New Reconfigurable Algorithm for Sorting Binary Numbers

IHIRRI Soukaina[1*], ERRAMI Ahmed[1], KHALDOUN Mohamed[2]
[1*, 1, 2] NEST Research Group, LRI Laboratory E.N.S.E.M
Hassan II University Casablanca. Morocco
soukainaihirri@ieee.org
aerrami@yahoo.fr
m.khaldoun@ensem.ac.ma

ABSTRACT: *Sorting is one of the major research topics. It is defined as the process of rearranging a sequence of values. Generally, many applications require implementation of sorting algorithm that can be evaluated according to different criteria related to time complexity. In this paper, we present a new reconfigurable algorithm for sorting binary numbers; based on iterative approach; where the data elements are represented in a binary manner. Our system is based on a systematic mathematical calculation, updated iteratively. The proposed approach offers a useful tradeoff between rapidity of the algorithm and the memory resources. A simulation program has been developed in C++ in order to validate the proposed approach. The complexity of our algorithm according to the number of iterations is O (n) in the worst case. However, the complexity reaches O (log_2 (n)) in the best case.*

Keywords: Sorting, Binary Numbers, Reconfigurable Algorithm, C++ simulation

Received: 8 May 2017, Revised 26 June 2017, Accepted 7 July 2017

© 2017 DLINE. All Rights Reserved

1. Introduction

Sorting is one of the huge demand research areas in computer science with great practical importance. Due to the importance of sorting in applications, quite a large number of sorting algorithms have been developed over the decades such as Quick sort[1] [2], Merge sort [3], selection sort and binary sort [4] [5]. Each algorithm has its own pros and cons and a specific methodology to arrange data like merging divide and conquer [6] [7], partitioning, iterative methods etc. Sorting algorithms are classified according to computational complexity, number of swaps, stability, memory requirements, iterative nature, number of comparisons etc. In this paper we proposed an algorithm for sorting binary numbers; providing information to facilitate the locating, searching and arranging information. Our algorithm is based on an iterative process using a systematic mathematical calculation.

The remainder of the paper is organized as follows: Section 2 details the proposed algorithm and its principal with some illustrative Figs. Section 3 discusses the implementation of the algorithm with the detail of procedures and calculation used in the proposed algorithm. Section 4 supports the whole discussion with experimental result to prove the effectiveness of the proposed algorithm and finally conclusion and future works are presented at the end of the paper.

2. Principal of the Algorithm

In this paper, we propose an algorithm for sorting n unsigned binary numbers, assuming that there are no duplicated ones. Let P_j denote the unsorted elements, and let j be the index of each, where $0 < j \leq n$. Our algorithm involves bit testing, with a single pass for each bit in the sortable elements. The idea behind this is to start by examining the most significant bit (MSB) and dealing with the next one in descending order. Our system is based on an iterative pipeline processing, which consists of a chain of stages as shown in Fig.1. The information that flows in this pipe is the index of the unsorted elements. The output of each stage is the input of one of the following stages for the next iteration. As soon as there is a saturation of the pipe, some elements will go to the queue. When several stages are empty, the elements present in the queue occupy the emptied stages according to an order which will be defined by equations presented in the section 3.2.

The comparison at each stage of the pipe is made according to the state of each binary element. Based on the result of this comparison, the element may take 4 positions in the system for its next iterations:

- May stay in the same stage.
- May move to one of the following stages.
- May move to the queue.
- Finish the battle and move to the output data.

We should mention that the elements of the same stage are compared according to the same bit index. For example, all elements of the 2nd stage are compared according to their 2nd bit. To reach the final step of sorting n binary elements, our system uses the concept of iterative processing. This means that the sorting process is iteratively computed till all elements achieve their last bit and sorted in increasing or decreasing order.

The aim contribution of our method is the tradeoff between the memory resources and the rapidity of the algorithm. The more the number of stages increases, the more the system became faster. While decreasing the number of stages lead to a best program in a view of complexity.

