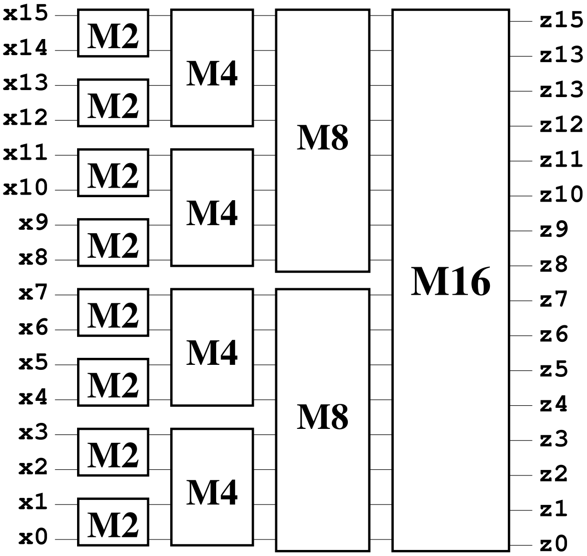
Bitonic Sort

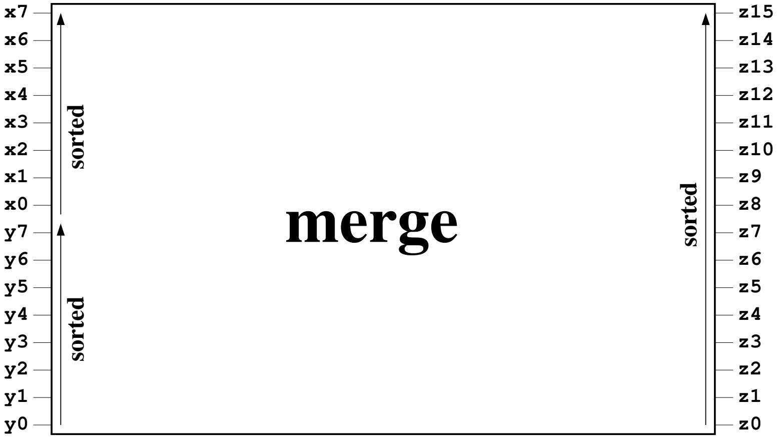
Mark Greenstreet

CpSc 418 – Feb 15, 2017

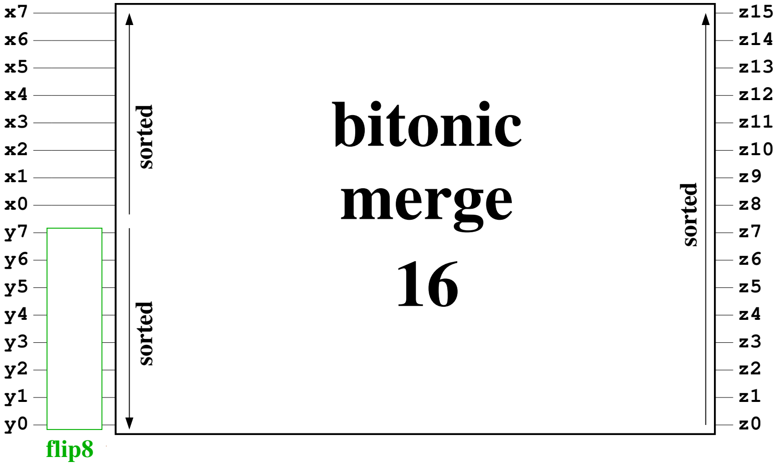
Parallelizing Mergesort



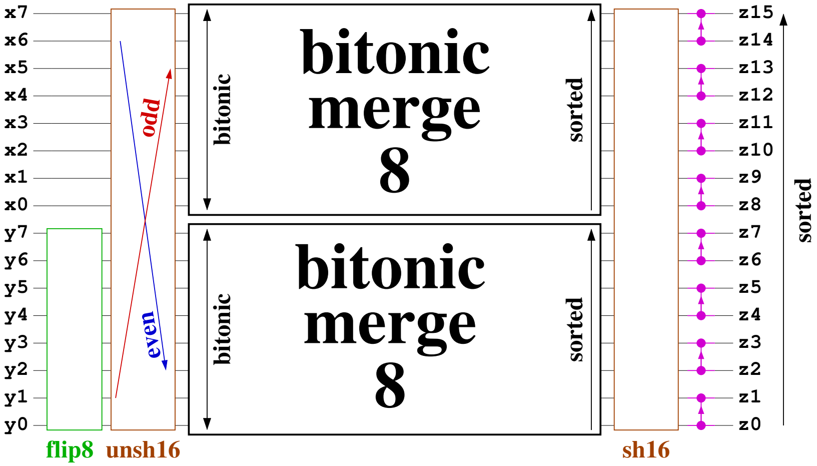
Bitonic Merge



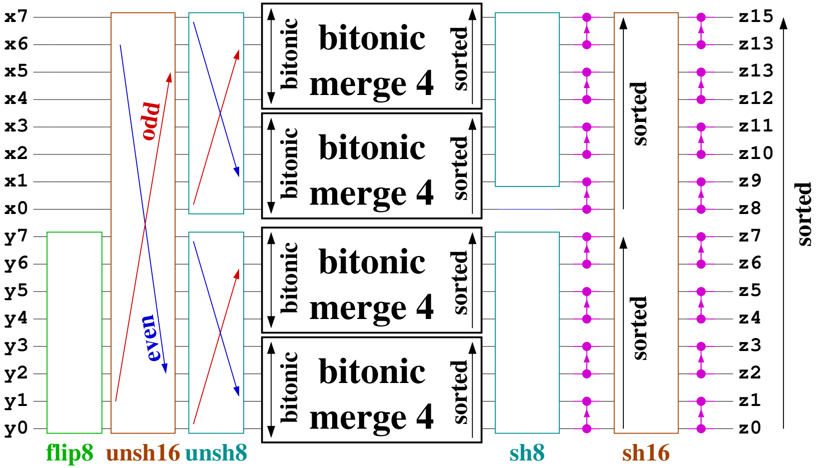
Bitonic Merge



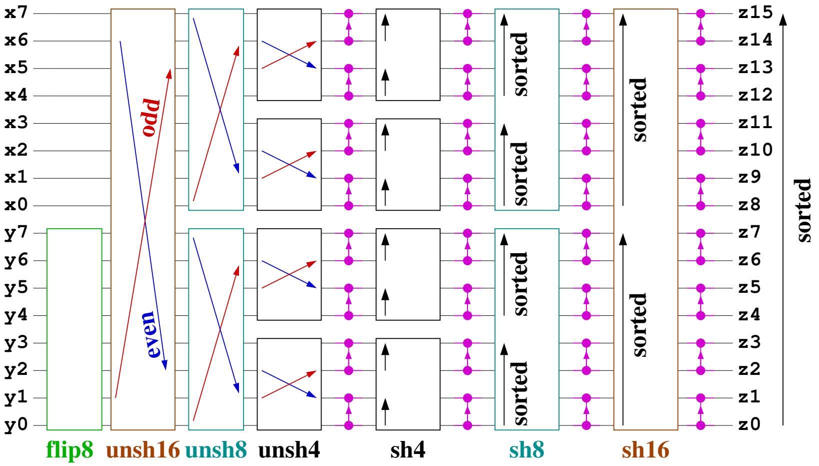
