CMPSCI 691AD – General Purpose Computation on the GPU

Spring 2009

Lecture 15: Sorting

Rui Wang

Topics

- Radix Sort
- Bitonic Sort
  - Bitonic Sequence
  - Bitonic Split
  - Bitonic Merge
  - Complexity
- Odd-even Sort

Radix Sort

- Assume all elements have the same number (n) of bits
- Perform 0-1 split n times, starting from the lowest bit
  - Split is computed using scan.
  - Important: order within group preserved.
- Example:

[5	7	3	1	4	2	7	2]
[101	111	011	001	100	010	111	010]

- Works well for integer numbers (32-bit requires 32 splits)

Radix Sort

- What about FP numbers?
  - IEEE 754 Standard
  - For positive FP numbers, radix sort still works.

```
float number;
*(unsigned int*)(&number);
```

  - Negative FP numbers are more complicated.

Bitonic Sort

- Bitonic Sequence: a sequence of elements

$$\langle a_{0,} a_{1,} a_{2,} \ldots a_{n-1} \rangle$$

that looks like this:

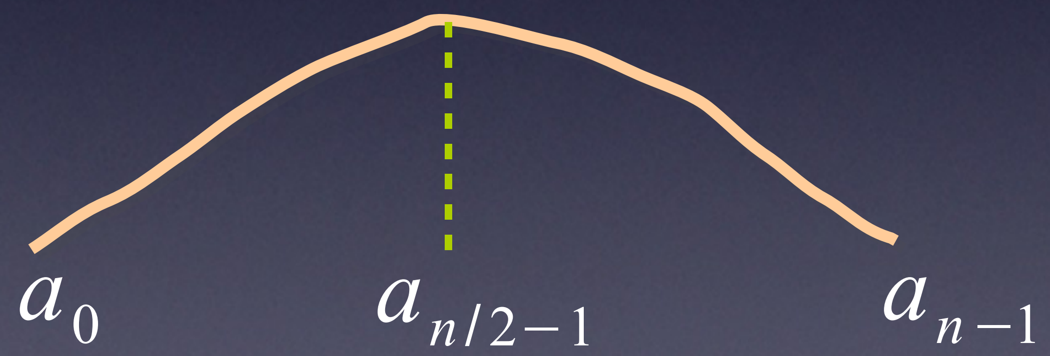
including the degenerate and cyclic shift cases.

Bitonic Sort

- Bitonic Sequence: here we will focus on bitonic sequences

$$\langle a_{0,} a_{1,} a_{2,} \ldots a_{n-1} \rangle$$

where the <u>left half</u> is non-decreasing, and the <u>right half</u> is non-increasing; or the reverse.



Bitonic Sort

- Bitonic Split: compare each element in the first half to its corresponding element in the second half, and swap in increasing order.

$$a_i = min\left(a_i, a_{i+n/2}\right)$$

$$a_{i+n/2} = max\left(a_i, a_{i+n/2}\right)$$

$$i \in [0, \frac{n}{2} - 1]$$

Bitonic Sort

- **Bitonic Split:** compare each element in the first half to its corresponding element in the second half, and swap in increasing order.

$$a_i = min\left(a_i, a_{i+n/2}\right)$$

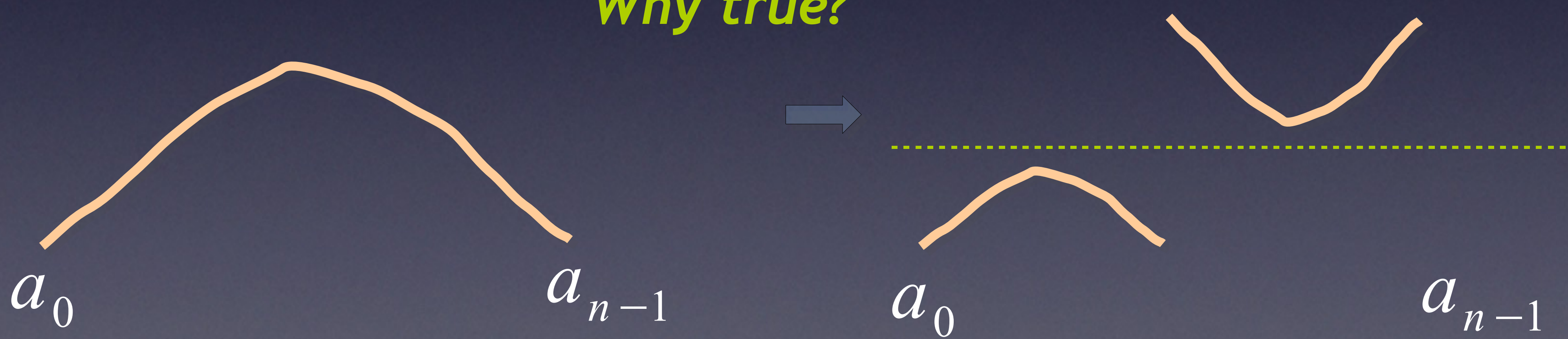
$$a_{i+n/2} = max\left(a_i, a_{i+n/2}\right)$$

