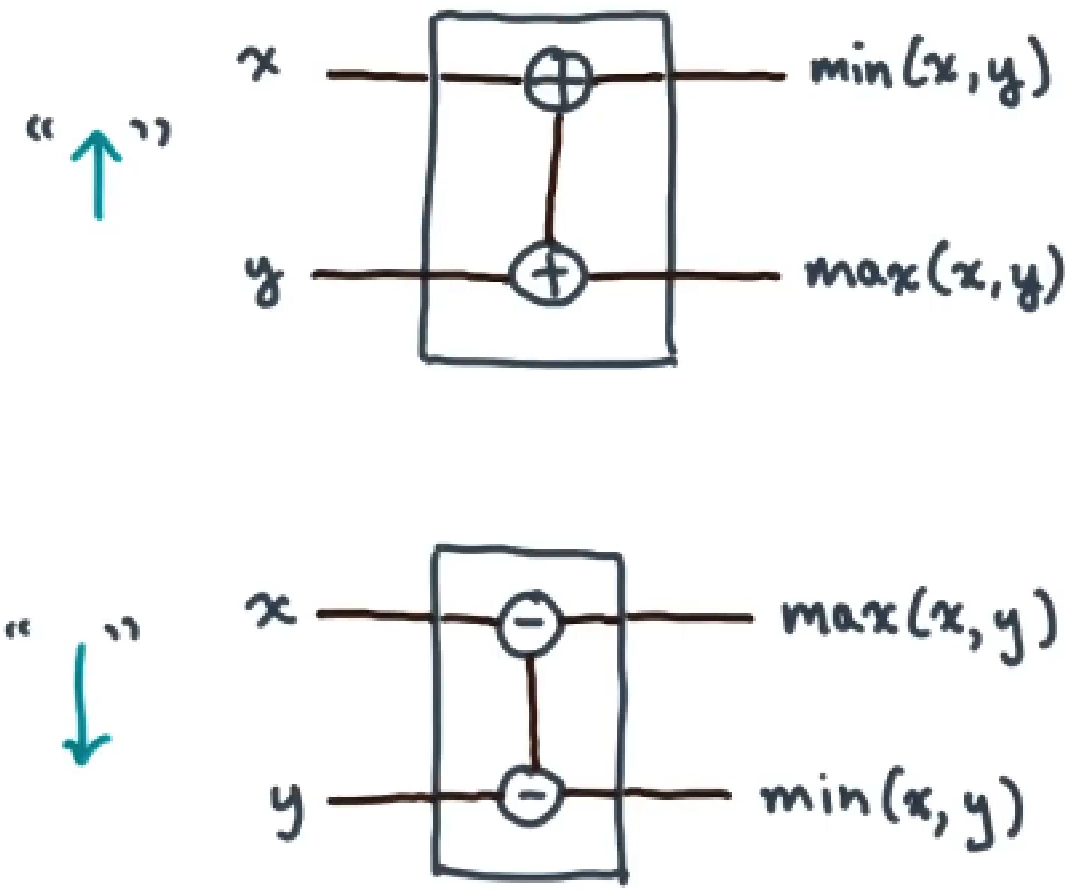
Comparison-based Sorting

Introduction

1. Covers parallel algorithms for sorting in the dynamic multi-threading model
   - Includes the idea of a sorting network

Comparator Networks

1. Sorting network: Fixed circuit that sorts its inputs using a special gate called a comparator
   - Increasing/plus comparator puts the smaller of its inputs on the top wire
   - Decreasing/minus comparator puts the larger of its inputs on the top wire
2. Similar to work and span, we can analyze a circuit by the number of operations it performs and the critical path length