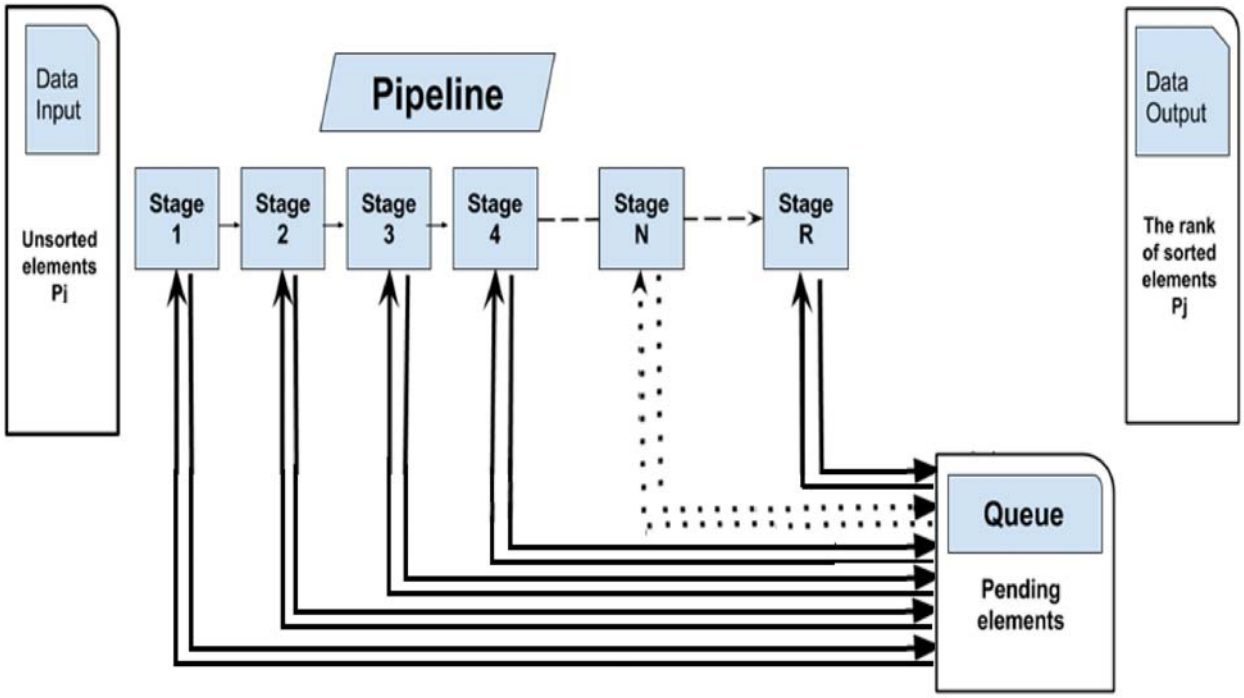

Figure 1. The pipeline system for sorting n binary elements

3. Implementation of the Algorithm

In order to sort n binary numbers, the implementation of our algorithm is based on the use of four matrixes; processing matrix, queue matrix Input and Output matrix. five vectors $x[i]$, $y[i]$, $z[i]$, $SR[i]$ and $M[i]$, where i refer to the index of the iteration.

The data elements are stored in the input matrix of n column and N rows, where n is the number of elements and N is the number of bits of each. We should highlight that each column of the Input matrix refer to the binary value of one of the unsorted element (Fig 2). The Output matrix (Fig3) is similar to the Input one, with same number of rows and columns. It is gradually fills by the binary sequences of the elements. The position of the concerned element in the output matrix refers to its final ranks index. For example, the elements with rank five will be stored in the column five of the output matrix.

Input Matrix

	$P_1[i]$	$P_2[i]$	$P_3[i]$		$P_{64}[i]$
Number Of Bit (N)	1	0	1	.....	1
	1	0	1	.....	0
	1	0	1	.....	1
	0	0	0	.....	1
	1	1	0	.....	0
	0	1	0	.....	0

Figure 2. The Input Matrix used to store data

Output Matrix

$P_5[i]$	$P_8[i]$	$P_{60}[i]$		$P_{20}[i]$
0	0	0	.....	1
0	0	0	.....	1
0	0	0	.....	1
0	0	0	.....	1
0	1	1	.....	1
1	0	1	.....	1

Figure 3. The output Matrix used to store data

To perform the process of sorting, our algorithm uses the processing matrix with *n* column and *N* rows; which refers to the pipeline's stages in order to indicate the position of the element in the pipe as shown in Fig 4. Each column of the matrix contains 0s except one cell contains 1, which corresponds to the stage in which the element P_j is presented in the pipe.