$$i \in [0, \frac{n}{2} - 1]$$

Bitonic Sort

- Bitonic Split: compare each element in the first half to its corresponding element in the second half, and swap in increasing order.
- This results in two sub-sequences that are both bitonic, and the first sub-sequence is smaller than the second.

Why true?



Bitonic Sort

- Bitonic Split: compare each element in the first half to its corresponding element in the second half, and swap in increasing order.
- This results in two sub-sequences that are both bitonic, and the first sub-sequence is smaller than the second.

Bitonic Sort

- Bitonic Merge: Repeatedly apply bitonic split on each sub-sequence, until the sub-sequence is only 1 element wide.

Now we have a method to sort a bitonic sequence!

Bitonic Sort

- Bitonic Merge: Repeatedly apply bitonic split on each sub-sequence, until the sub-sequence is only 1 element wide.
- Example:

  [3 8 10 14 20 18 9 0]

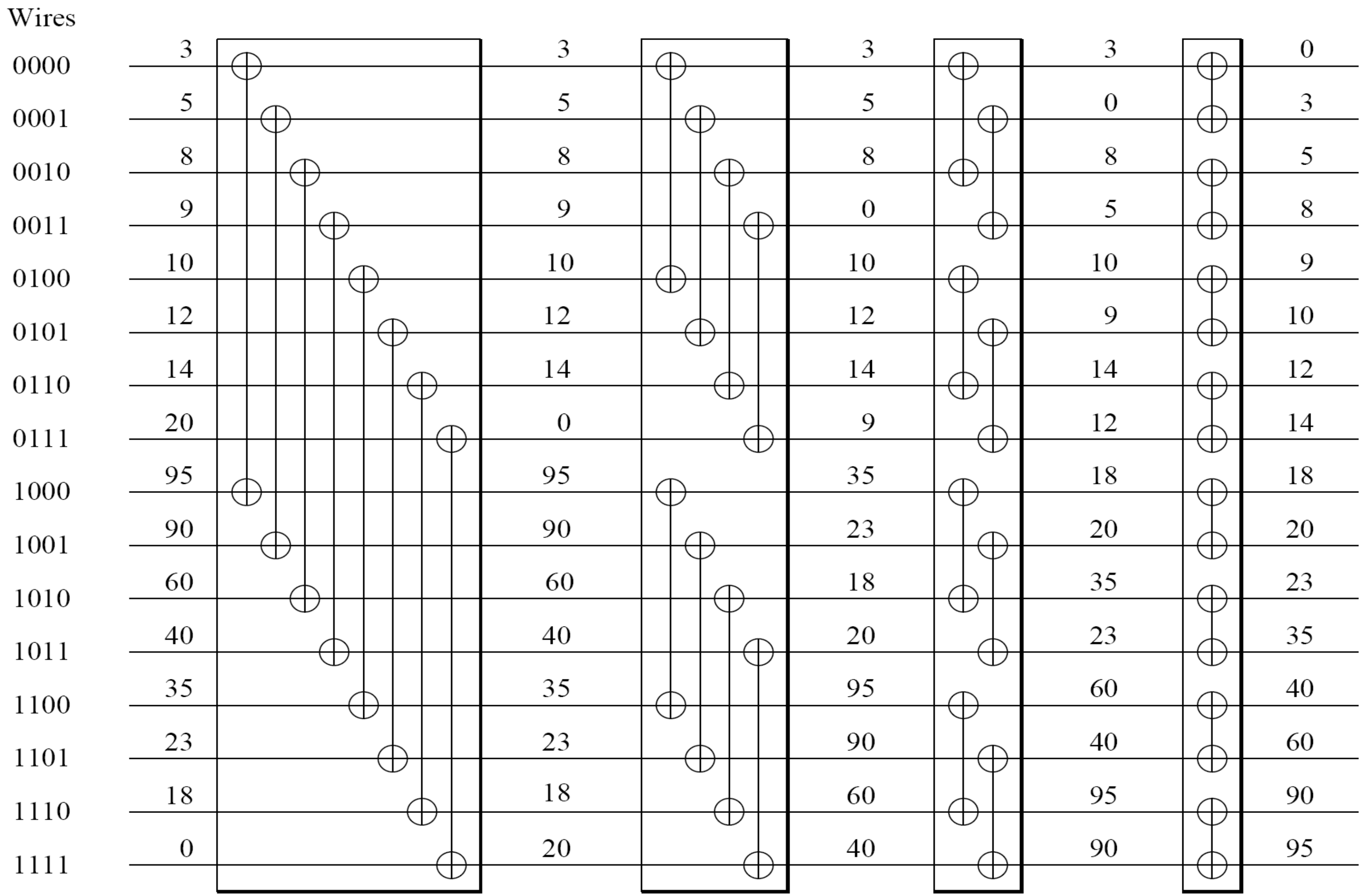


Figure 9.6 A bitonic merging network for $n = 16$. The input wires are numbered $0, 1 \ldots, n-1$, and the binary representation of these numbers is shown. Each column of comparators is drawn separately; the entire figure represents a $\oplus$BM[16] bitonic merging network. The network takes a bitonic sequence and outputs it in sorted order.

