VHDL Merge Sort Algorithm Implementation on FPGA

Marco La Barbera

February 11, 2024

Abstract

This paper discusses the implementation of a merge sort algorithm on a Xilinx FPGA using Vivado and VHDL. The project aimed to merge a set of numbers while facing limitations due to the finite hardware resources available on the PYNQ Z2 board. Despite encountering challenges with resource constraints, the project successfully implemented the core algorithm that merges 32 numbers using an AXI stream interface. There is also a second version, utilizing LEDs to indicate completion, but few times it presents a bug where one number was not correctly reordered.

This paper discusses the algorithm's design, implementation challenges, and potential avenues for optimization.

1 Introduction

Merge Sort is a fundamental sorting algorithm known for its efficiency and scalability. When implemented on hardware such as FPGAs, it offers parallelism and speed advantages. This project focuses on implementing merge sort using Vivado and VHDL on the PYNQ-Z2 board.

2 Merge Sort Algorithm

2.1 The algorithm

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array. The algorithm consists of five main stages:

1. Splitting: The input list is divided into smaller sub-lists.
2. Sorting: Each sub-list is sorted.
3. Merging: Sorted sub-lists are merged pairwise to produce larger sorted sub-lists.
4. Combining: Merged sub-lists are recursively combined until the entire list is sorted.
5. Output: The sorted list is produced.

2.2 RTL and Implementation with AXI Stream Interface

The merge sort algorithm is implemented using VHDL and integrated with an AXI stream interface for data communication. This allows for efficient data transfer between the FPGA and external components. The main structure is composed as follows:

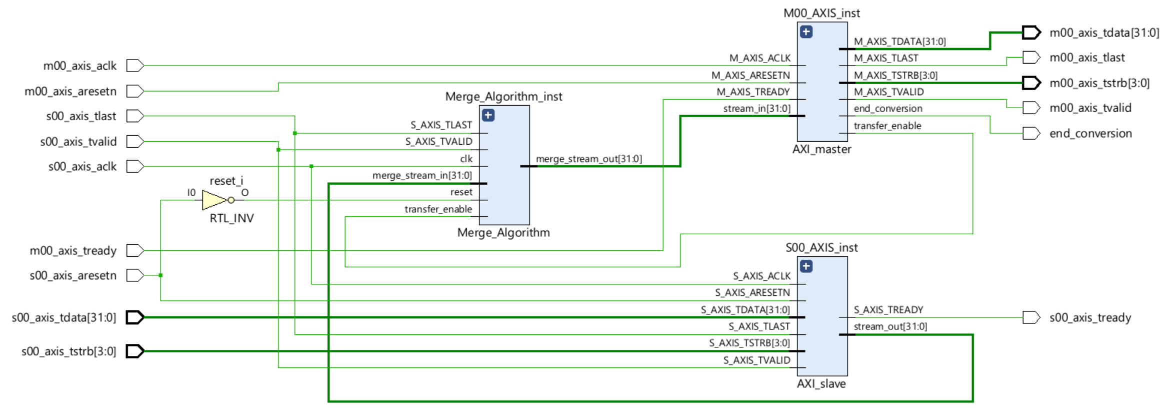


Figure 1: RTL scheme

The input goes through an AXI slave, followed by the merge sort module which processes the received data. Subsequently, an AXI master handles the delivery of the sorted data to the output.

The presence of the "end_conversion" output is noticeable, serving as a signal to inform the user about the completion of the conversion process. While the principal file lacks this output, its removal doesn't alter the RTL structure.

Algorithm 1 VHDL Merge-Sort Algorithm

```
Merge_Sort_algorithm:Process(reset , array_in)
variable a,b : integer;
begin
if reset='1' then
    output <= (Others => (Others => '0'));
else
    for i in 0 to (N_El-1)/part loop
        a:=0;
        b:=0;
        for j in 0 to (part - 1) loop
            if b >= (part/2) then
                array_out((part)*i+j) <= array_in((part)*i+j);
            elsif a >= (part/2) and j >= (part/2) then
                array_out((part)*i+j) <= array_in((part)*i+j-(part/2));
            elsif array_in((part)*i+j-a) <= array_in((part)*i+j+(part/2)-b) then
                array_out((part)*i+j) <= array_in((part)*i+j-a);
                b := b + 1;
            elsif array_in((part)*i+j-a) > array_in((part)*i+j+(part/2)-b) then
                array_out((part)*i+j) <= array_in((part)*i+j+(part/2)-b);
                a := a + 1;
            end if;
        end loop;
        end loop;
end if;
end process;
```

The algorithm above outlines a segment of the Merge-Sort algorithm implementation. The program initiates the process upon a change in either the reset signal or the input signal.

When entering the first loop, it becomes apparent that the total number of elements is divided by the "part" value (e.g., 32 elements: 8 arrays of 4 elements).The objective is to prepare for the next stage, where 8 arrays will be merged into 4 sub-arrays, each containing double the number of elements compared to the previous stage and sorted in relation to each other (e.g., 8 elements in this instance for 4 sub-arrays). Therefore, with part = 8 representing the desired number of elements for each new sub-arrays, the index of the main loop ranges from 0 to (32-1)/8 = 3, determining the exact number of new sub-arrays.

The inner loop serves as an index, ranging from 0 to 7, and its purpose is to sort each pair of input sub-arrays by scrolling through them until the output sub-array is completely filled. Subsequently, another main loop can start.

Upon completion of synthesis, the functional simulation yields the following results:

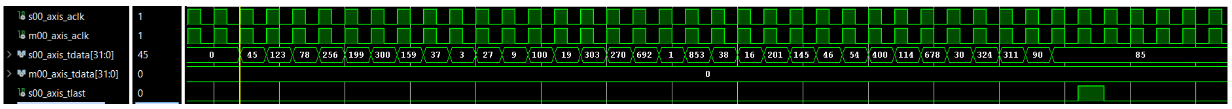


Figure 2: Synthesis Slave Input

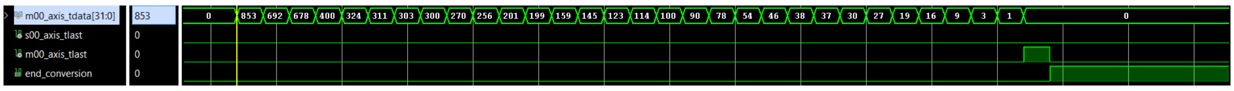


Figure 3: Synthesis Master Output

Now, the project is ready to be packaged as an IP repository file, ready for implementation within a block design.

2.3 Block Design implementation

The merge algorithm interacts with the DMA through the AXI Stream interface, which simplifies data management within the IP. Subsequently, the DMA is automatically linked to the Zynq processor using the AXI Lite protocol.

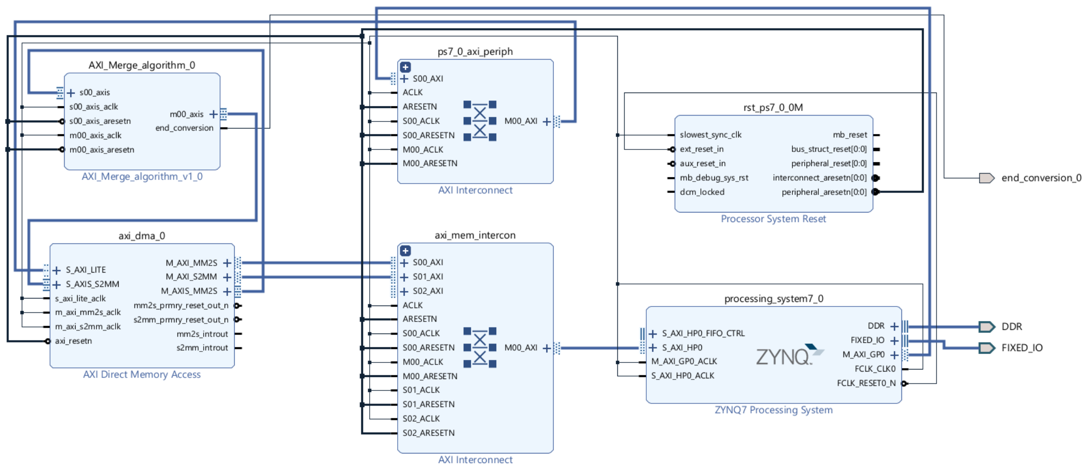
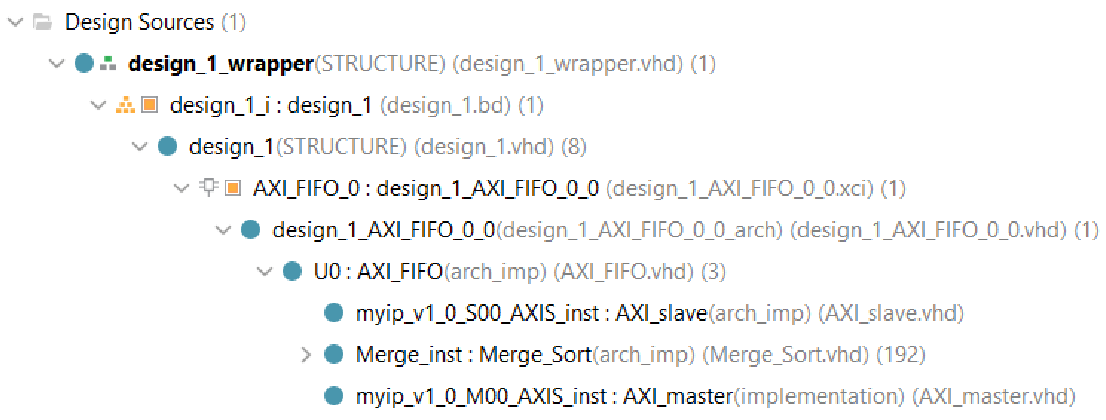


Figure 4: Block Design scheme, project with no LED

Following the utilization of the "create HDL wrapper" function, Vivado autonomously manages the entire structure, ultimately presenting the following arrangement:



```
Design Sources (1)
  design_1_wrapper(STRUCTURE) (design_1_wrapper.vhd) (1)
    design_1_i : design_1 (design_1.bd) (1)
      design_1(STRUCTURE) (design_1.vhd) (8)
        AXI_FIFO_0 : design_1_AXI_FIFO_0_0 (design_1_AXI_FIFO_0_0.xci) (1)
          design_1_AXI_FIFO_0_0(design_1_AXI_FIFO_0_0_arch) (design_1_AXI_FIFO_0_0.vhd) (1)
            U0 : AXI_FIFO(arch_imp) (AXI_FIFO.vhd) (3)
              myip_v1_0_S00_AXIS_inst : AXI_slave(arch_imp) (AXI_slave.vhd)
              Merge_inst : Merge_Sort(arch_imp) (Merge_Sort.vhd) (192)
              myip_v1_0_M00_AXIS_inst : AXI_master(implementation) (AXI_master.vhd)
```

Figure 5: Block Design hierarchy

It's clear how the Merge structure, the core component, is effectively stored and incorporated into the block design.

2.4 Resource Constraints and Optimization

One significant challenge encountered during implementation was the limited number of LUTs available on the FPGA. With only 32 numbers to sort, the algorithm reached the LUT limit (35.300/53.200), making it impossible to scale up to the required 1024 numbers. Further optimization strategies are necessary to address this limitation, such as optimizing the VHDL code or exploring alternative architectures that we will discuss in other paragraphs.

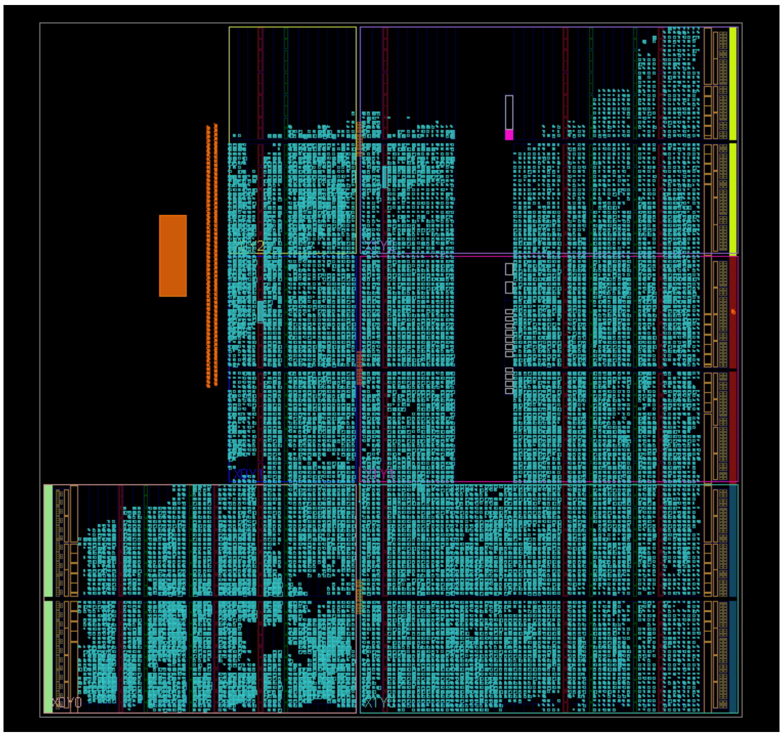
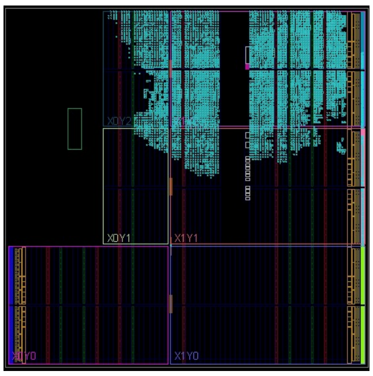
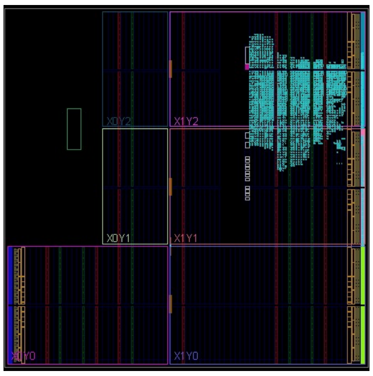


Figure 6: Device Used Area after BD Implementation

Across all the images, it is evident that there is a consistent increase in the covered area, which grows at a rate slightly less than quadratic with the number of sorted numbers.



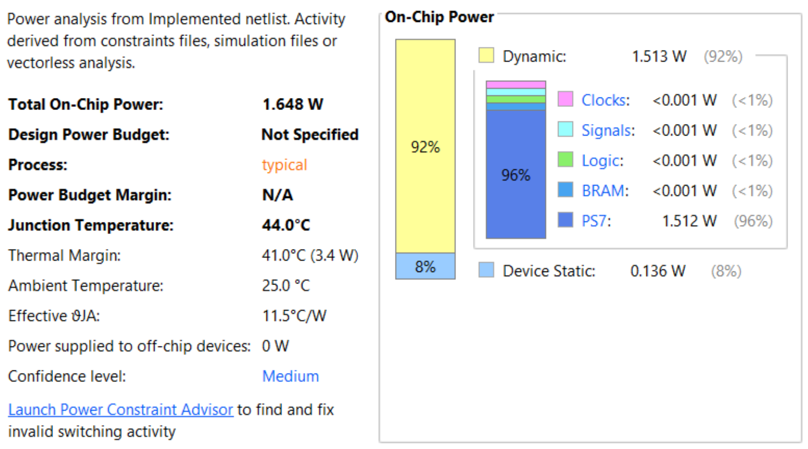
(a) 16 elements implementation



(b) 8 elements implementation

Figure 7: Implementation of the AXI Merge Algorithm with 16 (a) and 8 (b) numbers

In analyzing power consumption, it's apparent that dynamic power prevails over static power, with the former being influenced significantly by the PS7, which plays a pivotal role in consumption.


Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**1.648 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**44.0°C**
Thermal Margin:	41.0°C (3.4 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

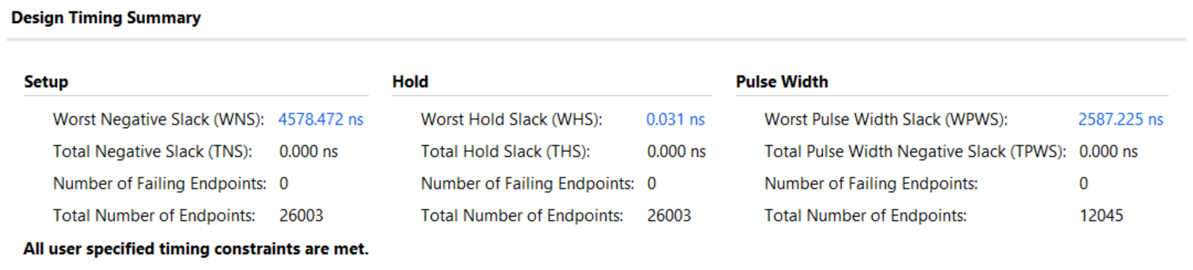
Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	1.513 W	(92%)	
Clocks:	<0.001 W	(<1%)	
Signals:	<0.001 W	(<1%)	
Logic:	<0.001 W	(<1%)	
BRAM:	<0.001 W	(<1%)	
PS7:	1.512 W	(96%)	
Device Static:	0.136 W	(8%)	

Figure 8: Power Consumption

All specified timing requirements have been successfully fulfilled, meeting the established criteria without any critical delays.


Design Timing Summary

Setup

Worst Negative Slack (WNS):	4578.472 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	26003

Hold

Worst Hold Slack (WHS):	0.031 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	26003

Pulse Width

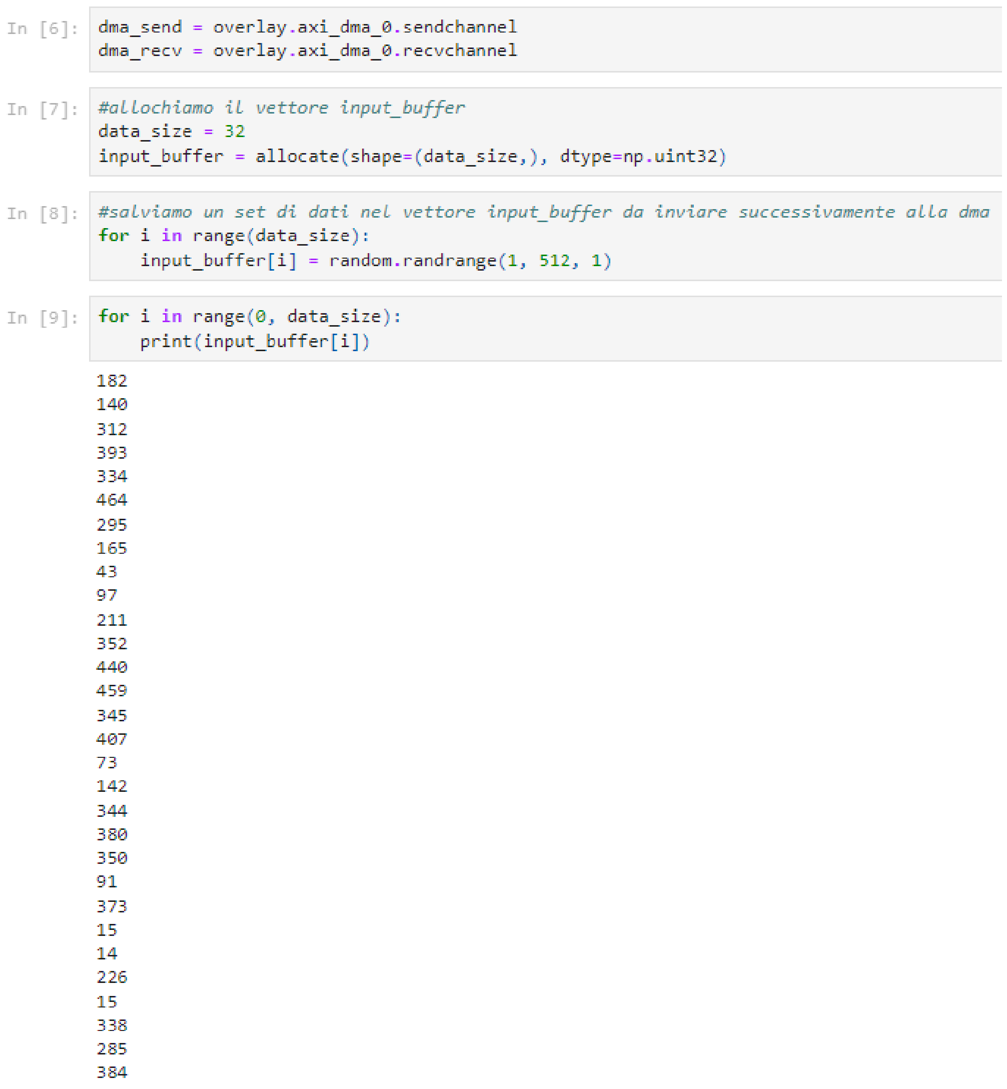
Worst Pulse Width Slack (WPWS):	2587.225 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	12045

All user specified timing constraints are met.

Figure 9: Timing requirements

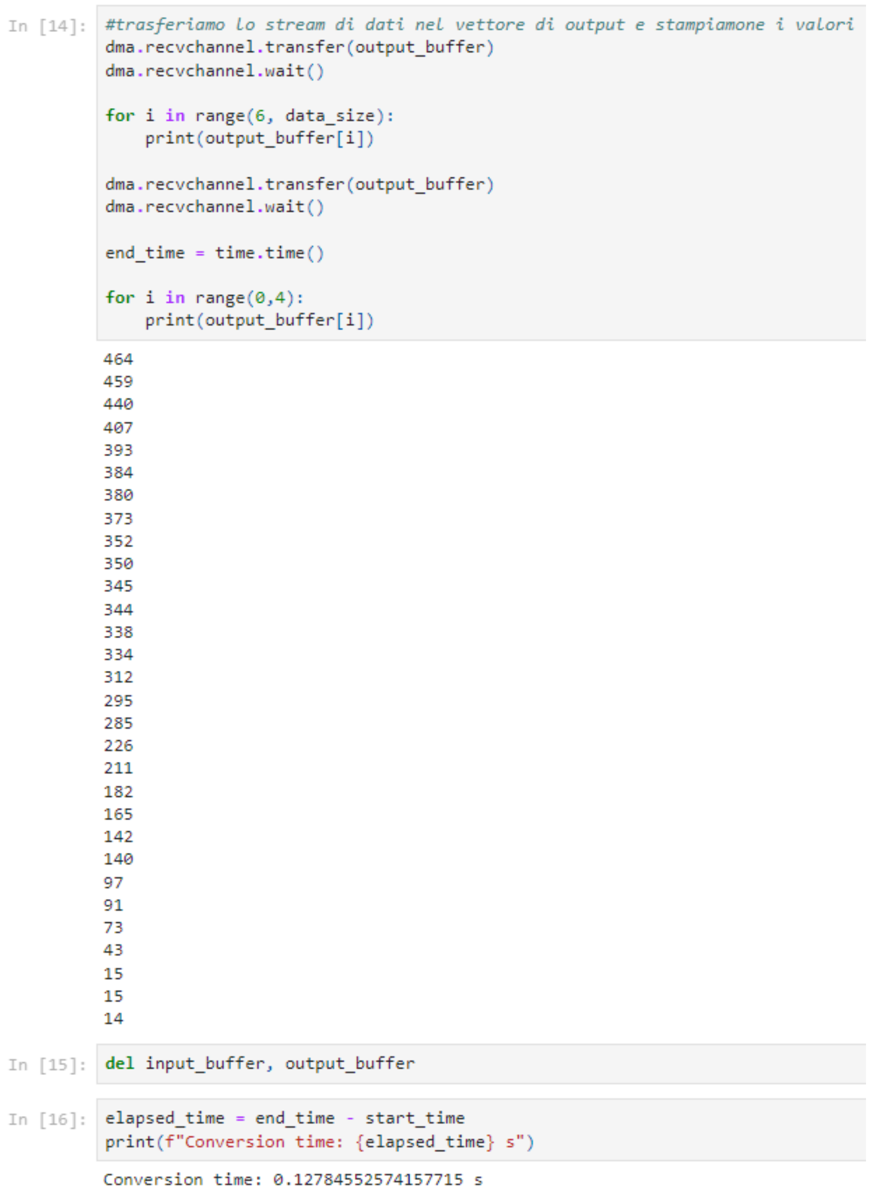
2.5 Jupiter Environment

Working with the DMA (Direct Memory Access) is highly efficient, especially when coding in Python. The process is streamlined: the unordered vector enters the DMA, and subsequently, we can retrieve the output buffer containing the ordered elements.



```
In [6]: dma_send = overlay.axi_dma_0.sendchannel
        dma_recv = overlay.axi_dma_0.recvchannel

In [7]: #allochiamo il vettore input_buffer
        data_size = 32
        input_buffer = allocate(shape=(data_size,), dtype=np.uint32)

In [8]: #salviamo un set di dati nel vettore input_buffer da inviare successivamente alla dma
        for i in range(data_size):
            input_buffer[i] = random.randrange(1, 512, 1)

In [9]: for i in range(0, data_size):
            print(input_buffer[i])

        182
        140
        312
        393
        334
        464
        295
        165
        43
        97
        211
        352
        440
        459
        345
        407
        73
        142
        344
        380
        350
        91
        373
        15
        14
        226
        15
        338
        285
        384
```

Figure 10: DMA instruction and Input Buffer



```
In [14]: #trasferiamo lo stream di dati nel vettore di output e stampiamone i valori
         dma.recvchannel.transfer(output_buffer)
         dma.recvchannel.wait()

         for i in range(6, data_size):
             print(output_buffer[i])

         dma.recvchannel.transfer(output_buffer)
         dma.recvchannel.wait()

         end_time = time.time()

         for i in range(0,4):
             print(output_buffer[i])
```

```
464
459
440
407
393
384
380
373
352
350
345
344
338
334
312
295
285
226
211
182
165
142
140
97
91
73
43
15
15
14
```

```
In [15]: del input_buffer, output_buffer
```

```
In [16]: elapsed_time = end_time - start_time
         print(f"Conversion time: {elapsed_time} s")
```

```
Conversion time: 0.12784552574157715 s
```

Figure 11: Output Buffer

It can be seen that it is also possible to calculate the overall processing time for the entire operation.

2.6 LED Indicator Implementation

As we have seen before, to provide visual feedback on completion, an LED indicator system was implemented. However, while printing the python output, few times a bug jumps out: one number is not correctly reordered, indicating a flaw in the implementation. Debugging efforts are ongoing to identify and rectify the issue. The initial version without led is bugs free.

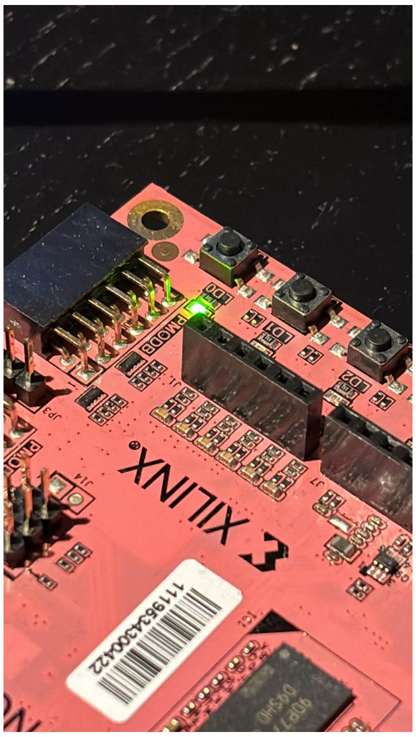
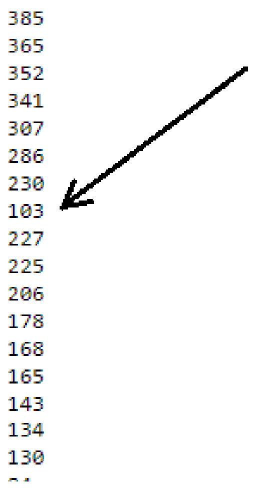


Figure 12: LD0 On when conversion ends



```
385
365
352
341
307
286
230
103
227
225
206
178
168
165
143
134
130
```

Figure 13: Sorting issue

3 Conclusion

In conclusion, it's evident that further enhancements are required for optimal performance. A new "tree" merge sort algorithm must be implemented to leverage parallelism effectively. Under this proposed system, each number is stored in a FIFO, which then enters a sorter. Subsequently, the sorted data enters another FIFO, where it merges with the data from the previous FIFOs, ensuring the preservation of order. However, it's worth noting that this approach may result in a larger number of clock cycles due to the increased complexity of the sorting process.

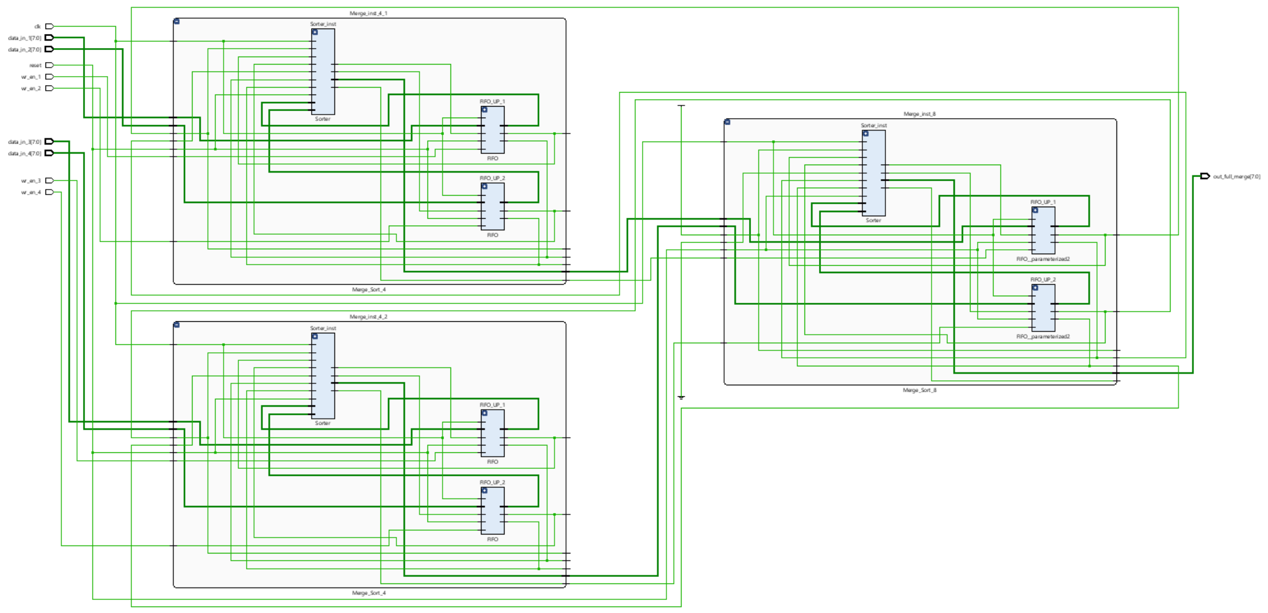


Figure 14: Sorting Tree

By repeating this structure 512 times, we can cover all input numbers. The number of these building blocks decreases by a power of two with each stage completion. Regrettably, while this structure functions correctly in behavioral simulation, it encounters signal issues during post-synthesis functional simulation.

Nonetheless, it holds promise for improving efficiency and scalability in sorting operations.