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 $T_x = x\Delta_t$, where Δ_t is a small constant delay. The encoded scores are then passed through a temporal bitonic sorter. The function of the temporal sorter is to route the input signals such that they appear in it's outputs in order from earlier to later arriving pulse. So if x_k is the k-th largest priority score and T_{sort} the constant latency of the sorter, the k-th output wire of the sorter will produce a pulse at time $x_k\Delta_t + T_{sort}$. Sorters are 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. Of course, 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 score x_i is in the top-k largest priority scores. This approach avoids complicated index multiplexing and decoding circuitry and produces directly usable binary flags of 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 $T(x) = +\infty$.

The comparator module sorts two input temporal signals at ports x_a and x_b , such that the 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_c a constant denoting the comparator's latency. The output 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 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. The encoding for the backward pass is simply binary rather than 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. The *sel* outputs encode the argmax of the two input priority scores.

To build the complete top-k argmax, comparator modules are arranged into a bitonic sorter, with x_{max} and x_{min} ports connected to x_a or x_b ports of the next layer's comparators and *sel* ports connected to *return* ports of preceding layer. First we pass pulses encoding the priority scores of n logical qubits to the n data input ports of the sorter circuit. After they have propagated through the sorter, the *return* ports corresponding to the top k outputs of the bitonic sorter are activated with a pulse. These pulses will follow the path their perspective data signal took through the comparators, thus only k ports at the first layer of the sorter will receive pulses at their related *sel* ports. The bitvector of these *sel* signals can then be used directly to route the syndromes of the selected logical qubits for external decoding.

However, we can see that a large portion of the sorter is sorting signals that can not belong to the top k, which is unnecessary. To improve on hardware efficiency, we can 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. We can then reduce the number of inputs by half by selecting 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.

0.2 Comparator circuit

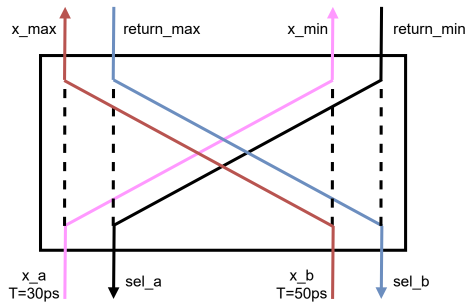
The first two race logic primitives we need for the arbiter are *First Arrival* (FA) and *Last Arrival* (LA), which correspond to min and max operations on temporal signals respectively. These can be implemented with a C element for LA and an inverted C element for FA. Both of theses cells are cheap area-wise and we have tuned them to have very similar delays, so differences in their propagation delay do not cause the order of signals' arrival to change for the

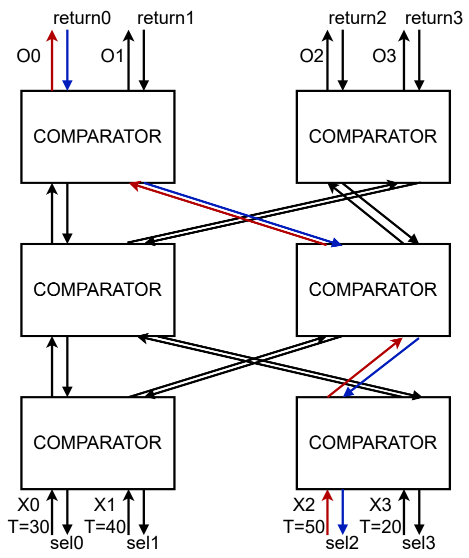
n	k	latency (ps)	area (JJ)
64	2	379	5580
128	4	588	23004
256	8	847	77760

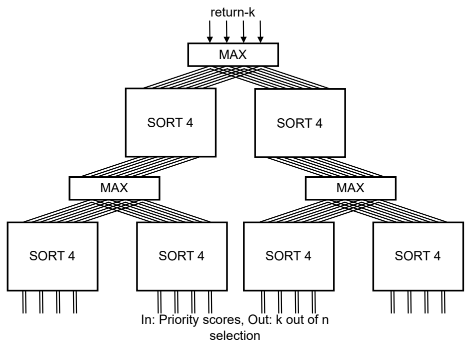
length of the sorter network. Thanks to this, synchronization to re-align the signals to their encoded value is not needed for the input sizes we consider.

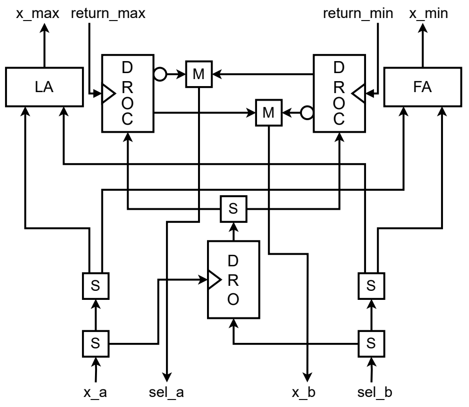
For the backrouting part of the comparator module, we need *a*) to detect whether a or b arrived first and *b*) to demultiplex the *return* signals based on that. The first case corresponds to an inhibit cell in temporal logic. $Inhibit(a,b)$ or $a \dashv b$ allows signal b to pass to the output only if a has not previously arrived, in which case it gets blocked. Typically in superconducting circuits clocked inverters are used as inhibit cells. Since in our case we only need to know whether input b came first or second, and not the temporal information of when it arrived, we can use the cheaper Destructive Readout(DRO) cell. Input 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 propagates to the DRO output upon a's arrival. In the case the two signals a and b are close to each other, such that they encode the same discrete temporal value, their comparison is a tie and either the presence or absence of a pulse at the DRO output is a valid result. This is important for the correct operation of the system.

To demultiplex $return_{max}$, a Complementary output Destructive Readout (DROC) cell is used. Upon clock arrival, this cell produces a pulse at either it's first output if it has not received a pulse in the data input port, or at it's second output otherwise. Connecting the output 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 allows it's usage in our resource constrained environment.


x_max
return_max
x_min
return_min
x_a
T=30ps
sel_a
x_b
T=50ps
sel_b


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


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
FA
S
D R O
S
S
S
S
x_a
sel_a
x_b
sel_b