Bitonic Sort

- Bitonic Merge: Repeatedly apply bitonic split on each sub-sequence, until the sub-sequence is only 1 element wide.
- Example:

  [3 8 10 14 20 18 9 0]

- Complexity?

  K processors sort k elements in parallel:

Bitonic Sort

- Bitonic Merge: Repeatedly apply bitonic split on each sub-sequence, until the sub-sequence is only 1 element wide.
- Example:

  [3 8 10 14 20 18 9 0]

- Complexity?

  K processors sort k elements in parallel: $log(k)$

  What about sequential (1 processor)? Compare with standard merge sort.

Bitonic Sort

- Bitonic Merge: Repeatedly apply bitonic split on each sub-sequence, until the sub-sequence is only 1 element wide.
- Example:

  [3 8 10 14 20 18 9 0]

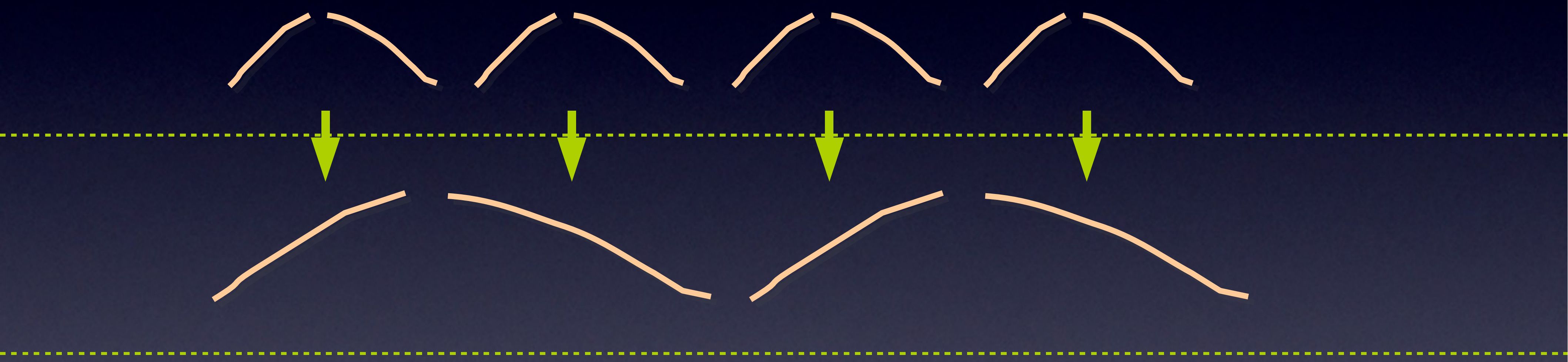
- Bitonic merge is similar to qsplit in quicksort, but has the bitonic constraints.
- What about a completely unsorted sequence?
  - Build bitonic sequences from bottom up
  - You will see how this is similar to standard merge sort.

