size 2 that will be the building block of our arbiter.

The comparator module's ports consist of four pairs, each pair consisting of a data port and a select port. It provides the functionality for both a forward pass that uses the data ports to sort the input priority scores and a backward pass that uses the select ports to route signals to go down the same path data signals did in the forward pass. The forward pass connects the x_0 and x_1 input wires which encode priority scores to FA, LA and INH cells. The outputs of FA and LA connect to the x_{min} and x_{min} output ports of the comparator, propagating the earliest and latest pulses from the inputs respectively. The output of the INH cell goes to the data input of two DROC cells, to prepare them for the backward pass. In the backward pass input ports sel_{min} and sel_{max} are connected to the clock ports of the DROC cells. The outputs of the DROC pair are merged such that a pulse from sel_{min} is routed to sel_0 if the DROC pair stores a pulse from the INH, meaning that signal x_0 arrived before x_1, and to sel_1 otherwise. Similarly, a signal from sel_{max} goes to sel_1 if x_0 arrived before x_1 and to sel_0 otherwise.

Fig?? shows an example of the comparator module's operation with the routes the various signals took. Fig ?? shows the gate level implementation of the comparator module. As it can be seen, only a handful of gates are needed compared to what a bit parallel arithmetic implementation would require.

To build the full arbiter that provides the functionality of top-k argmax, we arrange comparator modules in the configuration of a bitonic sorter. Fig ?? shows the arrangement for a 4 element bitonic sorter. Also shown is the path of the latest arriving encoded priority score through the sorter and the path a select signal given at the highest position of the sorter's final layer takes through the comparator modules to fire at the select port of the first layer, denoting the argmax of the data input pulses.

Scaling the bitonic sorter configuration to the number n of priority scores we need to evaluate would work for a top-k argmax if we first send the temporally encoded priority scores for all n qubits as input pulses for the forward pass and after that fire pulses at the select ports of the final layer corresponding to the k largest positions. These would get routed to the select ports of the first layer corresponding to the k largest scores entered in the data ports, providing a binary flag representation of the argmax that can be used without additional decoding. However, a large part of this structure would be wasted on comparators sorting signals that cannot be in the top-k, and thus are not needed. After removing the unnecessary comparators we are left with a structure where the data signals are split in groups of length k and each group goes through a full bitonic sorter of size k. The number of data signals is then cut in half by selecting the k largest outputs from each pair of k-sized sorters. For every comparator that performs this reduction only the max pair of data/select ports is connected to the next layer, whereas the x_{min} and sel_{min} are left unconnected. The $\frac{n}{2}$ signals left go through a similar sort and reduce process, although subsequent blocks of size k sorters require less comparators. After the final reduction k signals will be left, from which the select signals for the top-k will originate.