0.1 Temporal arbiter implementation

Given priority scores from all the candidate qubits, an arbiter circuit is needed to select the ones with the highest priority, in order for their syndromes to be routed to the allocated communication infrastructure for decoding.

The desired functionality is in essence a top-k argmax, which will give the indexes of the k logical qubits that produce the highest scores and thus are in most urgent need of complex correction. This would be relatively straightforward to implement the traditional way with arithmetic comparators and index multiplexers. However, this implementation carries a significant area cost for the limited budget allowed by the integration density of current superconducting fabrication processes.

To reduce the area and power requirements, as well as limit the latency, we instead use a race logic based design for the arbiter. First, the priority scores of each candidate qubit are translated into temporal signals. A priority score of $x \in [0, 5]$ gets encoded into a pulse arriving at time $T_x = x\Delta_t$. The encoded scores are then passed through a temporal bitonic sorter. Such a circuit is very well suited for temporal implementation, as only two functional cells are needed to sort two signals, leading to hardware efficiency impossible by other means. Because we need the top-k argmax of the signals, not just their max, a sorter by itself is insufficient. Instead, the signals passing through the bitonic sorter are also used to control a small routing circuitry, which will lead select signals $return_i, i \in [0, k-1]$ for each of the top k maximum outputs of the sorter through the same path the corresponding data signals took to reach that output. This ends in a pulse being emitted at port $sel_i, i \in [0, n-1]$, for exactly those qubits whose priority x_i is in the top-k largest. This approach avoids complicated index multiplexing and decoding circuitry and produces a directly usable multi-hot qubit selection.

The basic element of the temporal arbiter is the comparator module, shown in fig?. Let $T(x)$ denote the time a pulse arrives at port or wire x. No more than one pulse will reach any port during an execution of our temporal circuit, so we only need to represent the time for that pulse. In cases where no pulse reaches x during an execution, we assign $T(x) = +\infty$.

Two input temporal signals at ports x_a and x_b are sorted, such that the first output x_{max} will re-emit the larger of the two encoded priorities, meaning the one for which a pulse arrived the latest, such that $T(x_{max}) = \max T(x_a), T(x_b) + \tau_c$, where τ_x a constant delay of the sorting. The second x_{min} emits the smaller of x_a, x_b. In cases of ties the outputs are equivalent.

For the backward pass, the comparator module routes the select signal $return_{max}$ to the port sel_a if the x_a arrived after x_b and to port sel_b if it arrived before it or at the same time. A pulse appearing at $return_{min}$ will be routed the opposite way. We consider the encoding for the backward pass to be simply binary instead of temporal, so we don't care about the arrival times of $return$ signals as long as they occur after the inputs x_a, x_b.

This gives us $sel_a = \begin{cases} return_{max} & T(x_a) > T(x_b) \\ return_{min} & otherwise \end{cases}$. sel_b works similarly. By first passing encoded priorities x_a and x_b through the comparator and then sending a pulse in $return_{max}$, the *sel* outputs will encode the argmax of the two priorities.

To build the complete top-k module, we can combine these comparator modules in an arrangement of a bitonic sorter, with x_{max} and x_{min} ports connected to the input x_a or x_b ports of the following comparators and *sel* ports connected to the *return* ports of the preceding comparators in the same fashion. First we pass pulses at at times encoding the priority scores of the n logical qubits to the n input ports of the sorter circuit. After the signals have propagated through the sorter, pulses are issued at the *return* ports corresponding to the top k largest outputs of the bitonic sorter. These pulses will follow the path the perspective priority signal took through the comparators to get there, thus only the k x ports at the inputs of the sorter will receive pulses at their related *sel* ports. This selection bitvector can then be used directly to route the syndromes of the correct qubits to the communication module for external decoding.