Bitonic Merge



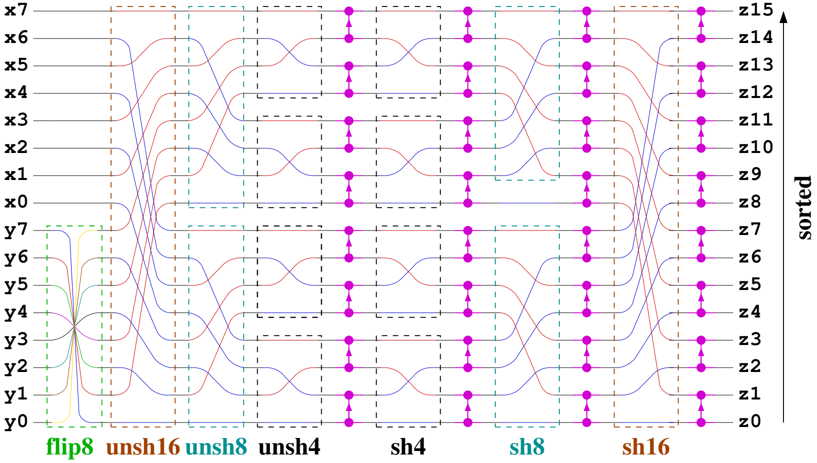
Bitonic Merge



Bitonic Merge



Bitonic Merge



Shuffle

- Given two sequences, X of length N where N is even, the **shuffle** of X is $Y = \text{shuffle}(X)$ where

$$\begin{aligned} Y_i &= X_{i/2}, && \text{if } i \text{ is even} \\ &= X_{(i+N-1)/2}, && \text{if } i \text{ is odd} \end{aligned}$$