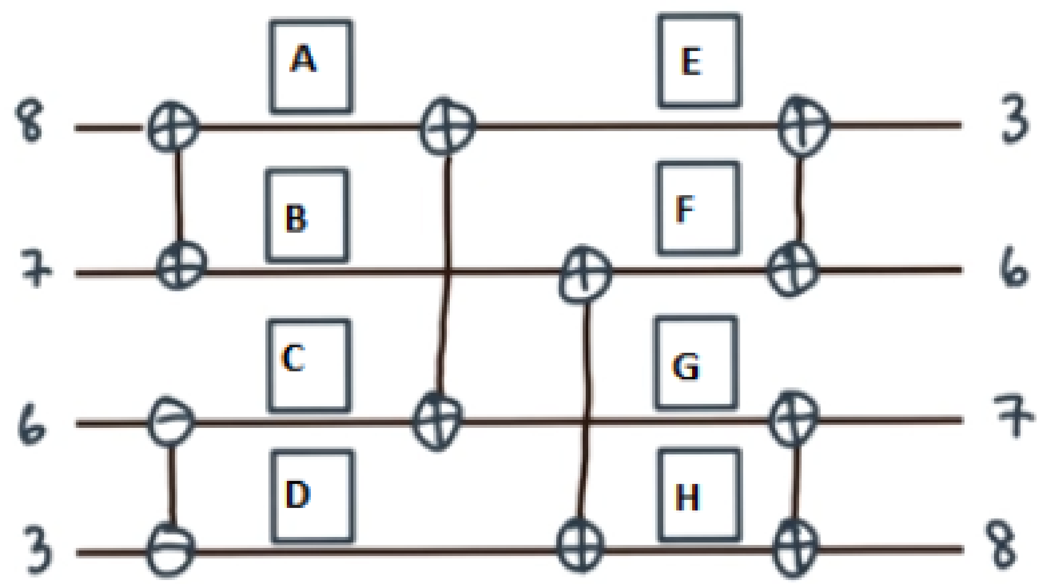


Comparator Networks

Sort 4 Values

1. Fill in the boxes with the values based on the wiring
   - A: 7
   - B: 8

- C: 6
- D: 3
- E: 6
- F: 3
- G: 7
- H: 8



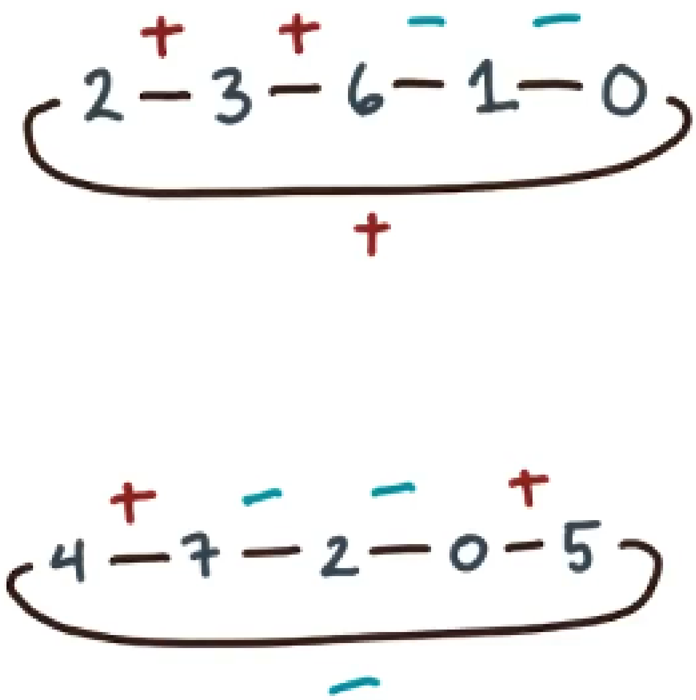
Comparator Quiz

Bitonic Sequences

1. Bitonic sequence: Sequence of numbers which is monotonically increasing initially, then monotonically decreasing past some point
   - (a0, a1, . . . an-1) is bitonic if:
     - a0 <= a1 <= . . . <= ai AND
     - ai+1 >= . . . >= an-1
   - This is also true if the above condition holds after some circular shift