However, we can see that a large portion of the sorter is dedicating to sorting signals that do not belong to the top k, which is unnecessary. To improve on hardware efficiency, we sparsify the bitonic sorter. First, we split the n inputs in groups of size k and pass each group through a full bitonic sorter of size k, implemented as described above. We can then reduce the number of signals by half by selecting only the k largest outputs from each pair of k-sized sorters. This is done the same way that selecting the top half of values is done in bitonic sorting, by applying a comparator to the i-th output of the first sorter and the $(k-i)$-th one of the second sorter. We then pass the $\frac{n}{2}$ signals through a similar sort and reduce process. We repeat this until only k signals are left, from which the *return* signals will originate. This reduces the number of comparators from $\frac{n\log_2(n)(\log_2(n)+1)}{4}$ to $(\frac{k\log_2(k)(\log_2(k)+1)}{4}+\frac{k}{2})\frac{n}{k}+(\frac{n}{k}-2)\frac{k\log_2(k)+k}{2}$, and the number of comparators a signal goes through from $\frac{\log_2(n)(\log_2(n)+1)}{2}$ to $\frac{\log_2(k)(\log_2(k)-1)}{2}+\log_2(\frac{n}{k})(\log_2(k)+1)$, which minimizes latency.

Since we only keep the x_{max} output from the comparators that implement the $2k$ to k reduction between sort steps, and only need to route the $return_{max}$ signal to a *sel* port, we can use a smaller version of the comparator module for these cases, saving additional resources.

0.2 Comparator circuit

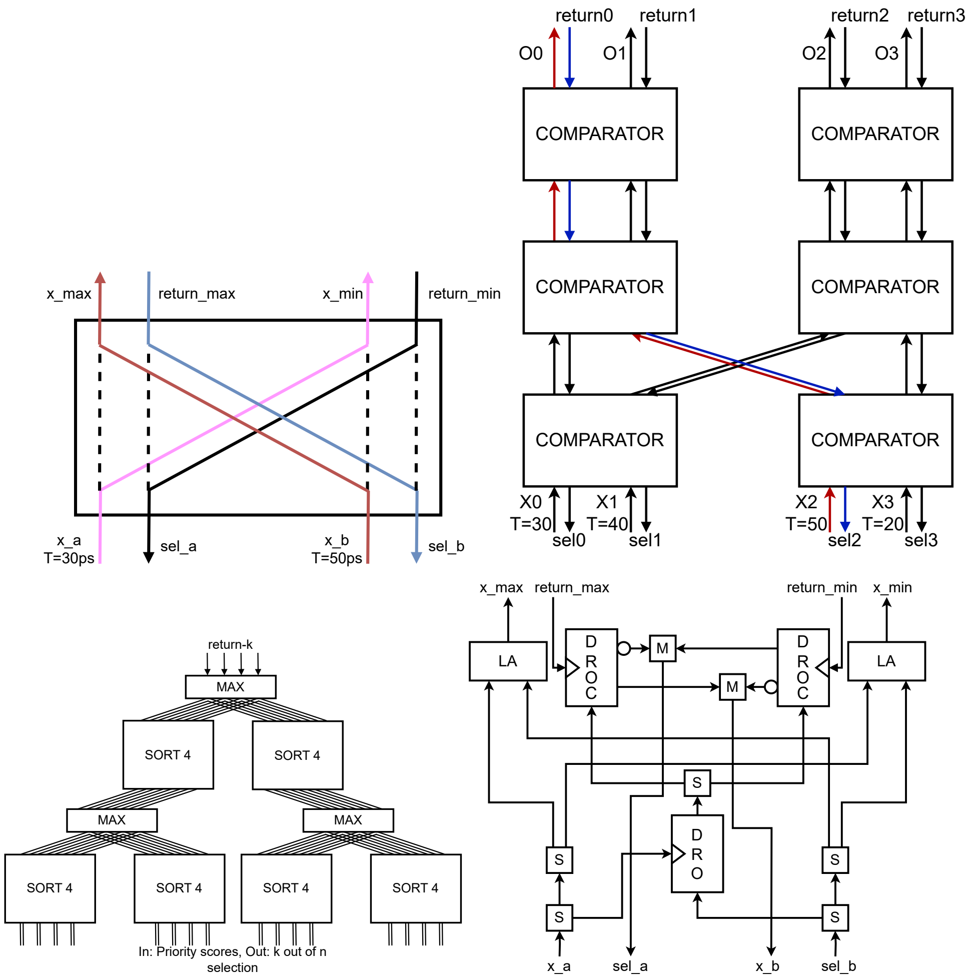
To implement the forward pass of the comparator, we only need two cells, one for the race logic primitive *First Arrival* (FA), which gives as the min of two signals, and another one for *Last Arrival* (LA), which gives the max. These can be implemented with a C element for LA and an inverted C element for FA. Both of theses cells are cheap area-wise and we have tuned them to have similar delays, so differences in their propagation delay do not cause the order of signals' arrival to change for the length of the sorter network. Thanks to this, synchronization to re-align the signals to their encoded value is not needed for distances of encoded temporal values $\Delta_t \sim 20ps$ we consider.

