LINEAR NON-COMPARISON SORT

Michael You[*]
youmichaelc@gmail.com

September 19, 2022

ABSTRACT

We introduce a non-comparison based sort that runs in linear time, with fixed number bit size, and logic gates of linear size. We show that this sort can be implemented with very simple hardware components, which makes it a good candidate for a sorting algorithm on ASICs or FPGAs.

Keywords Computational Complexity · Circuits · Sorting · Transistors · Parallel

1 Introduction

When we compare two numbers to find the larger one, we are essentially checking digits from the most to least significant bit, and the first bit that is less than the corresponding bit in the other number, is the number that is less. To give an example, consider the following algorithm:

```
LARGER_NUMBER($a$, $b$):
  Suppose $a = a_1 a_2 a_3 \dots a_k$ and similar for $b$

  for $i \in (1, 2, \dots, n)$:
    if $a_i > b_i$:
      return $a$
    else if $a_i < b_i$:
      return $b$

  return EQUAL
```

Listing 1: Algorithm to find the larger of two numbers.

With k binary bits, we can represent 2^k numbers, so it is usually the case that we consider fixed-size integer comparisons on computers constant time, since for $O(k) = O(1)$ for k constant.

A question we may think about with this number comparison algorithm is, *can we compare an arbitrary number of integers and find the largest one in k rounds?*

The idea for this paper is to develop such a parallel comparison algorithm, and apply it to sorting, where we can create a $O(n)$ time algorithm.

This sort was already developed by Ghosh et al. [2019] in 2019, but lacked formal details about the correctness of the proof, and justifications for time (and space) complexity, so this paper goes more in depth in those areas[2]. Their paper demonstrated that this algorithm can be implemented on FPGAs, with small area size and with fast runtimes.

Also mention some limitations of another hardware sort, the one that just maps you to the index (https://www.rroij.com/open-access/hardware-solution-to-sorting-algorithms-a-review.php?aid=90998) Issues with distinct numbers, also 2^k space potentially.

[*]Github: `mikinty`, graduate of Carnegie Mellon University, M.S. in ECE

[2]I also came up with this sorting algorithm independently, but that's not really a big deal because I doubt this algorithm is a new algorithm.

2 Algorithm

For the rest of the paper, we are going to assume that we are sorting binary numbers with k bits.

Also, whenever we have some binary number x_i, we assume it can be represented as

$$x_i = x_{i1}x_{i2}\dots x_{in}, \tag{1}$$

where x_i1 is the most significant bit.

Before we introduce the algorithm to find the largest of n numbers, we will derive the motivation for how we construct this algorithm.

The idea is, when are scanning from the most significant bit to the least significant bit, we will "eliminate" numbers which are clearly not the largest. How do we know which numbers are the largest? Well if we look at some digit, let's say the i^{th} digit, if there is some number with 1 in that digit, then it must be the case that the largest number will have 1 in that digit, otherwise it is smaller. The caveat here is, we have to also be careful with numbers that are already eliminated from previous rounds. E.g. if we have $1101, 1011$, even though 1011 is larger in the 3rd digit than 1101, it was already eliminated in the 2nd round.

With this in mind, let us introduce a few definitions.

Definition 1. For some number x_i, define

$$x'_{ij} = \begin{cases} x_{ij} \wedge \overline{(x_{i,j-1} \oplus b_{j-1})} & j > 1 \\ x_{ij} \wedge \sigma_i & j = 1 \end{cases} \tag{2}$$

.

We define

$$b_j = \bigvee_{i=1}^{n} x'_{ij} \tag{3}$$

We then define the result of comparison for each number

$$r_i = \overline{(x'_{ik} \oplus b_k)} \tag{4}$$

Finally, we define the **starter qualifier variable** as

$$\sigma_i = \begin{cases} 1 & \text{if we are considering } x_i \text{ in this comparison round} \\ 0 & \text{if we are not considering } x_i \end{cases} \tag{5}$$

Intuitively, what is happening here is that we define a new x'_{ij} that sets the bit to 0 if the number has already been disqualified, otherwise leaves it alone. Then, b_j is just taking the logic OR of all of the digits, and represents the largest digit for that digit. We use b_j to qualify numbers for the next round of comparisons. The last result r_i is equal to 1 if the number is the largest in the group, otherwise it is 0 if the number is not. Finally, we have σ_i, because once we find largest numbers, we want to disqualify them for the next comparison set, since we want to find the next largest numbers.

With these definitions, we now have a logic expression that finds the largest numbers in some set of numbers. Suppose this circuit is defined as

Definition 2. `FIND_LARGEST_NUMBER`. This circuit is defined as

- **Input**: $x_1, x_2, \dots, x_n, \sigma_1, \sigma_2, \dots, \sigma_n$
- **Output**: $r_1, r_2, \dots, r_n$

Lemma 1. `FIND_LARGEST_NUMBER` correctly finds the largest of n numbers.

For an example of such a circuit, please see the appendix.

Also give an example of how this algorithm would actually run, like Ghosh et al did.

```
SORT($x_1, x_2, \dots, x_n$):
  sorted_nums = []

  set $\forall i, \sigma_i = 1$
```

```
while $\bigvee \sigma_i = 1$:
  $r_1, r_2, \ldots, r_n$ = FIND_LARGEST_NUMBER($x_1, x_2, \ldots, x_n$)

  let $i'$ be the first of the $r_{i'} = 1, i' \in \{1, 2, \ldots, n\}$

  // We add the largest element to our sorted list
  $\sigma_{i'} = 0$
  sorted_nums.append(x_{i'})

return sorted_nums
```

Listing 2: Sorting algorithm with parallel comparison.

Theorem 1. *The algorithm described in 2 is a sorting algorithm that runs in $O(n)$ time for fixed integer bit size, and takes $O(n^2)$ resources to build.*

Proof. First, we will prove correctness.

We know from 1 that `FIND_LARGEST_NUMBER`correctly finds the largest of n numbers. In our algorithm, we are finding the largest number, and adding it to `sorted_nums`, and disqualifying it from subsequent rounds. This means in `sorted_nums`, every number is larger than all numbers that follow it in order, which means `sorted_nums` is indeed sorted.

For runtime, we have n comparison sets, where we find the largest number. For each of these sets, we run `FIND_LARGEST_NUMBER`, which takes $O(k)$ time. To find the first r_i that is equal to 1, we use the same mask encoder circuit as Ghosh et al. [2019], which has a constant critical path, with runs in constant time. Adding to the `sorted_nums` list takes $O(1)$ time. Therefore, each comparison set runs in $O(k)$ time, and with n sets, we have $O(nk)$ total runtime. As we have said that k is constant, the runtime is $O(n)$.

To calculate the additional space needed, we are going to consider

- Gates:
  - x'_{ij} contributes 2 gates at most, and we have nk of these.
  - b_j requires n gates, and we have k of these, so nk total here.
  - r_i requires 1 gate, and we have n of these.
- Wires:
  - x'_{ij}, b_j, r_i all contribute the same order of wires as the gates they have
  - The mask encoder circuit to find the first $r_i = 1$ takes $\sum_{i=1}^{n}$ wires, so we have $O(n^2)$ wires here.
- Storage:
  - σ_i: Each σ_i is one bit, and we have n of these
  - `sorted_nums`: We are storing all of the numbers in a sorted array, as this is not an in-place sort so we use $O(nk)$ space here.

Combining all of these components, we have

$$O\big(nk + nk + n + n^2 + n + n + nk\big) = O(nk + n^2) = O(n^2). \tag{6}$$

□

3 Discussion

Talk about pipelining and how it's not really necessary

Mention how this is hard to implement in the software level, need large amounts of hardware control.

References

Surajeet Ghosh, Shaon Dasgupta, and Sanchita Saha Ray. A comparison-free hardware sorting engine. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 586–591, 2019. doi:10.1109/ISVLSI.2019.00110.

Explain the parallel operation of the big OR gates.