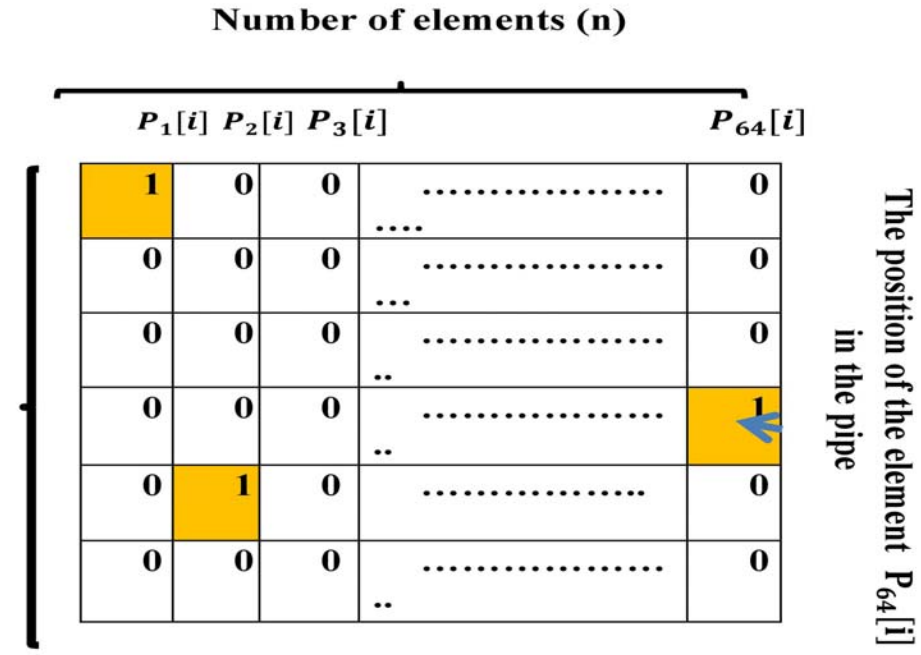

Figure 4. The processing Matrix used to indicate the position of the element in the pipe

For managing the queue, we use the queue matrix represented in Fig 5, similar to the processing matrix and undergoes all rules. Whose column's number corresponds to the number of elements *n*, and a variable line number that can be estimated.

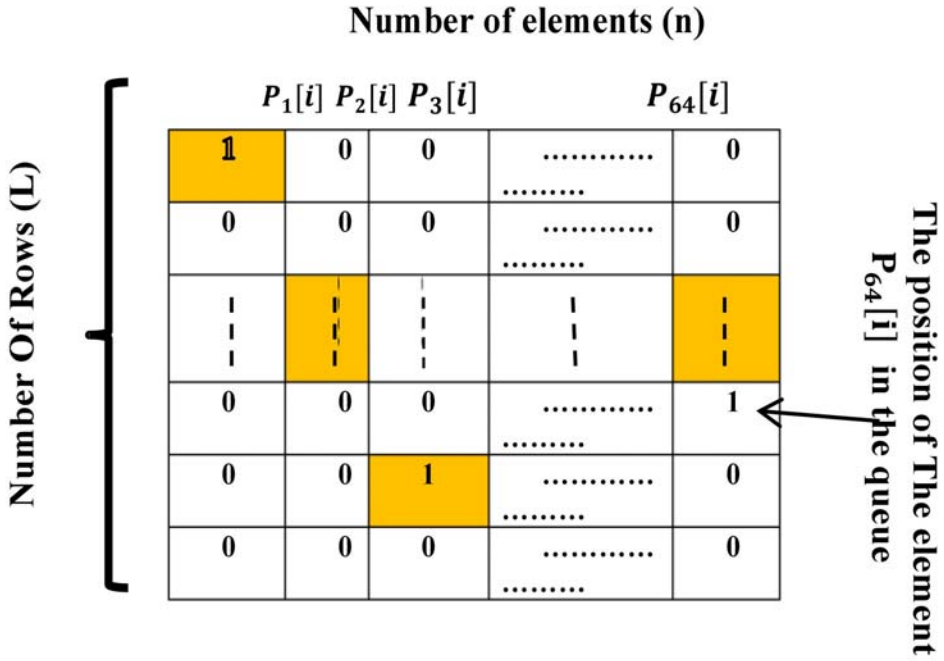

Figure 5. The queue Matrix used for the pending elements

Consequently, we define a vector $x[i]$ which refer to the concerned bit of each element P_j in the data matrix at each iteration i. let $x_j[i]$ denote the bit index of element P_j for the iteration i (see Fig 6).

Figure 6. Vectors used in the process of our algorithm

We also define another vector called $z[i]$ which refer to the stages index in which the element P_j will be processed for the iteration i. we denote $z_j[i]$ as the stage on which the element P_j is going to be processed. Moreover, we define a vector $y[i]$ that indicate on which iteration the element is going to be processed in the pipe, and $M[i]$ indicate the rank of P_j. And finally the algorithm use a stage identifier called $SR[i]$, which indicate the result of the comparison beyond elements of the same stage.

All this parameters are iteratively updated based on a systematic mathematical calculation described in section 2.

1. Procedure of the proposed algorithm

Our algorithm is based on an iterative process that takes place between all the elements of each stage of the pipe. Before describing he process of sorting, we should distinguish between elements in the pipe and others presented in the queue. We start first by inserting elements in the first row (stage) of the processing matrix. It's shown by the presence of 1 in all cells of the first row. All parameters are initialized by the following values related to all P_j's parameters: $x_j[i] = 1$, $y_j[i] = 1$, $z_j[i] = 0$ and $M_j[i] = n$, where $i = 1$ and $0 < j \leq n$. $SR_k[i] = 0$, where k refer to the index of the stage. Once the parameters are initialized, our algorithm works for elements in the processing matrix following the steps below:

Step 1: A comparison is made based on the concerned bit in data matrix of each element P_j; if it is 0 or 1; we define two classes of elements, winner's class and loser's class:

• An element is a winner if the concerned bit is 1, or 0 and any other element of the same stage has 1 as bit ;

• An element is a loser if its concerned bit is 0 and at least one of the elements of the same stage has 1;

Step 2: The operation now is to find if there is at least one loser in each stage; Defining the state of each stage of the "processing matrix" by the use of the vector $SR[i]$.