  - $\text{shuffle}([0,1,2,3,4,5,6,7]) \rightarrow [0,4,1,5,2,6,3,7]$.
- shuffle is like shuffling a deck of cards.
  - Split the deck in half.
  - Interleave the cards from the two halves.
- If N is a power of 2, then shuffle rotates the least-significant bit of the index to the most significant bit:

```
shuffle([000, 001, 010, 011, 100, 101, 110, 111]) ->
 [000, 100, 001, 101, 010, 110, 011, 111])
```

- If N is odd,

$$\begin{aligned} Z_i &= X_{i/2}, && \text{if } i \text{ is even} \\ &= X_{(i+N)/2}, && \text{if } i \text{ is odd} \end{aligned}$$

  - $\text{shuffle}([0,1,2,3,4]) \rightarrow [0,3,1,4,2]$

Unshuffle

- The inverse of shuffle.
- Let $N = \text{length}(Y)$ and $X = \text{unshuffle}(Y)$, then

$$\begin{array}{rcll} X_i & = & Y_{2i}, & \text{if } i < N/2 \\ & = & X_{2i-N+1}, & \text{if } N/2 \leq i \end{array}$$

- It's like dealing a deck of cards into two piles, and then stacking one pile on top of the other.
- If N is a power of 2, then unshuffle rotates the most significant bit of the index to the least significant bit:
- If N is odd,

$$\begin{array}{rcll} X_i & = & Y_{2i}, & \text{if } i < (N+1)/2 \\ & = & X_{2i-N}, & \text{if } (N+1)/2 \leq i \end{array}$$

Bit operations: `rotr` and `rotl`

- `rotr(I, W)` % Rotate the lower `W` bits of `I` one place to the right:

```
rotr(I, 0) -> I;
rotr(I, W) when is_integer(W), W > 0 ->
   Mask = (1 bsl W) - 1,       % ones in the W least-significant bits
   Ilo = I band Mask,          % the W least-significant bits of I
   Ihi = I band (bnot Mask),   % the rest of I
   Ilsb = I band 1,            % the least-significant-bit of I
   % Ilor is Ilo rotated one place to the right
   Ilor = (Ilsb bsl (W-1)) bor (Ilo bsr 1),
   Ihi bor Ilor.
```

  - `rotr(6,3) -> 5;`
  - `rotr(6,4) -> 12;`
- `rotr(I, W)` rotates the lower `W` bits of `I` 1 place to the left.
- Note: `rotr(I,1) -> I`, and `rotl(I,1) -> I`.

Shuffle, Unshuffle, and Bit-Operations

- If `K` is a power of 2, `x[0..(K-1)]` is the input of a `shuffle_K` module, and `y[0..(K-1)]` is the output, then
  - the `shuffle_K` operation moves `x[i]` to `y[rotl(i, log2(k))]`.
  - equivalently: `y[j] = x[rotr(j, log2(k))]`.
- If `K` is a power of 2, `x[0..(K-1)]` is the input of a `unshuffle_K` module, and `y[0..(K-1)]` is the output, then
  - the `unshuffle_K` operation moves `x[i]` to `y[rotr(i, log2(k))]`.
  - equivalently: `y[j] = x[rotl(j, log2(k))]`.

The Initial Unshuffles

- Bitonic merge for `K` elements starts with an `unshuffle_K`, followed by a `unshuffle_`$\frac{K}{2}$, followed by a `unshuffle_`$\frac{K}{4}$, . . . , followed by a `unshuffle_1`.
- If we let `x[0..(K-1)]` be the input to this network (I'm assuming we've already done the flip for inputs `x[0..((K/2)-1)]`), and `y[0..(K-1)]` be the output then:
  - `y[j] = x[rotl(rotl(...rotl(rotl(j, 1), 2), ..., log2(K)-1), log2(K))]`
  - and we note that:
    ```
      rotl(rotl(...rotl(rotl(j, 1), 2), ...,
    log2(K)-1), log2(K)) = bitrev(j, log2(K))
    ```
    where `bitrev(j, W)` is the bit-reverse of the lower `W` bits of `j`.
- More specifically, for the 16-way bitonic merge, $K = 16$ and $\log_2(K) = 4$.
  - If we write array indices as four bits, b_3, b_2, b_1, b_0,
  - Then `y[`b_3, b_2, b_1, b_0`] = x[`b_0, b_1, b_2, b_3`]`.

