Merge-Sort algorithm in VHDL

Giulio Lotto

February 12, 2024

Abstract

In this Pdf I will discuss the possible implementation of a Merge-Sort algorithm using Vhdl, showing its behavior of quality, deficiency, and possible improvements of the algorithm used.

1 Introduction

The Merge-Sort is a highly effective method for sorting data, relying on the approach of breaking down the task into smaller parts and conquering them individually. In the world of sorting, there exist a lot of algorithms that compute this task, such as the Quick-Sort, the Heap-Sort, the Bubble-Sort and many others. The Merge-Sort is one of the fastest sorting algorithms, it's stable and it has a consistent performance with a time complexity of O(n log n); it is also suitable for large datasets.
The goal is to acquire at the input of the Merge-Sort an array of a maximum of 1024 numbers in random order and give as output an array of the same length containing the same numbers but in ascending order. To implement this algorithm I used VHDL on the Vivado platform.

2 Working principle of a Merge-Sort algorithm

A Merge-Sort algorithm usually works in this way:

1. Divide: Divide the unsorted list (array) into two halves recursively until each sub-list contains only one element. This is done by finding the middle point of the array.

2. Merge: Merge the sorted sub-lists to produce new sorted sub-lists until there is only one sorted list remaining. This is done by comparing elements from each sub-list and combining them in a sorted manner.

The problem is that in VHDL is wiser not to use recursion, so the merge algorithm must be written in a non-recursive way. This problem will be discussed in the following section.

(1)

3 Merge-Sort algorithm

The numbers are taken from the AXI-Slave and stored in a SIPO. When the last data arrives the conversion can begin. The idea is to continuously cut down the input into multiple sub-lists until each of them has only one item, then merge those sub-lists into a sorted array.

Algorithm 1 Merge-Sort

```
1 Comparison:Process(reset,input)
2  variable a,b : integer;
3  begin
4       if reset='1' then
5       output<=(Others=>(Others=>'0'));
6       else
7       for i in 0 to (NUMBER_OF_INPUT_WORDS-1)/L loop
8          a:=0;
9          b:=0;
10         for j in 0 to L-1 loop
11            if b>=L/2 and j>=L/2 then
12             output(L*i+j)<=input(L*i+j);
13            elsif a>=L/2 and j>=L/2 then
14             output(L*i+j)<=input(L*i+j-L/2);
15            elsif input(L*i+j-a)<=input(L*i+j+L/2-b) then
16             output(L*i+j)<=input(L*i+j-a);
17             b:=b+1;
18            elsif input(L*i+j-a)>input(L*i+j+L/2-b) then
19             output(L*i+j)<=input(L*i+j+L/2-b);
20             a:=a+1;
21            end if;
22         end loop;
23      end loop;
24      end if;
25 end process;
```

This is part of the algorithm used to implement the Merge-Sort. The program enters the process when the reset signal or the input signal (that is a custom type containing 32 std_logic_vector(31 downto 0)) changes.
To better explain this process it can be useful to do an example: let's suppose that the input vector is an 8 elements array composed of two vectors of 4 elements already sorted (ex:[1,4,5,6; 2,3,7,8]) and we want the output to be the sorted array (ex:[1,2,3,4,5,6,7,8]). The first *for loop* divides the input vector to have sub-elements of length=L=8 (that is not the number of input values but the number of values that will be sorted at the end of the process,

it could have been also 4 or 2), while the second one is needed to check all the possible sorting cases. The third and the fourth *if statement* check which value is lower between the first of the first sub-vector and the second of the second sub-vector and put it at the output (ex: $1 < 2$ so the first value at the output will be 1). Then the counter of the respective sub-vector (a or b) is increased by one to let the next *for loop* check the next element (ex: b=1). The algorithm continues doing the same thing until the value a or b is equal at L/2 (The length of the sub-vector). Then the algorithms put the values of the remaining sub-vector into the output. For example, with input:[1,4,5,6; 2,3,7,8] we have:

1. $1 < 2$; b=1; output[1]
2. $4 > 2$; a=1; output[1,2]
3. $4 > 3$; a=2; output[1,2,3]
4. $4 < 7$; b=2; output[1,2,3,4]
5. $5 < 7$; b=3; output[1,2,3,4,5]
6. $6 < 7$; b=4; output[1,2,3,4,5,6]
7. remaining vector:=[7,8]; b=4; output[1,2,3,4,5,6,7]
8. remaining vector:=[8]; b=4; output[1,2,3,4,5,6,7,8]

All these steps will be computed in just one clock cycle (leading us to an interesting result about the maximum number of elements that can be sorted with this algorithm).
When the conversion is over the numbers are put in a PISO that will store the data and send them to the output using an Axi-Master.