Bitonic Sequences Quiz

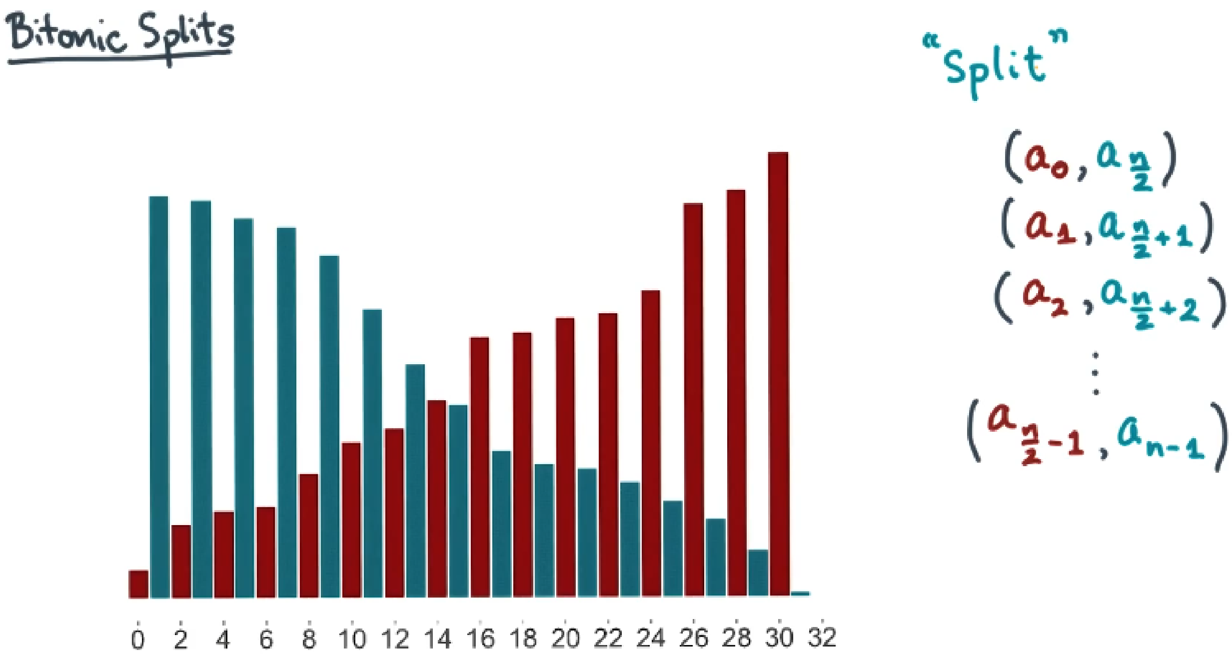
1. Which sequence is bitonic?
   - 2, 3, 6, 1, 0 (yes)
   - 4, 7, 2, 0, 5 (no)



Bitonic Sequences Quiz

Bitonic Splits

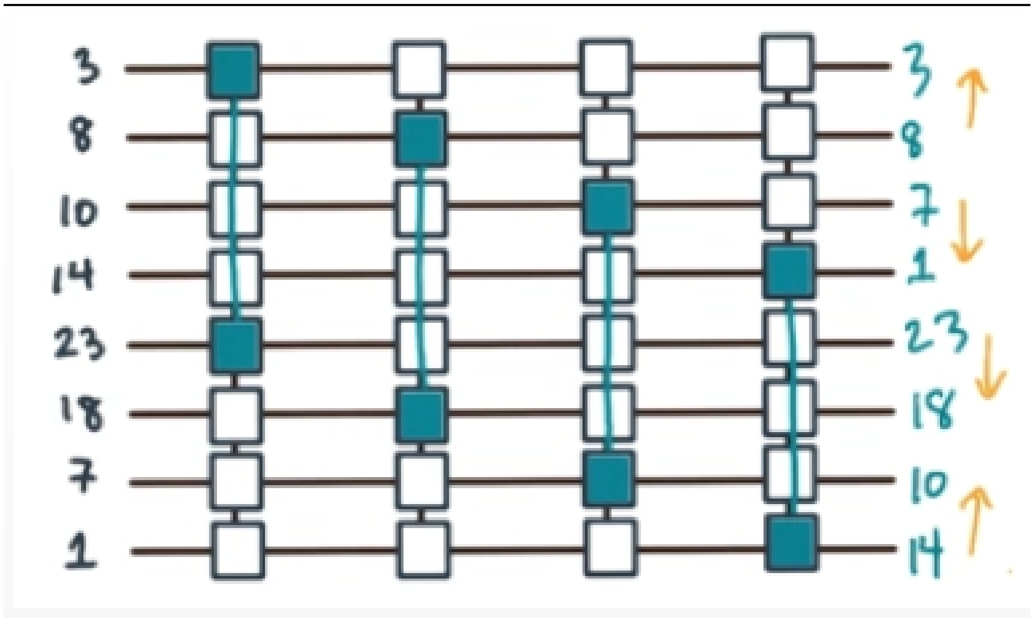
1. Bitonic split: Pair the elements of a bitonic input sequence and then apply mins/maxes to pairs
    - Results in two bitonic subsequences
    - All elements of the max sequence are greater than or equal to all the elements of the minimum sequence