The first compare-and-swap

The first compare-and-swap operates on `y`$[b_3, b_2, b_1, 0]$ and `y`$[b_3, b_2, b_1, 1]$, for all 8 choices of b_3, b_2, and b_1.

- This corresponds to a compare-and-swap of `x`$[b_0, b_1, b_2, 0]$ with `x`$[b_0, b_1, b_2, 1]$.
- I'll call the result of the compare-and-swap `z` where
  - `z`$[b_3, b_2, b_1, 0]$ `= min(y`$[b_3, b_2, b_1, 0]$`, y`$[b_3, b_2, b_1, 1]$`);`
  - `z`$[b_3, b_2, b_1, 1]$ `= max(y`$[b_3, b_2, b_1, 0]$`, y`$[b_3, b_2, b_1, 1]$`);`
- And I'll write $\widetilde{\mathtt{z}}$ for `z` with "`x` indexing":
  - $\widetilde{\mathtt{z}}[b_3, b_2, b_1, b_0]$ `= z`$[b_0, b_1, b_2, b_3]$`;`
  - $\widetilde{\mathtt{z}}[0, b_2, b_1, b_0]$ `= min(x`$[0, b_2, b_1, b_0]$`, x`$[1, b_2, b_1, b_0]$`);`
  - $\widetilde{\mathtt{z}}[1, b_2, b_1, b_0]$ `= max(x`$[0, b_2, b_1, b_0]$`, x`$[1, b_2, b_1, b_0]$`)`
  - These are comparisons with a "stride" of 8 (for `x`).

The first shuffle

- The first shuffle takes `z` as an input and I'll call the output `w`.
- The first shuffle is a `shuffle_4`; so
  - `w[i] = z[rotl(i,2)]`.
  - Equivalently, $\mathtt{w}[b_3, b_2, b_1, b_0] = \mathtt{z}[b_3, b_2, b_0, b_1]$.
- Let

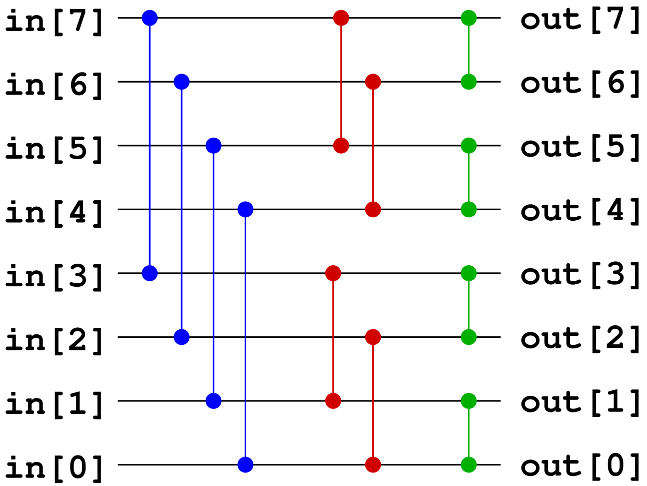
$$\begin{aligned}\widetilde{\mathtt{w}}[b_3, b_2, b_1, b_0] &= \mathtt{w}[b_0, b_1, b_3, b_2] \\ &= \mathtt{z}[b_0, b_1, b_2, b_3] \\ &= \widetilde{\mathtt{z}}[b_3, b_2, b_1, b_0]\end{aligned}$$

- The second stage of compare-and-swap modules operates on
  - $\mathtt{w}[b_3, b_2, b_1, 0]$ and $\mathtt{w}[b_3, b_2, b_1, 1]$
  - Equivalently, $\widetilde{\mathtt{w}}[b_1, 0, b_2, b_3]$ and $\widetilde{\mathtt{w}}[b_1, 1, b_2, b_3]$.
  - These are comparisons with a stride of 4 for $\widetilde{\mathtt{z}}$ and $\widetilde{\mathtt{w}}$.