n	k	latency (ps)	area (JJ)
64	2	348	6696
128	4	556	26928
256	8	816	90144

To implement the backrouting part of the comparator module, we need a) to detect whether a or b arrived first and b) to demultiplex the *return* signals based on that. The first case corresponds to an inhibit cell in temporal logic. $\mathit{Inhibit}(a, b)$ or $a \dashv b$ allows signal b to pass to the output only if a has not previously arrived to block it. In superconducting circuits this operation is typically instanciated via a clocked inverter cell. However, because in our case we don't need the temporal information of when b arrived, but only need the boolean information of whether it came second, we instead use the cheaper Destructive Readout(DRO) cell, which is the SFQ equivalent to the CMOS DFF. This choice also makes it easier to reset the cell with an external signal after the operation is done. Signal a is mapped to the clock port of the DRO and b to the data input port. If a arrives before b, no pulse is stored in the DRO and thus no pulse is produced at the output. If instead the pulse at b arrives first at the data port of the DRO, it stores a pulse in the superconducting loop that is makes it to the DRO output upon a's arrival. In the case the two signals a and b are close enough in time to be in the undefined behavior region of the DRO, they encode the same discrete temporal value, thus their comparison results in a tie and either the presence or absence of a pulse at the DRO output is a valid result for our purposes. This is important for the correct operation of the system.

To demultiplex $return_{max}$, a Complementary output Destructive Readout (DROC) cell is used. This cell produces a pulse at it's first output upon clock arrival if it has not received a pulse in the data input port since the last clock pulse, and produces a pulse at the second output otherwise. By plugging the result of the $a \dashv b$ DRO in the data port of the DROC and the $return_{max}$ wire to it's clock port, as well as connecting the first and second DROC outputs to the comparator module's sel_b and sel_a, a pulse at $return_{max}$ gets routed to sel_a if a was the last signal of the two and to sel_b in the other case. Because we might need to select both of the comparator's operands for the top-k, we also need a second DROC with it's outputs swapped to route $return_{min}$ to the select signal of the input that came first. We use a splitter to pass $a \dashv b$ to the data input of both DROCs and two merger cells to combine their outputs for sel_a and sel_b. Four more splitters are used to send inputs a and b to the FA, LA and DRO cells. The comparator module only requires a handful of gates, significantly less than a bit-parallel implementation would, which makes it ideal for our resource constrained environment.

The arbiter circuit has been implemented in the PyLse pulse-transfer simulator for various sizes. Measurements for area and latency are shown in table x and fig y.


return0
return1
return2
return3
O0
O1
O2
O3
COMPARATOR
COMPARATOR
COMPARATOR
COMPARATOR
COMPARATOR
COMPARATOR
X0
T=30
sel0
X1
T=40
sel1
X2
T=50
sel2
X3
T=20
sel3
x_max
return_max
x_min
return_min
x_a
T=30ps
sel_a
x_b
T=50ps
sel_b
return-k
MAX
SORT 4
SORT 4
MAX
MAX
SORT 4
SORT 4
SORT 4
SORT 4
In: Priority scores, Out: k out of n selection
x_max
return_max
return_min
x_min
LA
D R O C
M
M
D R O C
LA
S
S
D R O
S
S
S
x_a
sel_a
x_b
sel_b