Bitonic Split

Bitonic Split Quiz

1. Connect the comparators to implement a bitonic split



Bitonic Split Quiz

Bitonic Splits: A Parallel Scheme

1. The following algorithm implements a parallel bitonic split:
    - Assume 2 | n

```
bitonicSplit(A[0:n-1)
{
    parfor i <- 0 to n/2-1 do
        a <- A[i]
        b <- A[i+n/2]
        A[i] <- min(a, b)
        A[i+n/2] <- max(a, b)
}
```

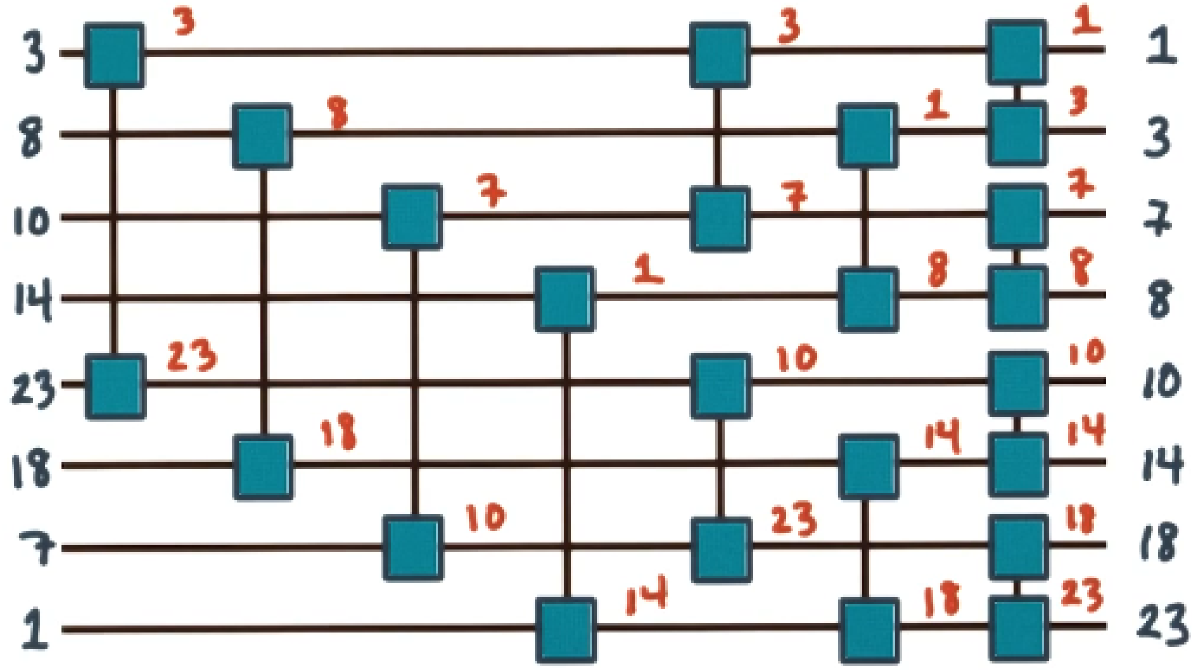
Bitonic Merge

1. Bitonic split provides a simple divide-and-conquer framework for sorting
    - Recursively performing bitonic splits on each pair until no pairs remain will sort the list
    - Assume 2 | n
    - Because all the elements of one subsequence are less than or equal to all the elements of the other subsequence, we can spawn one of the merges

```
bitonicMerge(A[0:n-1])
{
    if n >= 2 then
        bitonicSplit(A[:])
        bitonicMerge(A[0:n/2-1]) // spawn
        bitonicMerge(A[n/2:n-1])
```