The rest of the merge

- In the same way, the third stage of compare-and-swap modules operates has a stride of 2 for `x` indices,
- And the final stage has a stride of 1.
- More generally, to merge two sequences of length 2^L:
  - Flip the lower sequence
  -
  - Or, just sort it in reverse in the first place.
  - Perform compare-and-swap operations with stride L.
  - Perform compare-and-swap operations with stride $L/2$ – note that these operate on pairs of elements whose indices differ in the $L/2$ bit, and all of their other index bits are the same.
  - Perform compare-and-swap operations with stride $L/4$, ...
  - Perform compare-and-swap operations with stride 1 – this compares the element at $2*i$ with the element at $2*i+1$ for $0 \leq i < 2^L$.
- Done!

The "Textbook" Diagram

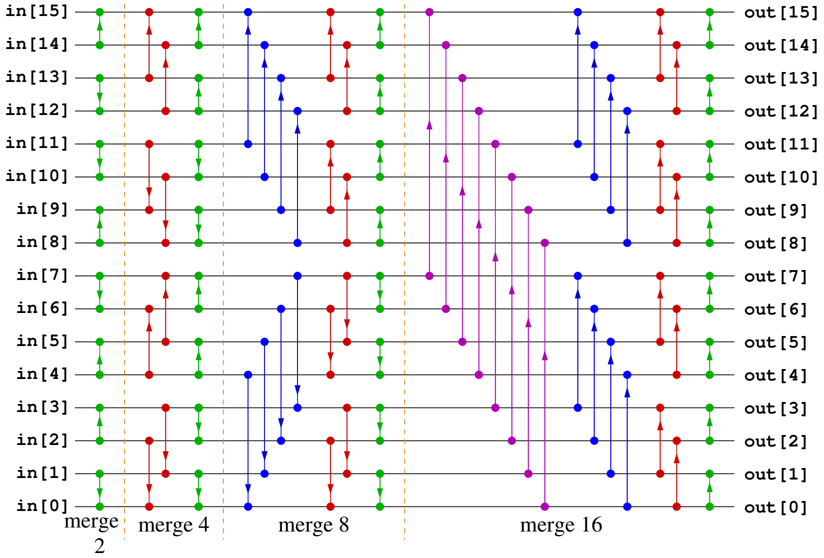


Flipping Out

What should we do about the flips?

- Push them back (right-to-left) through the network
  - Keep track of how many flips we've accumulated.
  - Sort up for an even number of flips.
  - Sort down for an odd number of flips.
- Flip the wiring in the bottom half of each unshuffle.
- In practice:
  - Do the one that's easier for your implementation.

Bitonic Sort



Bitonic Sort in practice

- Sorting networks can be used to design practical sorting algorithms.
- To sort N values with P processors:
  - Divide input into $2P$ segments of length $\frac{N}{2P}$.
  - Each processor sorts its pair of segments into one long segment.
    - The sorted segments are the inputs to the sorting network.
  - Now, follow the actions of the sorting network:
    - Processor I handles rows $2I$ and $2I + 1$ of the sorting network.
    - Each compare-and-swap is replaced with "merge two sorted sequences and split into top half and bottom half."
    - When the sorting network has a compare-and-swap between rows $2I$ and $2I + 1$, each processor handles it locally.
    - When the sorting network has a compare-and-swap between rows $2I$ and $2I + K$ for $K > 1$, then processor I sends the upper half of its data to processor $I + (K/2)$, and processor $I + (K/2)$ sends the lower half of its data to processor I. Both perform merges.
    - Note, if the compare-and-swap was flipped, then flip "upper-half" and "lower half".

Practical performance

- Complexity
  - Total number of comparisons: $O(N(\log N \log^2 P))$.
  - Time: $O\left(\frac{N}{P}(\log N + \log^2 P)\right)$, assuming each processor sorts N/P elements in $O((N/P)\log(N/P))$ time and merges two sequences of N/P elements in $O(N/P)$ time.
- Remarks:
  - The idea of replacing compare-and-swap modules with processors that can perform merge using an algorithm optimized for the processor, is an extremely powerful and general one. It is used in the design of many practical parallel sorting algorithms.
  - Sorting networks are cool because they avoid branches:
    - ★ Ideal for SIMD machines that can't really branch.
    - ★ Need to experiment some to see the trade-offs of branch-divergence vs. higher asymptotic complexity on a GPU.

Related Algorithms

- Counting Networks
  - How to match servers to requests.
- FFT
  - The Platonic Ideal of a Divide-and-Conquer Algorithm
  - Used for speech processing, signal processing, and lots of scientific computing tasks.