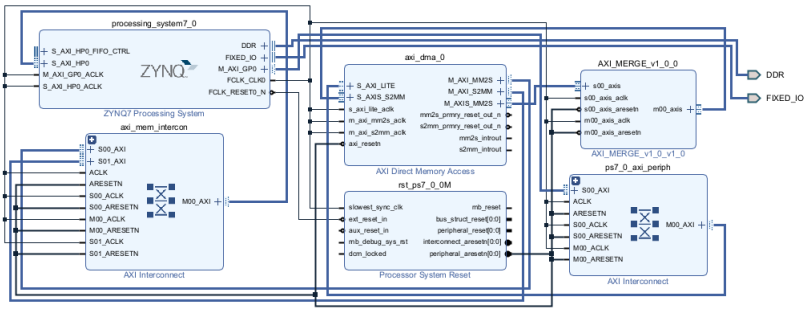


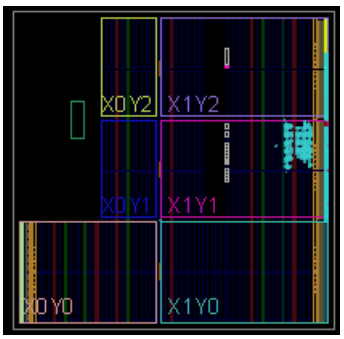
Figure 1: Block Design

4 Final considerations

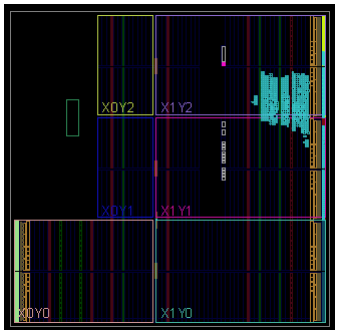
The initial plan was to develop a Merge-Sort algorithm capable of sorting 1024 input numbers. However, the algorithm's computational demands surpass the board's capabilities. As the number of input numbers increases, it becomes evident that the current hardware is insufficient to meet the algorithm's requirements. Therefore, it is crucial to explore alternative solutions, such as upgrading the hardware or optimizing the algorithm to use resources more efficiently.

One potential solution could be using a Merge-Sort tree approach that computes only one value per clock cycle. Implementing multiple Merge-Sort trees of various lengths could solve the problem, although the algorithm would be slower, it would be more lightweight.

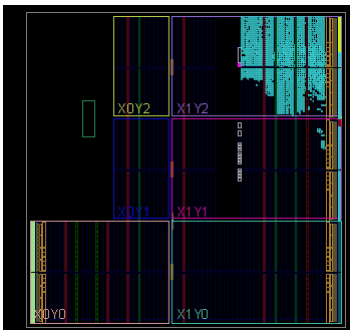
Some pictures of the implementations of the algorithm for different numbers of bits will follow. It is clear that using this algorithm the maximum number of elements (that are power of two) that can be sorted is 32.



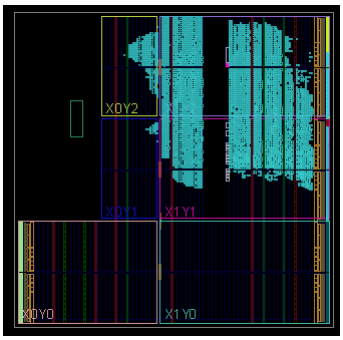
(a) Implementation 2 elements



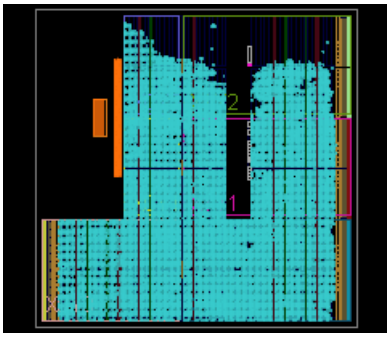
(b) Implementation 4 elements



(c) Implementation 8 elements



(a) Implementation 16 elements



(b) Implementation 32 elements