Bitonic Sort

- Bitonic Sort:

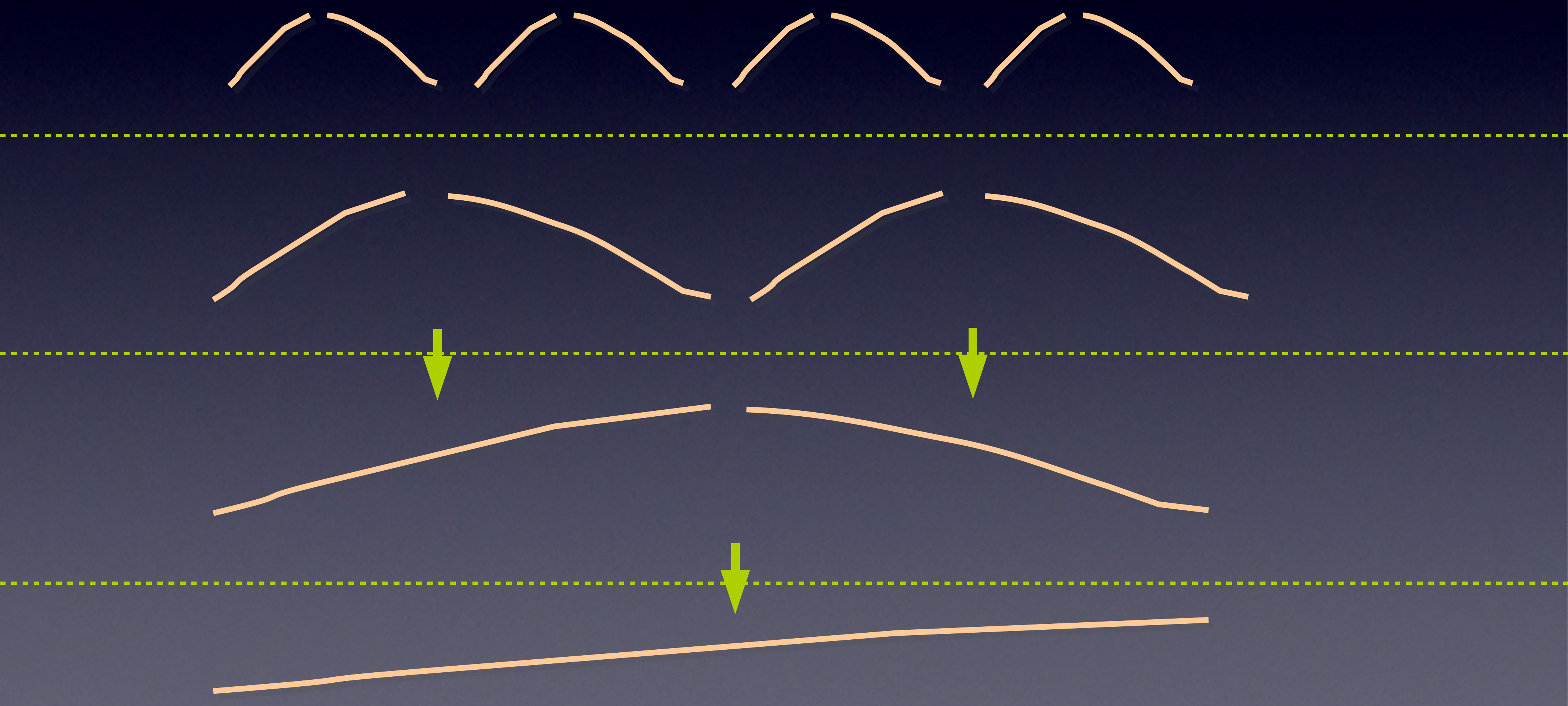
Bitonic Sort

- Bitonic Sort:



Bitonic Sort

- Bitonic Sort:



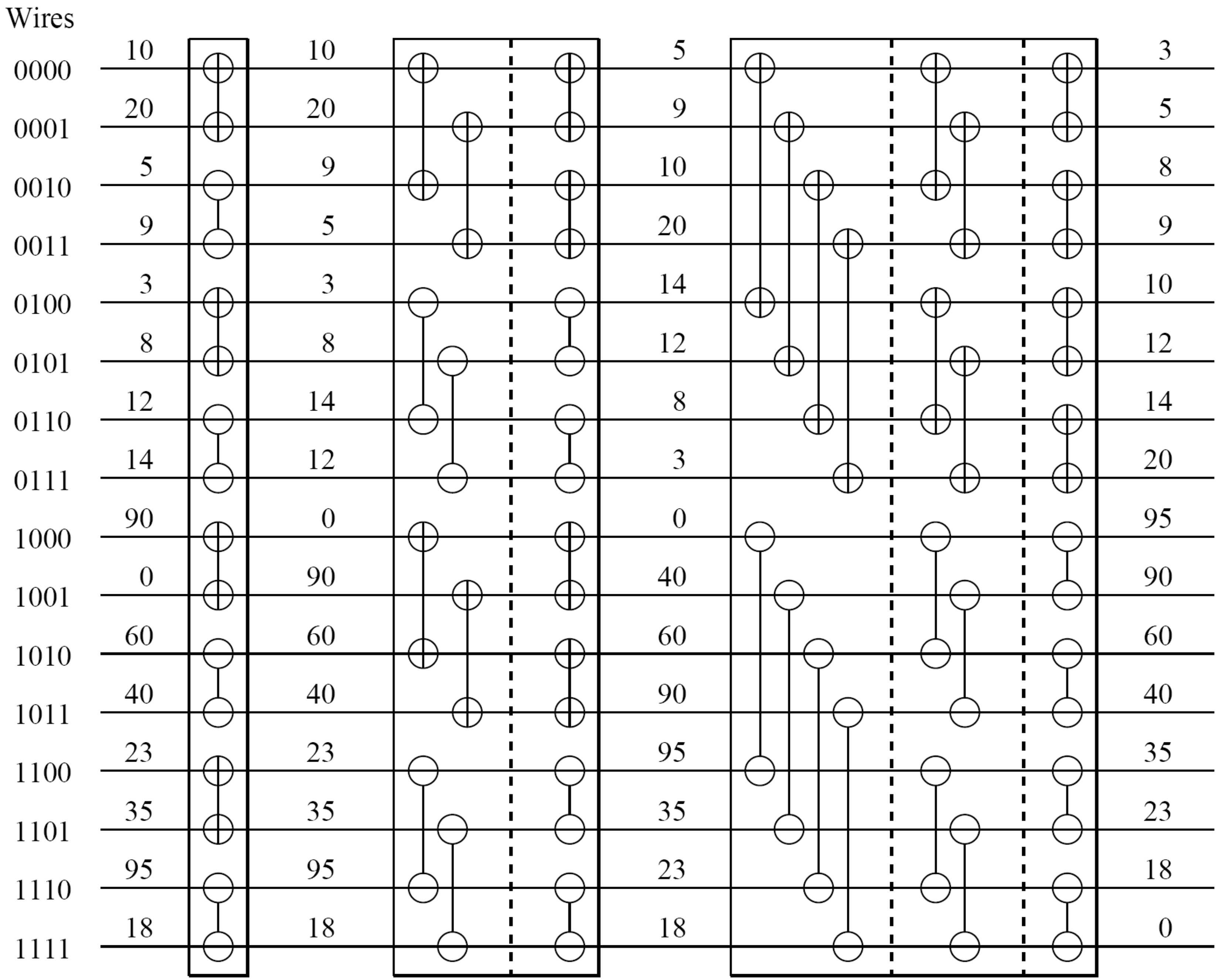


Figure 9.8 The comparator network that transforms an input sequence of 16 unordered numbers into a bitonic sequence. In contrast to Figure 9.6, the columns of comparators in each bitonic merging network are drawn in a single box, separated by a dashed line.

Bitonic Sort

```
for (unsigned int k = 2; k <= NUM; k *= 2) {

    // Bitonic merge:
    for (unsigned int j = k / 2; j>0; j /= 2) {

         unsigned int ixj = tid ^ j;

        // Bitonic split:
        if (ixj > tid) {

            if ((tid & k) == 0) {

                if (shared[tid] > shared[ixj])  swap(shared[tid], shared[ixj]);

            } else {

                if (shared[tid] < shared[ixj])  swap(shared[tid], shared[ixj]);

            }
        }
        __syncthreads();
    }
}
```

Bitonic Sort

- Complexity?
  - Parallel (n processors sort n elements):