Bitonic Merge Networks

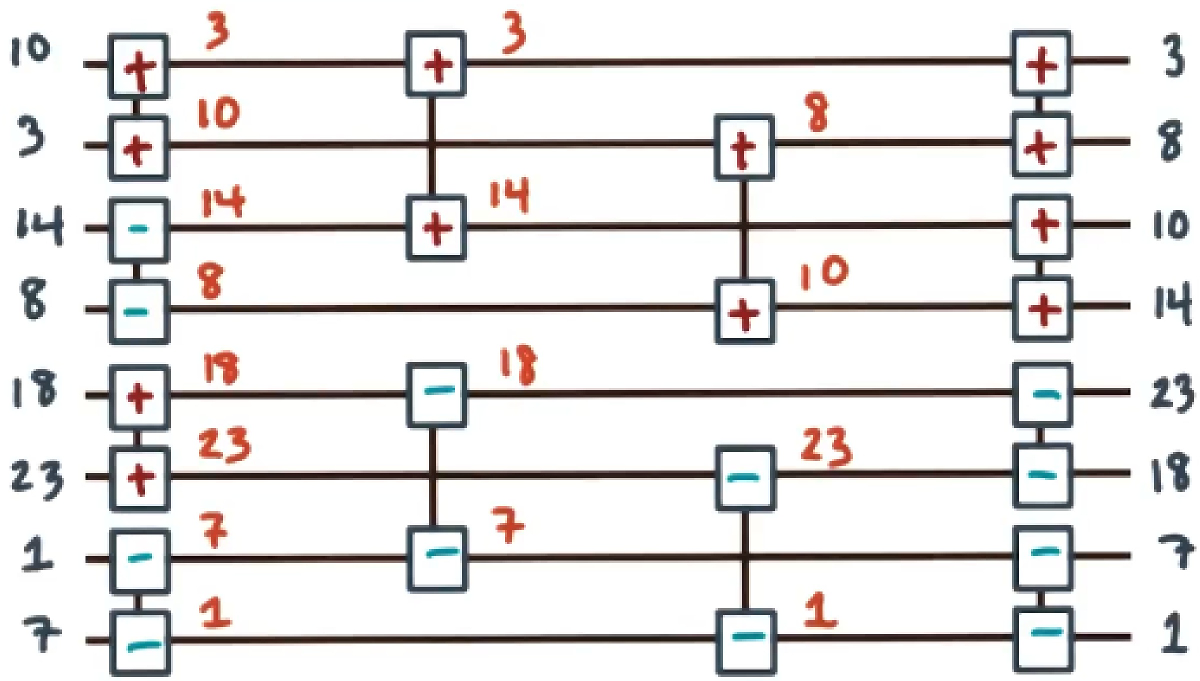
1. Connect the comparators to create a bitonic merge network



Bitonic Merge Quiz

Generate a Bitonic Sequence

1. Connect the comparators to generate a bitonic sequence



Bitonic Sequence Quiz

Bitonic Sort

1. First, generate a bitonic sequence
    - Assume 2 | n

```
genBitonic(A[0:n-1])
{
    if n >= 2 then
        spawn genBitonic(A[0:n/2-1])
        genBitonic(A[n/2:n-1])
        sync
        spawn bitonicMerge+(A[0:n/2-1])
        bitonicMerge-(A[n/2:n-1])
}
```

2. To sort, do the following:

```
bitonicSort(A[0:n-1])
{
    genBitonic(A[:])
    bitonicMerge+(A[:])
}
```

3. The work and span are:
    - W(n) = O(n * (log(n)^2))
    - D(n) = O(log(n)^2)
    - The span is polylogarithmic which is good, but this algorithm is not work-optimal
        - Comparison-based sorts can be completed in O(n * log(n))

Conclusion

1. Bitonic sort has a fixed, regular parallel structure which lends itself to implementation on an FPGA
    - Also maps well to fixed data-parallel hardware (SIMD or graphics co-processors (GPUs))
2. Downside is that it's not work-optimal, even when restricted to the class of comparison-based algorithms
    - Have to make an engineering tradeoff based on the platform and scale you are interested in