Each element $SR_k[i]$of the vector $SR[i]$ takes two values, either 0 or 1 depending on the presence of losers in the corresponding stage (row) of the "processing matrix":

• $SR_k[i]$ =1 if at least there is one loser in the stage *k*;

• $SR_k[i]$ = 0 if not;

Step 3: The sum of SR_k in any one row is computed and stored to be used in the parameters computation;

Step 4: Each element P_j determines its rank according to its state if it's a winner or loser by the use of equations (1) and (2) of $M_j[i]$.

Step 5: Each element P_j computes its parameters $x_j[i]$, $y_j[i]$ and $z_j[i]$ for the next iteration as defined by the uses of equations (3),(4),(5) and (6).

Step 6: We start again at step 1 iteratively while the element didn't achieve its last bit" or didn't move to the "queue matrix". Once a PE transmitted its last bit ,it is preserved in the output matrix in the colomne whose index refer to the final rank of the concerned element. In this way, each elemenet will occupy its corresponding column according to its final rank until all the columns are fulfils.

After *N* iterations, we will start filling the data output with the rank of each element. The elements that could not be in the pipe will be waiting in the queue matrix. The transition from the processing matrix to the queue is automatically calculated by the main of parameter *y*. Once $y_j[i+1] > y_j[i] +1$, the element is going to be in the queue for the next iteration. Where $y_j[i]$ represents the current value, $y_j[i+1]$ the next one to be used at the next iteration according to equations (5) and (6). The detailed procedures for elements in queue matrix can be summarized as follows:

Step 1: Place the elements from pending list at their location in the queue matrix according to (8) and (9).

Step 2: All elements in the queue matrix update their parameters except the bit index, remains in standby state until going into the processing matrix by the use of equations (8),(9),(10), (11) and (12).

Step 3: The pending elements are shifted recursively in the queue matrix until $y_j[i+1] = y_j[i] +1$of each one in the queue.

2. Calculation of Parameters

At each iteration *i*, we recover the values of the parameters compute in the preceding iteration. And subsequently we calculated the rank of the elements followed by the calculation of the next values of parameters to be used for the next iteration.

Before we move to describe the computation of the parameters, we check first if the element is ether in the processing matrix or on the queue, and has not achieved its last bit yet.

2.1 For elements in the processing matrix:

a) Each element determines its rank according to its state if it's a winner of loser as we described in step1 of the first process.

• The rank of winners is computed as follow:

$$\mathrm{M_j[i+1] = M_j[i] - \sum_{k=0}^{z_{j-1}[i]} SR_k[i]} \quad (1)$$

• The rank of losers is computed as follow:

$$M_j[i+1] = M_j[i] - \sum_{k=0}^{z_j[i]} SR_k[i] \quad (2)$$

Where *i*: The number of iteration,

j: The index of the element,

k: The index of the stage,

Once the rank of each element is defined, we compute the parameters used for the next iteration:

b) The element's stage in the pipe devoted by $z[i]$ is calculated according to the element's class in the process:

• For winners

$$z_j[i+1] = modulo\left(z_j[i] + \sum_{k=0}^{z_{j-1}[i]} SR_k[i], R \right) \quad (3)$$

• For losers

$$z_j[i+1] = modulo\left(z_j[i] + \sum_{k=0}^{z_j[i]} SR_k[i], R \right) \quad (4)$$

c) The computation of the parameter *y*; which indicate on which iteration the element will be processed in the "processing matrix"; is based on the result of the parameter *z*:

• If $z[i+1] = 0$;

$$y_j[i+1] = y_j[i] + \frac{\sum_{k=0}^{z_j[i]} SR_k[i] + z_j[i]}{R} + 1 \quad (5)$$

• If $z[i+1] \geq 0$;

$$y_j[i+1] = y_j[i] + \frac{\sum_{k=0}^{z_j[i]} SR_k[i] + z_j[i] - 1}{R} + 1 \quad (6)$$

d) For the bit index *x*, it is incremented automatically expect the case of elements in the queue.

$$x_j[i+1] = x_j[i] + 1 \quad (7)$$

2.2 For the Queue Matrix Elements

The difference with the calculation in the case of processing lies in the sum of $SR_k[i]$ which varies up to *R* which represents the total number of stages used. No distinction between loser and winner is taken into consideration in the queue.

