Delay efficient mux approach for finding the first two minimum values

Nakka Sivaraju¹ S Suman²

¹PG scholar, ECE Department, CEC, AP, India
²Assistant Professor, ECE Department, CEC, AP, India

Abstract
This brief paper presents delay efficient tree based architecture for finding the first two minimum values in a given set of numbers. This paper actually follows comparator architecture that helps to reduce the delay. The tree architecture actually reduces the number of comparisons, which indirectly reduces the delay. The proposed architecture efficiently draws the two minimum values from given set of numbers.

The proposed architecture uses the intermediate comparisons, this reduces the delay elapsed for comparing the already compared number with the other number. The synthesis and simulation is carried out using XILINX ISE 12.3i and HDL is developed using VERILOG language.

Key words: tree architecture, comparator search, sort.

1. Introduction:
Low-density parity-check (LDPC) codes deliver very good performance when decoded with the belief-propagation (BP) or the sum-product algorithm. As LDPC codes are being considered for use in a wide range of applications, the search for efficient implementations of decoding algorithms is being pursued intensively. The BP algorithm can be simplified using the so-called BP-based approximation also known as the “min-sum” approximation, which greatly reduces the implementation complexity but incurs degradation in decoding performance. This has lead to the development of many reduced-complexity variants of the BP algorithm that nonetheless deliver near-optimum decoding performance. To eliminate the complicated hyperbolic computations required in the sum–product decoding algorithm, recent LDPC decoders are implemented based on the min–sum (MS) decoding algorithm. In the MS algorithm, the check-node (CN) operation computes the first two minima and the index of the first minimum among many variable-to-check messages given as inputs.

A magnitude digital comparator is a combinational circuit that compares two digital or binary numbers (consider A and B) and determines their relative magnitudes in order to find out whether one, less than or greater than the other digital number. Three binary variables are used to indicate the outcome of the comparison as A>B, A<B, or A=B. The below figure shows the number is equal block diagram of a n-bit comparator which compares the two numbers of n-bit length and generates their relation between themselves.

Fig 1 These comparator can compare 2-bit, 4-bit and 8-bit depending on the requirement. A comparator used to compare two bits is called a single bit Comparator. It consists of two inputs for allowing two single bit numbers and three outputs to generate less than, equal and greater than comparison outputs. The figure below shows the block diagram of a single bit magnitude comparator. This comparator compares the two bits and produces one of the 3 outputs as L (A<B), E (A=B) and G (A>B).
The truth table for the single bit comparator is given below. When A0 B0 = 00 & 11, both inputs are equal, therefore A=B output will be high. When A0 B0 = 01, B is more than A and hence AB is active.

A 2-bit comparator compares two binary numbers, each of two bits and produces their relation such as one number is equal or greater than or less than the other. The figure below shows the block diagram of a two-bit comparator which has four inputs and three outputs. The first number A is designated as A = A1A0 and the second number is designated as B = B1B0. This comparator produces three outputs as G (G = 1 if A>B), E (E = 1, if A = B) and L (L = 1 if A<B).

By using above obtained Boolean equation for each output, the logic diagram can be implemented by using four NOT gates, seven AND gates, two OR gates and two Ex-NOR gates.

4-Bit Comparator:
It can be used to compare two four-bit words. The two 4-bit numbers are A = A3 A2 A1 A0 and B3 B2 B1 B0 where A3 and B3 are the most significant bits. It compares each of these bits in one number with bits in that of other number and produces one of the following outputs as A = B, A < B and A>B. The output logic statements of this converter are, If A3 = 1 and B3 = 0, then A is greater than B (A>B). Or If A3 and B3 are equal, and if A2 = 1 and B2 = 0, then A > B. Or If A3 and B3 are equal & A2 and B2 are equal, and if A1 = 1, and B1 = 0, then A>B. Or If A3 and B3 are equal, A2 and B2 are equal and A1 and B1 are equal, and if A0 = 1 and B0 = 0, then A > B.

2. Searching and sorting techniques:
In computer science, a search algorithm is an algorithm that retrieves information stored within some data structure. Data structures can include linked lists, arrays, search trees, hash tables, or various other storage methods. Search algorithms can be classified based on their mechanism of searching. Linear search algorithms check every record for the one associated with a target key in a linear fashion. Binary, or half interval searches, repeatedly target the center of the search structure and divide the search space in half. Comparison search algorithms improve on linear searching by successively eliminating records based on comparisons of the keys until the target record is found, and can be applied on data structures with a defined order. Digital search algorithms work based on the properties of digits in data structures that use numerical keys. Finally, hashing directly maps keys to records based on a hash function. Searches outside of a linear search require that the data be sorted in some way.

Search functions are also evaluated on the basis of their complexity, or maximum theoretical run time. Binary search functions, for example, have a maximum complexity of O(log(n)), or logarithmic
time. This means that the maximum number of operations needed to find the search target is a logarithmic function of the size of the search space.

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

**Increasing Order:**
A sequence of values is said to be in increasing order, if the successive element is greater than the previous one. For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.