Bitonic Sort

- Complexity?
  - Parallel (n processors sort n elements):

$$O(log^2(n))$$

  - Sequential (1 processor sort n elements):

$$O(n\,log^2(n))$$

Odd-Even Sort

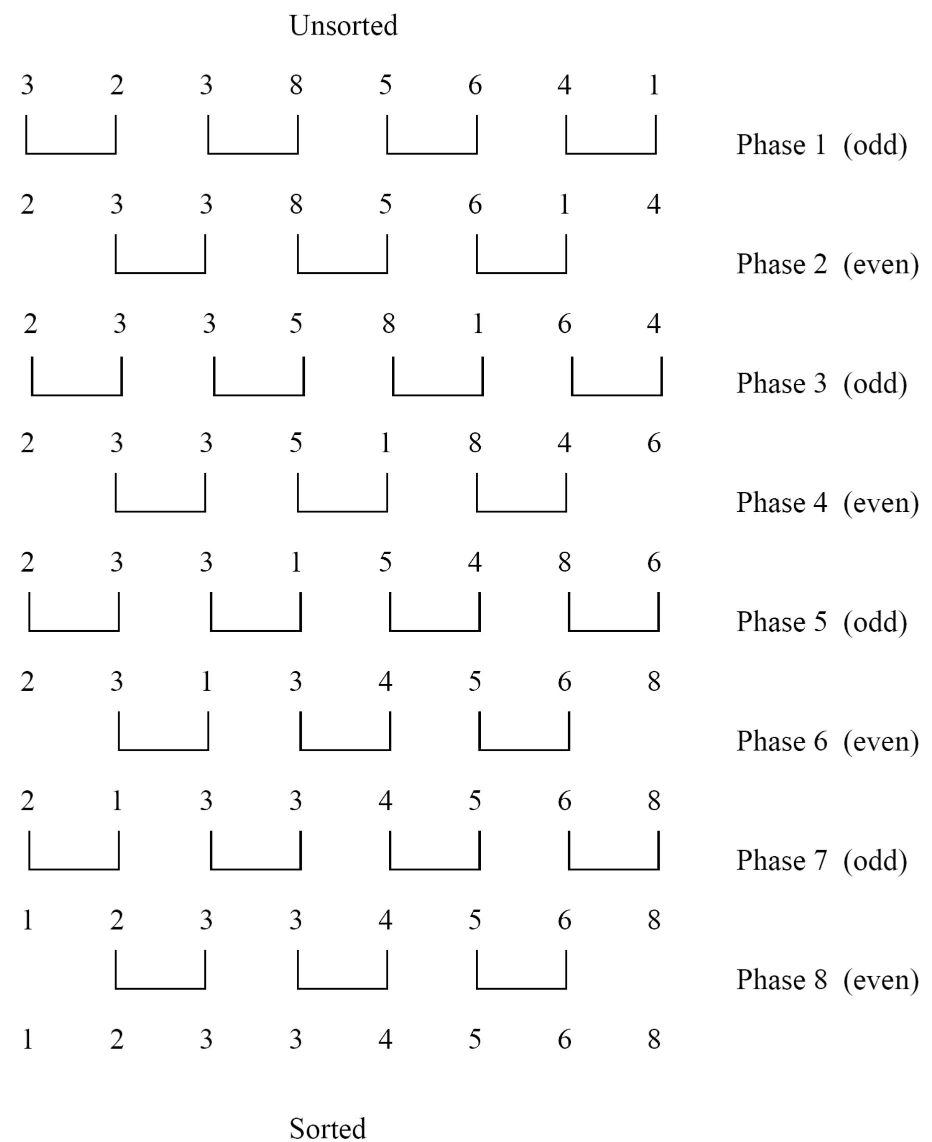


Figure 9.13 Sorting $n = 8$ elements, using the odd-even transposition sort algorithm. During each phase, $n = 8$ elements are compared.

Odd-Even Sort

- Complexity?
  - Parallel (n processors sort n elements):

$$O(n)$$

  - Sequential (1 processor sort n elements):

$$O(n^2)$$

Sort Large Arrays

- Strategy 1: top-down
  - Quick sort or bucket sort to partition large array into smaller sub-arrays, such that each sub-array can fit in one SM.
- Strategy 2: bottom-up
  - Merge sort or bitonic sort to form sorted sub-arrays, then use global merge sort.
- Strategy 3: Global radix sort