e) The calculation of the rank of each pending unit (*Pj*) at each iteration is as follows:

$$M_j[i] = M_j[i-1] - \sum_{k=0}^{R} SR_k[i] \quad (8)$$

Where *i*: The number of iteration,

j: The index of the element,

k: The index of the stage,

R: The total number of stages

f) The stage of each (Pj) on the queue is calculated as follows:

$$z_j[i+1] = modulo\left(z_j[i] + \sum_{k=0}^{z_j[i]} SR_k[i], R \right) \quad (9)$$

g) For the parameter *y*, we check the value of *z* computed for the next iteration:

• If $z[i+1] = 0$;

$$y_j[i+1] = y_j[i] + \frac{\sum_{k=0}^{R} SR_k[i] + z_j[i]}{R} + 1 \tag{10}$$

• If $z_j[i+1] \geq 1$;

$$y_j[i+1] = y_j[i] + \frac{\sum_{k=0}^{R} SR_k[i] + z_j[i] - 1}{R} + 1 \tag{11}$$

h) No calculation is done for the parameter $x[i]$, as it remains in standby until the element goes into the processing matrix.

$$x_j[i+1] = x_j[i] \tag{12}$$

3. Example

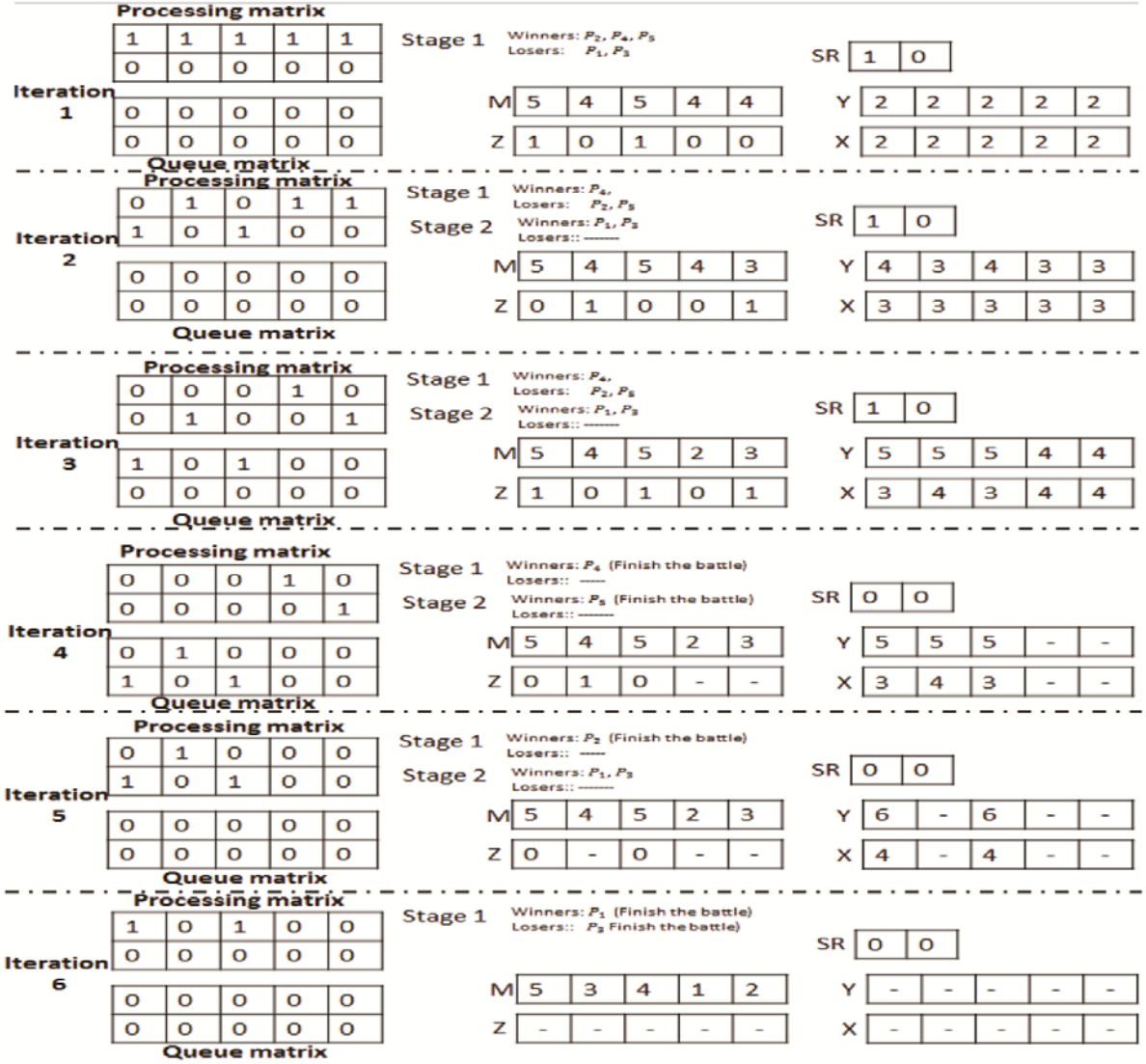

Figure 7. Example of The process of sorting 5-4 binary numbers

In order to describe more our algorithm, we use a representative example of a sample input matrix consisting of 5-4 bit binary numbers P1 → P5:{0011, 1000, 0010, 1100, 1011}, we are required to sort these numbers. At the start of the algorithm, we initialized our vectors as it's shown in the section 3.1. All the numbers are awarded a rank 5 which refers to the max value. In the first iteration, all the elements are in the first stages, which means in the first row of the processing matrix, and it's shown by the presence of 1 in the all cells of the first row. Then we define the state of each element if its winner or loser. After that we move to define the state of each stage by the presence of 1 in the cell of the vector SR, which refer to the stage where is at least one loser. Then we move to compute the next parameters. Fig 7 illustrates the process of sorting this numbers.

1) Experimental Result

In order to test the proposed sorting algorithm, many experiments with different size of data, different number of stages and different numbers of elements were carried out.to validate the approach; we implemented the algorithm in C++. As previously mentioned, the number of stages is taken as input parameters. By means of this point, our algorithm can be reconfigurable. Fig 8 shows number of iterations according to different values of stages for elements size 5,6,7 and 8 bits. It is found that the number of iterations becomes a constant as the number of stages increases.

A more careful analysis reveals that the number of iterations can be reduced by increasing the number of stages until we achieved iteration equal to *N* with $2^N/2$ stages. It is also found that even with a minimum number of stages (1 stage), the number of iteration is equal to the number of unsorted elements. Our results verify that our algorithm is reconfigurable in view of number of stages. The more the number of stages is increased, the more the system gets faster.