**Decreasing Order:**
A sequence of values is said to be in decreasing order, if the successive element is less than the current one. For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.

**Non-Increasing Order:**
A sequence of values is said to be in non-increasing order, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.

**Non-Decreasing Order:**
A sequence of values is said to be in non-decreasing order, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.

**Bubble Sort:**
The algorithm works by comparing each item in the list with the item next to it, and swapping them if required. In other words, the largest element has bubbled to the top of the array. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items.

**Selection Sort:**
The algorithm works by selecting the smallest unsorted item and then swapping it with the item in the next position to be filled. The selection sort works as follows: you look through the entire array for the smallest element, once you find it you swap it (the smallest element) with the first element of the array. Then you look for the smallest element in the remaining array (an array without the first element) and swap it with the second element. Then you look for the smallest element in the remaining array (an array without first and second elements) and swap it with the third element, and so on.

Multiplexer means many into one. A multiplexer is a circuit used to select and route any one of the several input signals to a signal output. A simple example of an non electronic circuit of a multiplexer is a single pole multi-position switch. Multi-position switches are widely used in many electronics circuits. However circuits that operate at high speed require the multiplexer to be automatically selected. A mechanical switch cannot perform this task satisfactorily. Therefore, multiplexer used to perform high speed switching are constructed of electronic components. Multiplexer handle two type of data that is analog and digital. For analog application, multiplexer are built of relays and transistor switches. For digital application, they are built from standard logic gates. The multiplexer used for digital applications, also called digital multiplexer, is a circuit with many input but only one output. By applying control signals, we can steer any input to the output. Few types of multiplexer are 2-to-1, 4-to-1, 8-to-1, 16-to-1 multiplexer.

**2 to 1 multiplexer:**
A logic value of 0 would connect to the output while a logic value of 1 would connect to the output. In larger multiplexers, the number of selection pins is equal to where is the number of inputs.

![Fig.5 4-input multiplexer](image)

For S=0, Y=D1  
For S=1, Y=D0.

**Searching method:**
There are two types of search algorithms: algorithms that don’t make any assumptions about the order of the list, and algorithms that assume the list is already in order. We’ll look at the former first, derive the number of comparisons required for this algorithm, and then look at an example of the latter. In the discussion that follows, we use the term search term.
to indicate the item for which we are searching. We assume the list to search is an array of integers, although these algorithms will work just as well on any other primitive data type (doubles, characters, etc.). We refer to the array elements as items and the array as a list. 3.1 Linear search The simplest search algorithm is linear search. In linear search, we look at each item in the list in turn, quitting once we find an item that matches the search term or once we’ve reached the end of the list. Our “return value” is the index at which the search term was found, or some indicator that the search term was not found in the list. 3.1.1 Algorithm for linear search for (each item in list) { compare search term to current item if match, save index of matching item break } return index of matching item, or -1 if item not found.

A novel tree structure is proposed in this brief to minimize the number of comparators as well as the area–time (AT) complexity. Instead of finding the exact second minimum after finding the first minimum, the proposed algorithm collects the candidates of the second minimum while searching for the first minimum. The candidate set is easily constructed by reusing the comparison results performed for the first minimum. Compared to the previous SM, the proposed SM reduces the number of comparators by more than 40%. The rest of this brief is organized as follows. Performance of linear search. When comparing search algorithms, we only look at the number of comparisons, since we don’t swap any values while searching. Often, when comparing performance, we look at three cases:

Best case: What is the fewest number of comparisons necessary to find an item? Worst case: What is the most number of comparisons necessary to find an item? Average case: On average, how many comparisons does it take to find an item in the list? For linear search, our cases look like this: 2 • Best case: The best case occurs when the search term is in the first slot in the array. The number of comparisons in this case is 1. • Worst case: The worst case occurs when the search term is in the last slot in the array, or is not in the array. The number of comparisons in this case is equal to the size of the array. If our array has N items, then it takes N comparisons in the worst case. • Average case: On average, the search term will be somewhere in the middle of the array. The number of comparisons in this case is approximately N/2. In both the worst case and the average case, the number of comparisons is proportional to the number of items in the array, N. Thus, we say in these two cases that the number of comparisons is order N, or O(N) for short. For the best case, we say the number of comparisons is order 1, or O(1) for short. 3.2 Binary search Linear search works well in many cases, particularly if we don’t know if our list is in order. Its one drawback is that it can be slow. If N, the number of items in our list, is 1,000,000, then it can take a long time on average to find the search term in the list (on average, it will take 500,000 comparisons). What if our list is already in order? Think about looking up a name in the phone book. The names in the phone book are ordered alphabetically. Does it make sense, then, to look for “Sanjay Kumar” by starting at the beginning and looking at each name in turn? No! It makes more sense to exploit the ordering of the names, start our search somewhere near the K’s, and refine the search from there. Binary search exploits the ordering of a list. The idea behind binary search is that each time we make a comparison, we eliminate half of the list, until we either find the search term or determine that the term is not in the list. We do this by looking at the middle item in the list, and determining if our search term is higher or lower than the middle item. If it’s lower, we eliminate the upper half of the list and repeat our search starting at the point halfway between the first item and the middle item. If it’s higher, we eliminate the lower half of the list and repeat our search starting at the point halfway between the middle item and the last item.