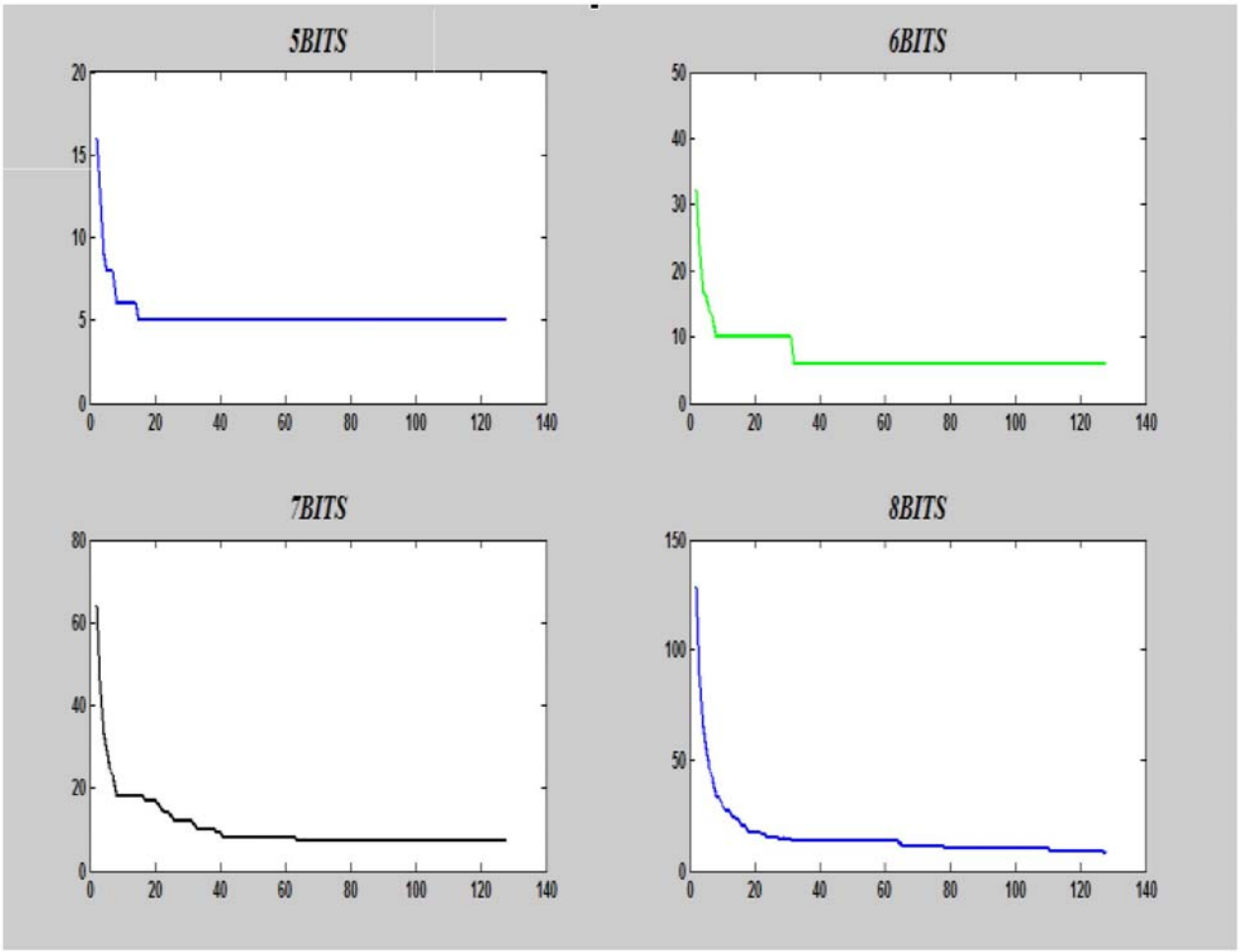

Figure 8. Illustrate the number of iterations according to the number of stage for 5, 6, 7 and 8 bits

Fig 9 reports the achieved sorting iteration for different values of elements. For uniform, the range of values for each bit value is taken from 1 and 2^N. Based on our experiments, we reveal that our algorithm recorded minimum sorting iteration for very large data numbers. For example for N =10 bit, the max number of elements is 1024 = 2^N, with 1 stage, the number of iteration is 2^N, while if we use for example 50 stages, we have just 28 iterations. Or with 500 stages, we can reach 10 = $\log_2$ (2014) Iterations for 1024 elements.

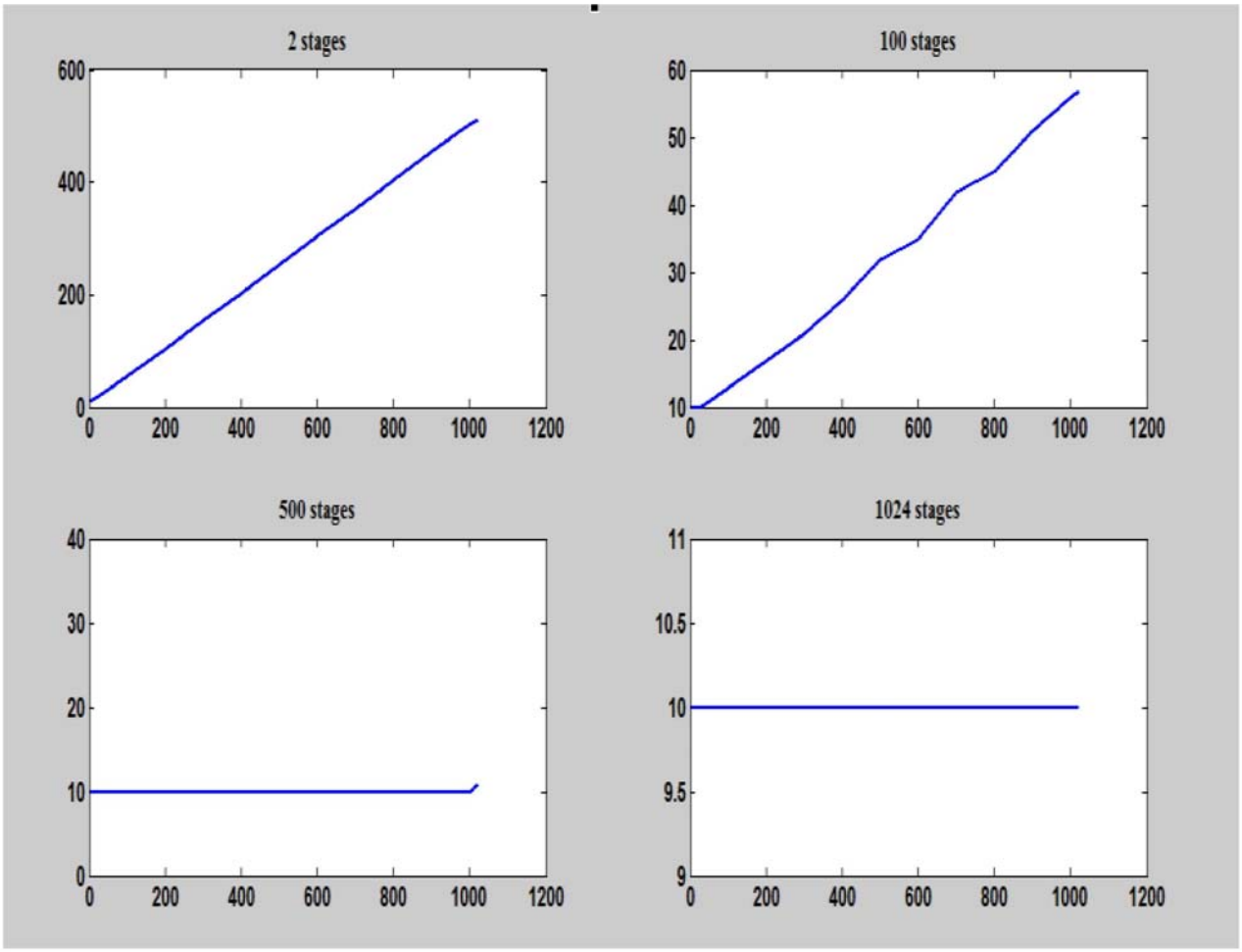

Figure 9. Illustrate the number of iteration according to the number of elements for 2, 100, 500 and 1024 stages

According to the result of Fig 8 and Fig 9 and to a more careful analysis, the complexity for sorting n elements is O ($\log_2(n)$) in the best case and $O(n)$ in its worst case. The main iteration basically consists of a series of computation of the element's rank and parameters used in the next iteration of the sorting process.

2) Comparison with other Sorting Algorithm

Sorting algorithms are evaluated according to the complexity and the algorithm's feasibility in a view of computer architecture. The measurement of any sorting algorithm is based especially in the computational complexity as discussed in[8] [9]. For typical sorting algorithm good behavior is O ($nlog$ (n)) and bad behavior is Ω (n^2). If the problem size is n, the good nature of sorting algorithm is O (n), in average case, the medium nature of sorting algorithm is O ($nlog$ (n)) and in worst case, the bad nature of sorting algorithm is O (2^n). An ideal sorting algorithm in all the cases takes O ($nlog$ (n)) time[7]. Based on our experiments with different size of elements, and comparing to the existing results of sorting algorithms, our proposed algorithm is one of the fastest sorting algorithm that have been proposed. Our computational complexity is significantly faster in practice with O (n) in the worst case. It's also found that in the best case, we achieved to implement our algorithm with complexity of O ($\log_2(n)$).

Conclusion and Future work

In this paper, we have presented a new algorithm for sorting binary numbers, which in turn is an efficient reconfigurable algorithm. By changing the size of the pipe, we can balance the performance in terms of memory. The more the number of stages is increased, the more the system gets faster. Thus, this is a very efficient and reconfigurable algorithm that can dramatically reduce the time taken for tasks accomplished by computers. In order to validate our algorithm, we implemented in C++. As a performance analysis, we show that the complexity in view of number of iterations of our algorithm is with $O(n)$ in the worst case and $O(\log_2(n))$ in the best case.

Our future work is to parallelize our proposed algorithm. It seems interesting to make the parallel version of the algorithm and to implement the algorithm in parallel hardware architecture because instead of going through all the elements (n operations), it become one operation. We look for sorting a list of items with the presence of duplicated ones.

References

[1] Yueying, P., Shicai, L., Miao, L. (2007). Quick Sorting Algorithm of Matrix, *In*: 8th International Conference on Electronic Measurement and Instruments. ICEMI '07, 2007, p. 2-601-2–605.

[2] Xiang, W. (2011). Analysis of the Time Complexity of Quick Sort Algorithm, *In*: 2011 International Conference on Information Management, *Innovation Management and Industrial Engineering*, vol. 1, p. 408–410.

[3] Zhao, F., Xiao, G., Song, Z., C. Peng, (2016). Insertion sort correction of two-way merge sort algorithm for balancing capacitor voltages in MMC with reduced computational load, *In*: 2016 IEEE 8th International Power Electronics and Motion Control Conference (IPEMC-ECCE Asia), p. 748–753.

[4] Alaparthi, S., Gulati, K., Khatri, S. P. (2009). Sorting binary numbers in hardware - A novel algorithm and its implementation, p. 2225–2228.

[5] Hatirnaz, I., Leblebici, Y. (2000). Scalable binary sorting architecture based on rank ordering with linear area-time complexity, p. 369–373.

[6] Patel, Y. S., Singh, N. K., Vashishtha, L. K. (2014). Fuse sort algorithm a proposal of divide & conquer based sorting approach with O(nloglogn) time and linear space complexity, p. 1–6.

[7] Patel, Y. S., Singh, N. K., Vashishtha, L. K. (2014). Fuse sort algorithm a proposal of divide amp; conquer based sorting approach with O(nloglogn) time and linear space complexity, *In*: 2014 International Conference on Data Mining and Intelligent Computing (ICDMIC), p. 1–6.

[8] Alnihoud, J., Mansi, R. (2008). An Enhancement of Major Sorting Algorithms.

[9] Aki, S. G. (1989). The design and analysis of parallel algorithms.