3. Proposed Method:

For a given set of k w-bit inputs, \( X = \{x_0, x_1, \ldots, x_{k-1}\} \), the SM for \( k \) inputs produces three outputs: 1) the first minimum value \( \text{MIN1} = \min(X) \), 2) the second minimum value, \( \text{MIN2} = \min(X - \{\text{MIN1}\}) \), and 3) the index of the first minimum \( \text{IDX} \), which is \( i \) if \( xi \) is \( \text{MIN1} \). Two 2-input primitive units, \( \text{C1M1} \) and \( \text{C1M2} \), are widely used to realize an SM. As shown in Fig. 1(a), the \( \text{C1M1} \) unit that selects the smaller value from two inputs consists of one comparator and one w-bit as described in the previous literatures. 2-to-1 multiplexer. On the other hand, the \( \text{C1M2} \) unit is made of one comparator and two w-bit 2-to-1 multiplexors to determine both the larger and smaller values, as depicted in Fig. 1(b). For be obtained using the results of comparisons performed for \( \text{MIN1} \). In addition, let the number of inputs \( k \) be a power of 2,
i.e., $k = 2m$. When $k$ is not a power of 2, such an SM can be achieved by pruning some leaf nodes of the balanced SM built with $2m$ inputs where $2m > k$.

**Searching module design:**

![Fig: Tree based searching module design](image)

For a high-speed realization, the tree-based SM architecture, referred to as SMtree, was proposed in. The SM tree designed for eight inputs is exemplified, where the processing times for MIN1 and MIN2 are almost the same as they are both based on the hierarchical tree structure. To calculate exact MIN2 in each sub-tree, however, SM tree requires more comparators than SM-sort. Three C1M1 units and one 2-to-1 multiplexer are additionally used to combine two sub-trees, but the serially connected block required for finding MIN2 in SM sort is removed so that the critical delay of SM-tree is reduced to $3TC + 5TM2$. A faster tree-based SM, denoted as SM-radix, was achieved by adopting the mixed-radix scheme. However, realizing the high-radix computation increases comparators and multiplexors drastically.

It is possible to reduce the number of comparators needed for the second minimum by reusing the comparison results performed for the first minimum. In the proposed architecture, a candidate set $Y$ for MIN2 is first constructed by using the prior comparison results, and then a comparison network is additionally constructed to select MIN2. This two-step approach is conceptually similar to SM-sort, but the second step is much faster in the proposed architecture. As previously discussed, SM-sort requires complex multiplexing networks to construct the candidate set and thus suffers from the long critical delay resulting from $k$-to-1 multiplexors. To eliminate the complex $k$-to-1 multiplexors, the proposed architecture introduces a basic unit, i.e., PRO$k$, which produces the first minimum of $k$ inputs and $m = \log_2 k$ candidates for the second minimum. Similar to SMtree, a PRO$k$ unit can be recursively designed with two smaller PRO$k/2$ units. The first minimum of $k$ inputs, i.e., MIN1, is simply selected by comparing two minima, i.e., MIN(L) and MIN(R), produced in PRO(L) $k/2$ and PRO(R) $k/2$ units, respectively. Depending on the comparison result of the C1M2, the PRO$k$ decides $m − 1$ candidates for the second minimum by selecting either the candidate set of PRO(L) $k/2$ or that of PRO(R) $k/2$. If MIN(L) is smaller than MIN(R), all the $m − 1$ candidates of PRO(R) $k/2$ cannot be the second minimum, because MIN(R) is the smallest value among the $2m−1$ inputs on the right side. Therefore, $m$ candidates for the second minimum are simply formed by including one of MIN(L) and MIN(R) to the $m − 1$ candidates selected by the result of the C1M2. In short, a PRO$k$ unit can be realized with two PRO$k/2$ units, one comparator and $m + 1$ 2-to-1 multiplexors. It is apparent that a PRO2 unit.

![Fig.7 Proposed searching module for eight inputs.](image)
RESULTS:

RTL schematic

![RTL schematic](image)

RTL Internal schematic

![RTL Internal schematic](image)

Technological schematic

![Technological schematic](image)

Output waveforms

![Output waveforms](image)

Conclusion

This paper implements delay efficient architecture for finding the first two minimum values in a given 8-set of numbers. Where, the conventional architecture focuses on finding the first minimum value only. The proposed system consists of 4-bit comparator architecture for finding the minimum and maximum values in a given set, whereas in the conventional system this facility is not included. By the use of this comparator the design achieved low latency then the conventional system.

In this paper we proposed an architecture which consists of 8 elements each of 4-bits width. By using the conventional architecture the design takes 2017.8 pico-seconds on the other hand the proposed tree architecture takes just 2007.4 pico-seconds. On increasing the element size and the bit width this delay may further reduces for the proposed architecture.

References: