
UNIVERSITÀ DI PISA

Computer Engineering

Electronics and Communication Systems

# SORTING ALGORITHM HARDWARE ACCELERATOR

## VHDL description and Xilinx Vivado Synthesis

---

*STUDENTS:*
Lorenzo Cima
Simone Pampaloni

Academic Year 2020/2021

**Abstract**

The aim of this project is to demonstrate the knowledge of the VHDL language to design a digital circuit and the use of Xilinx Vivado tool to synthesize this circuit, and analyze the results.

# Contents

# 1 | Introduction

Sorting algorithms have been extensively studied in recent years, Taking into consideration some famous ones and in particular their complexity.

**Selection sort:** At any iteration it searches, from the elements of the vector not ordered yet, the minimum. Its complexity is $\mathbf{O(n^2)}$ in every case.

**Bubble sort:** It compares couple of elements to order them. Its complexity is $\mathbf{O(n^2)}$ in every case.

**Quick sort:** This algorithm starts dividing the vector in two sub-arrays and selecting a pivot. Then, the elements below the pivot are put left to it and the elements above the pivot right to it. Its complexity is generally $\mathbf{O(n^2)}$, but $\mathbf{O(n \log n)}$ in the best case, this depends principally by the selection of the pivot number.

**Merge sort:** It follows the "Divide et Impera" approach, dividing and merging the vector at every iteration. Its complexity is $\mathbf{O(n \log n)}$ in every case.

**Insertion Sort:** This algorithm takes one by one the vector elements and put it in the right ordered position. Its complexity is generally $\mathbf{O(n^2)}$, but $\mathbf{O(n)}$ in the best case, depending of the initial values of the vector

As we can see nowadays no algorithms exists with complexity $\mathbf{O(n)}$ in every case (the insertion sort do it only in the best case), if they run in a single core. So, instead of use software for sorting, sometimes is convenient to do this operation in hardware, with programmable logic, that allows to improve performances.

# 2 | Possible Applications

Sorting operation is one of the most common and probably most often performed operation and is the bottleneck of many computing problems. It is crucial part of many algorithms realized by data computing machines. Having a hardware accelerator for sorting can bring about big advantages, in particular in some specific areas:

**Database Systems:** sorting is particularly important in data collection solutions i.e. database systems where several indexing techniques allow fast searching and rapid access to saved data. To build any index system sorting algorithms are required. At present when huge number of data are processed and stored (Internet servers for example) efficient sorting seems to gain bigger importance than ever.

**Large-Scale Systems:** with the rise of big data applications, efficiency has become more important for data processing, and simple sorting methods require higher efficiency in large-scale scenarios.

**Variuos Application Scenarios:** in general, sorting hardware accelerator could bring big advantages in many application scenarios where sorting is a fundamental function as pattern recognition, image processing, multi-media processing (video, audio, etc.), signal processing, and high-energy physics.

# 3 | Possible architectures

A sorting hardware accelerator can be implemented with several different architectures, in this chapter we will discuss the most important ones.

## 3.1 Sorting Algorithm Hardware-Implemented

This architecture is the direct implementation of one of the sorting algorithms. This architecture needs M clocks (as many as the number of objects to be sorted) to save the objects in registers and other clocks necessary to execute the sorting algorithm implemented in hardware.

This type of architecture is very dependent on the sorting algorithm that has been chosen to implement in hardware, but above all it is not possible to reach a complexity of **O(n)**.

## 3.2 Sorting During Insertion

This architecture is based on sorting the numbers as they are inserted in the cells, thus eliminating the M clocks needed for loading in the registers of the previous architecture. This solution is composed of M stage, initially empty. At each clock a new number is inserted in the cell and is placed on the right stage following the current order. So, by ordering during insertion, it is possible to reach a complexity of **O(n)**. In fact the worst case requires 2M clocks (M to inserting in the system all the elements and M to put the last element at the end of the vector)

## 3.3 Pipeline

This architecture is an improvement of the previous one. It is obtained by inserting pipeline registers after each stage as shown in the figure.

The principal improvements in using a pipeline architecture are an higher clock frequency thanks to reduced amount of logic between two registers and so an higher throughput.

# 4 | Design

## 4.1 Architecture

The general scheme of the cell interface is the following:

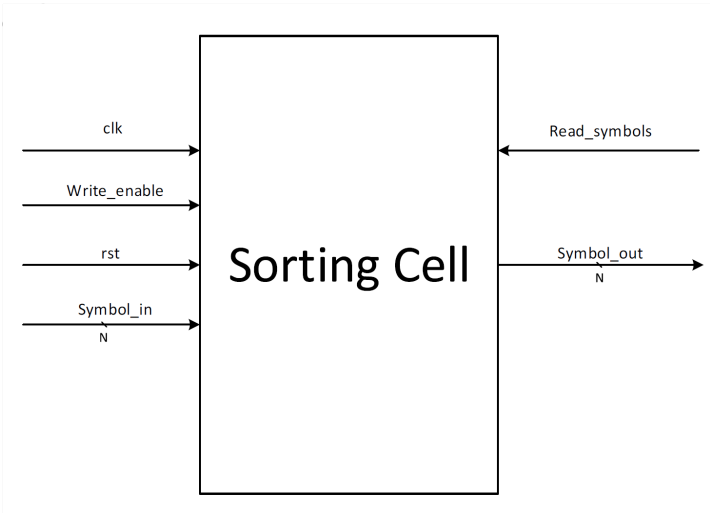


**Figure 4.1:** Input/Output Interface Scheme

The SortingCell has to keep memory of the last M elements received at the input and to sort them inside according to ascending order considering the inputs (of N bits) as unsigned. The input symbols are red and processed by the SortingCell only if the *write_enable* signal is high. The procedure for reading the ordered symbols at the output is activated when *read_symbol* is asserted for a clock cycle, and consists in reading all the elements, present and sorted within the block, in ascending order. Furthermore, the cell must manage a conflict of simultaneous reading and writing, any overflow of the M numbers contained within and any request for reading without there being no numbers inside the cell.

Our choice for the implementation of sorting algorithm hardware accelerator is a **Sorting During Insertion Architecture**. The most logical choice would be to use **Pipeline Architecture** for its improvements of high clock frequency and, as direct consequence, of high throughput. However, the constraints imposed in the cell construction (read/write concurrency and overflow/empty cell management) do not allow to fully exploit the

advantages of the pipeline architecture because they require additional logic external to the pipeline, which will therefore creates a bottleneck for the speed of the entire system.

The solution adopted is to compose the cell of two modules, a module that manages the various constraints **Sorting Cell Module** and a module with pipeline architecture that manages the actual ordering **Pipeline Sorting Stage Module**.

### 4.1.1 Sorting Cell Module

The sorting cell module represents the main module of the cell. Its job is to manage all constraints. In particular, it has to manage:

- **Writing request during a reading:** The precedence is for the writing. The data read request that occurs during the writing phase (*read_symbols* for one clock cycle) is saved and pending the end of the writing. Then, when it is finished (*write_enable* set to 0), the reading is performed without request symbols another time. This rule is valid also if writing and reading requests arrive at the same clock.
- **Reading request during a writing:** In this case, before the end of reading, the writing request is not considered. Numbers in input are discarded.
- **Overflow:** When the array is full, new elements on input are discarded. This choice permits a very simple and fast management of the elements. One alternative approach can be to discard the oldest element, but it requires more logic which can impact significantly to the clock frequency, so it isn't used.
- **Reading for an empty vector:** In this case the system returns an array of M elements with default value(0).
- **Reading for a not-complete vector:** The system returns all the K elements of the array, followed by M-K zeros. This choice is been done because normally zeros stay at the begin of the vector when it is sorted, so when a reader sees zeros at the end, he knows that the array wasn't full and discards them.

### 4.1.2 Pipeline Sorting Stage Module

The other module manages the sorting. It is a pipeline of submodules, named Sorting Stage submodule. Each Sorting Stage submodule performs the real comparison between numbers.

It has as input a N-bit word *received_data*, that represents the input number to compare with the number contained in the stage (if present), and a *received_flag* signal which is set to 1 when a new number is submitted as


**Figure 4.2:** Sorting Stage Scheme

an input. It has as output a N-bit word *forwarded_data* that represents the result of the comparison between the number presented in input and the number present in the stage. The result is the larger of the two and it must be forwarded to the next stage. Signal *forward_flag* must be set to 1 when a new number has to be forwarded to the next stage. The output of each stage will be the input of the next one, the first stage will take as input the *Symbol_in*.

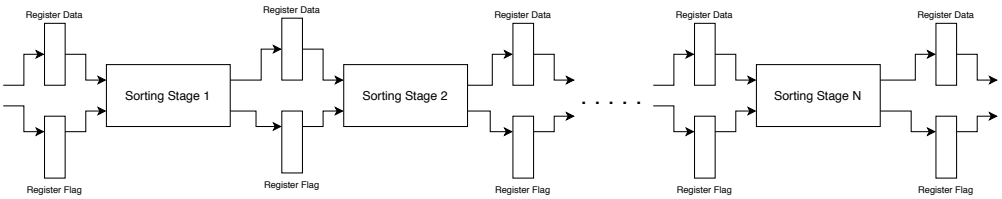


**Figure 4.3:** Block Diagram Pipeline

## 4.2 VHDL code

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  --this is the main module, which contains the pipeline
       ↪ sorting stages
6  -- and implements all the synchronization features
7
8  -- The generic N is the number of bits of the unsigned that
       ↪ we want to sort.
9  -- The generic M is the number of unsigned that the
       ↪ SortingCell can store.
10
11 -- SortingCell entity ------------------------------------
12
13 entity SortingCell is generic( M: INTEGER := 5; N: INTEGER
       ↪ := 8);
14
15   port(
```

```
16
17      write_enable: in std_logic; --when we want to insert a
            ↪ number to sort must be set to 1
18      symbol_in: in std_logic_vector(N-1 downto 0);
19
20      read_symbols: in std_logic; --when we want to read all
            ↪ sorted numbers must be set to 1 from one clock
            ↪ cycle
21
22      symbol_out: out std_logic_vector(N-1 downto 0);
23
24      clk: in std_logic;
25      rst: in std_logic
26    );
27  end SortingCell;
28  ----------------------------------------------------------
29
30  --SortingCell architecture ---------------------------------
31  architecture bhv of SortingCell is
32
33    -- stage component neeeded
34    component sorting_stage
35
36      generic(N: INTEGER := 8);
37      port(
38        received_data: in std_logic_vector(N-1 downto 0);
39        received_flag: in std_logic;
40
41        forwarded_data: out std_logic_vector(N-1 downto 0);
42        forward_flag: out std_logic;
43
44        clk: in std_logic;
45        rst: in std_logic
46      );
47    end component;
48
49    -- these arrays are the input/output for each pipeline
          ↪ stage
50    type data is array (0 to M-1) of std_logic_vector(N-1
          ↪ downto 0);
51    type flag is array (0 to M-1) of std_logic;
52
53    -- this array is for the reset of each pipeline stage
54    type reset is array (0 to M-1) of std_logic;
55
56    -- Signals
57    signal stage_data : data;
58    signal stage_flag : flag;
59    signal stage_rst : reset := (others => '0');
60
61    signal counter : integer := 0; --number of unsigned stored
          ↪  in sortingCell
62    signal actual_output : integer := 0; --useful for identify
          ↪  output to send out
63    signal data_requested: boolean := false; --flag useful
```

```
    ↪ when a reading request arrives, but sortingcell can
    ↪ not serve it immediately
  signal serving_data_request: boolean := false; -- state
    ↪ representing sending out sorted unsigned
  signal reading_data : std_logic := '0'; -- state
    ↪ represents the storage of new data

begin

  -- port mapping ----------------------------------------
  sorting_pipeline: for stage in 0 to M-1 generate

    -- first pipeline stage
    first_stage: if stage = 0 generate
      begin frist_stage : sorting_stage port map (  clk =>
        ↪ clk,
                rst => stage_rst(0),
                received_data => symbol_in,
                received_flag => reading_data,
                forwarded_data => stage_data(stage),
                forward_flag => stage_flag(stage));
    end generate first_stage;

    -- last pipeline stage
    last_stage: if stage = M-1 generate

      begin last_stage : sorting_stage port map (   clk =>
        ↪ clk,
                rst => stage_rst(stage),
                received_data => stage_data(stage -1),
                received_flag => stage_flag(stage -1),
                forwarded_data => stage_data(stage));
    end generate last_stage;

    -- other pipeline stages
    normal_stage : if (stage /= 0) and (stage /= M-1)
      ↪ generate

              begin regular_cells : sorting_stage port map
                ↪  (  clk => clk,
                rst => stage_rst(stage),
                received_data => stage_data(stage -1),
                received_flag => stage_flag(stage -1),
                forwarded_data => stage_data(stage),
                forward_flag => stage_flag(stage));
            end generate normal_stage;
  end generate sorting_pipeline;
  -------------------------------------------------------


  -- each time the current value of a stage is sent out as
    ↪ output of the sorting cell,
  -- the stage is resetted (set state to empty and currrent
    ↪ value equal to 0) putting its rst to 0
  rst_setup: for k in 0 to M-1 generate
```

```
    stage_rst(k) <= '0' when( rst = '0' or (
       ↪ serving_data_request and actual_output = k)) else
       ↪ '1';
  end generate rst_setup;

  -- sorting cell accepts data in input if write_enable is
     ↪ set to 1, sorting cell is not full and when
  -- it is not send out data as output
  reading_data <= '1' when ( write_enable = '1' and counter
     ↪ /= M and not serving_data_request) else '0';

  -- if sortingcell is in "serving_data_request" state, it
     ↪ sends out as output a current value of a stage for
  -- each clock cycle
  symbol_out <= stage_data(actual_output) when (
     ↪ serving_data_request) else (others => 'Z');

  --actual output represents the stage index whose value is
     ↪ to be sent out as output.
  --At each clock, as long as we are in the "
     ↪ serving_data_request" state, its value is increased.
  actual_output_state: process(clk)
  begin
    if rising_edge(clk) then
      if rst = '0' then
        actual_output <= 0;
      elsif serving_data_request then
        actual_output <= actual_output +1;
      else
        actual_output <= 0;
      end if;
    end if;
  end process actual_output_state;


  output_state: process(clk)
  begin
    if rising_edge(clk) then
      if rst = '0' then
        counter <= 0;
        data_requested <= false;
        serving_data_request <= false;
      elsif reading_data = '1' then -- if sortingcell is in
         ↪ "reading_data" state, it is accepting new data
        counter <= counter +1; --so counter is incremented
        if read_symbols = '1' then -- if sortingcell is
           ↪ storing new data and read_symbols is equal to
           ↪ 1
          data_requested <= true; -- data_requested flag is
             ↪ set to 1 -> pending request
        end if;
      --if sortingcell is not storing new data and,
         ↪ read_symbols is equal to 1 or there is a pending
         ↪  request (data_requested is true)
      -- sortingcell passes into the state of "
```

```
          ↪ serving_data_request"
150       elsif reading_data = '0' and (data_requested or
             ↪ read_symbols = '1') then
151         serving_data_request <= true;
152         data_requested <= false;
153       elsif actual_output = M -1 then --when all the M
             ↪ unisgned are sent out, sortingcell exits from
             ↪ the state of "serving_data_request"
154         serving_data_request <= false;
155         counter <= 0;
156       end if;
157     end if;
158   end process output_state;
159
160
161 end bhv;
```

**Listing 4.1:** Code for Sorting Cell

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  -- This module is a stage of the pipeline that makes the
      ↪ sorting.
6  -- The generic N is the number of bits of the unsigned that
      ↪ we want to sort.
7
8  --------Sorting Stage Entity -------------------------------
9  entity sorting_stage is generic(N: INTEGER := 5);
10   port(
11     received_data: in std_logic_vector(N-1 downto 0);
12     received_flag: in std_logic; -- '1' when previous stage
          ↪ wants to send data
13
14     forwarded_data: out std_logic_vector(N-1 downto 0);
15     forward_flag: out std_logic; -- '1' when this stage
          ↪ wants to send data to next stage
16
17     clk: in std_logic;
18     rst: in std_logic
19   );
20 end sorting_stage;
21
22 ---------------------------------------------------------
23
24 ------- Sorting Stage Architecture ------------------
25 architecture bhv of sorting_stage is
26
27   signal empty : boolean := true; -- when empty is true, the
       ↪  stage does not contain any unsigned value
28   signal current_data : std_logic_vector (N-1 downto 0) := (
       ↪ others => '0');
29 begin
30
```

```
31  state: process(clk)
32  begin
33    if rising_edge(clk) then
34      if rst = '0' then
35        empty <= true;
36      elsif received_flag = '1' then -- a new unsigned is
            ↪ arrived from previous stage
37        empty <= false;
38        end if;
39    end if;
40  end process state;
41
42  flag: process(clk)
43  begin
44    if rising_edge(clk) then
45      if rst = '0' then
46        forward_flag <= '0';
47      elsif received_flag = '1' and not empty then --if it
            ↪ receives data and it is not empty
48        forward_flag <= '1'; -- it forwards data, so forward
              ↪  flag is set to 1
49      else
50        forward_flag <= '0'; -- else, it set to 0
51      end if;
52    end if;
53  end process flag;
54
55  data: process(clk)
56  begin
57    if rising_edge(clk) then
58      if rst = '0' then
59        current_data <= (others => '0');
60        forwarded_data <= (others =>'0');
61      elsif received_flag = '1' then
62        case empty is
63          when true => --when it receives data and it is
                ↪ empty, current data become the received data
64            current_data <= received_data;
65          when false => -- when is not empty
66            if unsigned(received_data) < unsigned(
                  ↪ current_data) then -- if received data is
                  ↪ less than current data
67              forwarded_data <= current_data; -- stage
                    ↪ forwards its current data
68              current_data <= received_data; -- and received
                    ↪  data become the current data
69            else
70              forwarded_data <= received_data; --else the
                    ↪ received data is forwarded
71            end if;
72        end case;
73      else
74        forwarded_data <= current_data;
75      end if;
76    end if;
```

```
77    end process data;
78
79 end bhv;
```

**Listing 4.2:** Code for Sorting Stage

# 5 | Test plan

In the sorting hardware accelerator there are some particular events, managed as described in Chapter 4 “Design”

- **Writing request during a reading**
- **Reading request during a writing**
- **Overflow**
- **Reading for an empty vector**
- **Reading for a not-complete vector**

To check the coherence between what is declared on the design and to test the correctness of the overall system's implementation, we have to extract the output through a simulation with the testbench shown in Listing 5.1. In this phase we analyze for visual comfort the situation with minimum number of bits N = 8 and very little vector dimension M = 5 (not used later into synthesis and implementation).

## 5.1 Example of algorithm

We can see an example of the hardware algorithm (Figure 5.1), using M=5. Notice that the number of clocks required is 11 and it corresponds to the worst case, which happens when the maximum value is the last number red

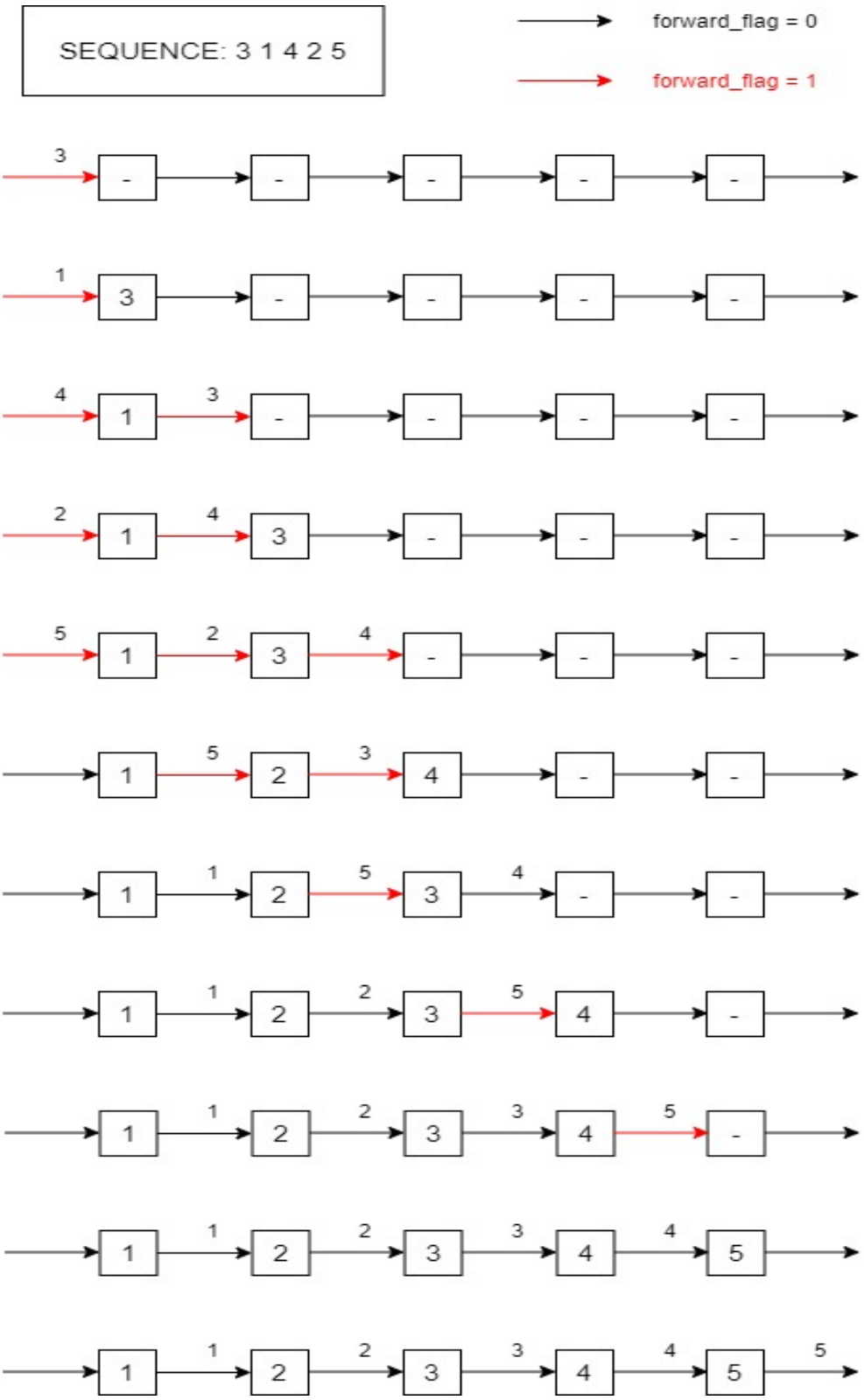


**Figure 5.1:** Example of algorithm

## 5.2 VHDL Code

```
library IEEE;
use IEEE.std_logic_1164.all;
use STD.textio.all;
use IEEE.std_logic_textio.all;
use IEEE.numeric_std.all;

-- TestBench

--sortingcell_tb entity ---------------------------------
entity sortingcell_tb is
end sortingcell_tb;
-------------------------------------------------------

--sortingcell_tb architecture ----------------------------

architecture bhv of sortingcell_tb is

--constants
constant N : integer := 8;
constant M : integer := 5;

constant T_CLK : time := 10 ns;
constant T_RESET : time := 50 ns;
--

--signals
signal clk : std_logic := '0';
signal rst : std_logic := '1';

signal write_enable : std_logic := '0';
signal symbol_in : std_logic_vector(N-1 downto 0);

signal read_symbols : std_logic := '0';
signal symbol_out : std_logic_vector(N-1 downto 0);


signal end_sim : std_logic := '1';
--

--component
component SortingCell is generic( M: INTEGER ; N: INTEGER );

  port(

    write_enable: in std_logic;
    symbol_in: in std_logic_vector(N-1 downto 0);

    read_symbols: in std_logic;

    symbol_out: out std_logic_vector(N-1 downto 0);

    clk: in std_logic;
```

```
    rst: in std_logic
  );
end component SortingCell;
--

begin


  clk <= (not(clk) and end_sim) after T_CLK/2;
  rst <= '1' after T_RESET;

--port mapping
  dut: SortingCell generic map( N => N, M => M)

    port map ( write_enable => write_enable,
      symbol_in => symbol_in,
      read_symbols => read_symbols,
      symbol_out => symbol_out,
      clk => clk,
      rst => rst);
--

  d_process : process(clk)
    variable t : integer := 0;
  begin
    if(rst = '0') then
      write_enable <= '0';
      read_symbols <= '0';
      end_sim <= '1';
      symbol_in <= (others => 'X');
      t := 0;
    elsif(rising_edge(clk)) then
      case(t) is
        when 1 => symbol_in <= std_logic_vector(to_unsigned(
            ↪ 3,8)); write_enable <= '1';    --Start to
            ↪ write (5 numbers)
        when 2 => symbol_in <= std_logic_vector(to_unsigned(
            ↪ 1,8));
        when 3 => symbol_in <= std_logic_vector(to_unsigned(
            ↪ 3,8));
        when 4 => symbol_in <= std_logic_vector(to_unsigned(
            ↪ 2,8));
        when 5 => symbol_in <= std_logic_vector(to_unsigned(
            ↪ 0,8));  -- Last element
        when 6 => symbol_in <= std_logic_vector(to_unsigned(
            ↪ 9,8));      -- Overflow: 9 is discarded
        when 7 => symbol_in <= (others => 'X'); write_enable
            ↪  <= '0';  -- Stop writing
        when 8 => read_symbols <= '1';  --Read request (
            ↪ starts after one clock and lasts for 5 clocks)
            ↪ . Sequence: [0 1 2 3 3]
        when 9 => read_symbols <= '0';  --The read request
            ↪ stay to 1 only for one clock every time
        when 10 => null;
```

```
96          when 11 => symbol_in <= std_logic_vector(to_unsigned
               ↪ (9,8)); write_enable <= '1'; --Write request
               ↪ during a reading (ignored)
97          when 12 => symbol_in <= (others => 'X');
               ↪ write_enable <= '0';
98          when 13 => null;  --End of reading. The vector of
               ↪ numbers is emptied (it contains now 5 zeros)
99          when 14 => read_symbols <= '1'; --Read request for a
               ↪  empty vector. Reads this sequence: [0 0 0 0
               ↪ 0]
100         when 15 => read_symbols <= '0';
101         when 19 => null;  --End of reading. The vector of
               ↪ numbers is emptied (it contains now 5 zeros)
102         when 20 => symbol_in <= std_logic_vector(to_unsigned
               ↪ (7,8)); write_enable <= '1'; read_symbols <='1
               ↪ '; --Both writing request and reading request:
               ↪  it writes until write_enable is 1, then it
               ↪ reads
103         when 21 => symbol_in <= std_logic_vector(to_unsigned
               ↪ (4,8)); read_symbols <= '0';
104         when 22 => symbol_in <= std_logic_vector(to_unsigned
               ↪ (9,8));
105         when 23 => symbol_in <= (others => 'X');
               ↪ write_enable <= '0'; --Finish to write. The
               ↪ next clock start to read with a non-full
               ↪ vector. Sequence: [4 7 9 0 0]
106         when 24 => read_symbols <= '1'; --Reading request
               ↪ when a reading is performed: ignored
107         when 25 => read_symbols <= '0';
108
109         when 30 => end_sim <= '0';  --End simulation
110         when others => null;
111       end case;
112       t := t+1;
113     end if;
114   end process;
115
116 end bhv;
117
118
119 ------------------------------------------------------------
```

**Listing 5.1:** Testbench for the system

## 5.3 Steps and results

In the first part of the simulation, shown in Figure 5.2 we can see the correct insertion of 5 elements (array full) and a sixth element (9), that must be discarded because of overflow rule. Later, a reading request is performed and the array is returned correctly sorted. Notice that, during the reading, a writing request arrives. According to the rules, it isn't considered.

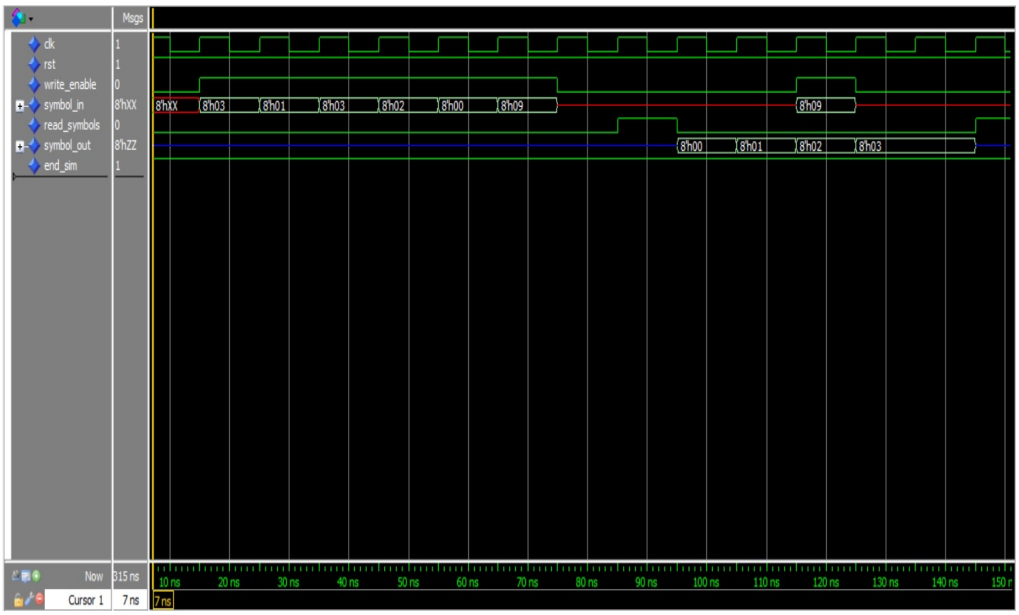


**Figure 5.2:** First part of the simulation

In the second part, shown in Figure 5.3 we have first a reading request on an empty vector, which is answered with M elements zero. Then, a reading and a writing request arrive simultaneously. According to the selected rules, the system gives privileges to the writing and, after that write enable goes to zero, it performs the reading. In this case the writing doesn't fill the array, so on output the systems shows first the inserted elements in order, and then the remaining elements set to zero. Notice that a new reading request arrives when the systems performs it yet, so it is ignored.

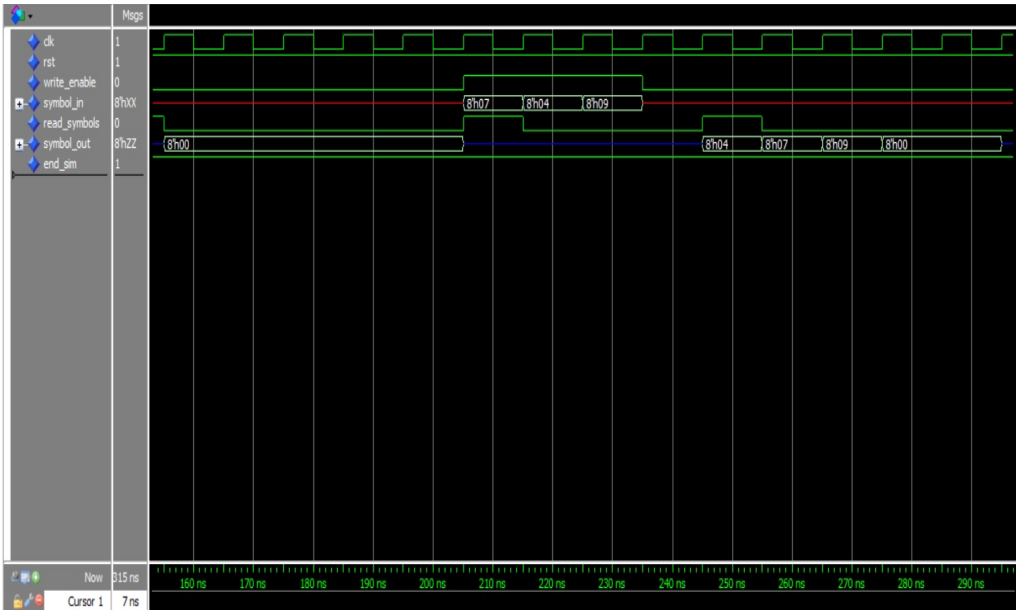


**Figure 5.3:** Second part of the simulation

# 6 | Synthesis

After the verification of the correctness of our VHDL model through ModelSIM we can now proceed with the synthesis and implementation on the Xilinx's ZyBo Development Board using Xilinx Vivado. A rough vision of the resulting system is shown in the Figure 6.1

In this section we present the results of our synthesis in terms of resource utilization, timing and power consumption.

Before to start the analysis of the results we want to clarify some assumption we have to make in order to correctly understand them:

- The results are obtained running first the *synthesis*, then the *implementation* on the board `ZYNQ XC7Z010-1CLG400C-1`, using the default synthesis and strategy provided by Vivado;
- The constraint file we have provided specifies only the desired clock frequency of $125MHz$, this frequency has been chosen without the purpose of a real implementation, just following the hints provided during laboratory lectures;
- The pin placement is completely left to Vivado, so pin assignment used are the default one;
- The maximum clock frequency obtained has been computed supposing that, after the implementation, Vivado has found the best solution possible using the default strategy.
- The illustrated values are shown with default settings of N=8 and M=10, which we consider as minimum considerable values for the use of this system

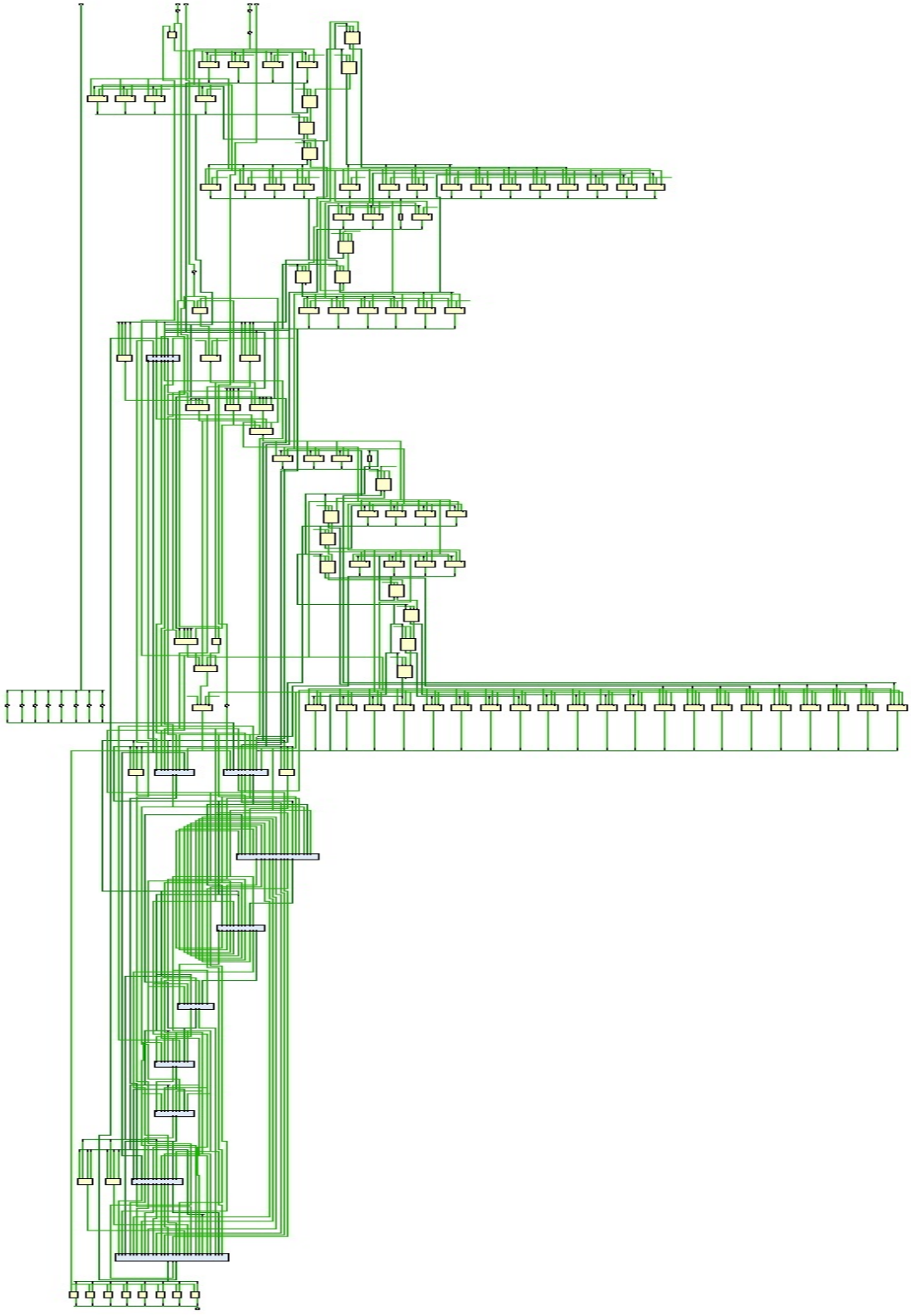

**Figure 6.1:** System architecture

## 6.1 Timing

The first thing we are going to analyze is the timing report, produced after the implementation step, that effectively place the components chosen during the synthesis on the board. The purposes for the timing are:

- *Avoid the setup violation.* This requirement is met when the WNS (Worst Negative Slack) is not negative
- *Avoid the hold violation.* This requirement is met when the WHS (Worst Hold Slack) is not negative

As we can see from the timing report in Figure 6.2, all the timing constraints are met

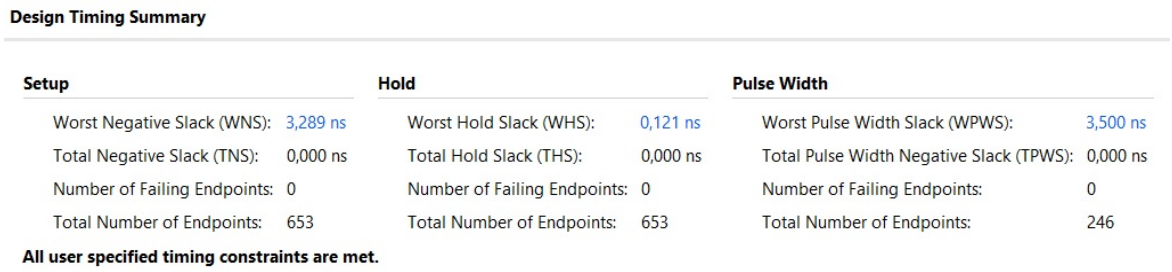


**Figure 6.2:** Timing report

This means that we can drive the board to an higher clock frequency than the specified one. More precisely we can compute the Maximum clock frequency $f_{\max}$ as in Equation (6.1)

$$f_{\max} = \frac{1}{T_{clk} - WNS} = 212,26MHz \tag{6.1}$$

Where $T_{clk} = 8ns$ that is the period corresponding to $125MHz$.

There are four equals (in time) **Max Delay Paths**, mostly determined by the net delay as we can see from the report in Figure 6.3. All these starts from a common counter register (the $31^{\text{st}}$) and arrive in different counter registers ($17^{\text{th}}$, $18^{\text{th}}$, $19^{\text{th}}$ and $20^{\text{th}}$). In the Figure 6.4 we show only one of this paths.

| Name | Slack ^1 | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | 3.289 | 4 | 47 | counter_reg[30]/C | counter_reg[16]/R | 4.227 | 0.952 | 3.275 | 8.0 | PL_clock |
| Path 2 | 3.289 | 4 | 47 | counter_reg[30]/C | counter_reg[17]/R | 4.227 | 0.952 | 3.275 | 8.0 | PL_clock |
| Path 3 | 3.289 | 4 | 47 | counter_reg[30]/C | counter_reg[18]/R | 4.227 | 0.952 | 3.275 | 8.0 | PL_clock |
| Path 4 | 3.289 | 4 | 47 | counter_reg[30]/C | counter_reg[19]/R | 4.227 | 0.952 | 3.275 | 8.0 | PL_clock |
| Path 5 | 3.296 | 4 | 47 | counter_reg[30]/C | counter_reg[24]/R | 4.218 | 0.952 | 3.266 | 8.0 | PL_clock |
| Path 6 | 3.296 | 4 | 47 | counter_reg[30]/C | counter_reg[25]/R | 4.218 | 0.952 | 3.266 | 8.0 | PL_clock |
| Path 7 | 3.296 | 4 | 47 | counter_reg[30]/C | counter_reg[26]/R | 4.218 | 0.952 | 3.266 | 8.0 | PL_clock |
| Path 8 | 3.296 | 4 | 47 | counter_reg[30]/C | counter_reg[27]/R | 4.218 | 0.952 | 3.266 | 8.0 | PL_clock |
| Path 9 | 3.330 | 3 | 18 | actual_output_reg[31]/C | sorting_pipe...ata_reg[0]/R | 3.980 | 0.828 | 3.152 | 8.0 | PL_clock |
| Path 10 | 3.330 | 3 | 18 | actual_output_reg[31]/C | sorting_pipe...ata_reg[1]/R | 3.980 | 0.828 | 3.152 | 8.0 | PL_clock |

**Figure 6.3:** Delay report

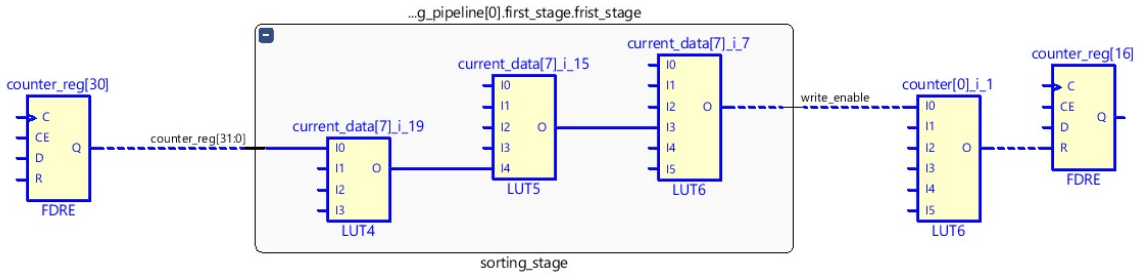


**Figure 6.4:** Maximum delay path

## 6.2 Utilization

The utilization report in Figure 6.5 shows that the system uses very few resources ($1,34\%$ of LUTs and $0,44\%$ of Flip Flops), with the only exception of IO pins, used for 20% of the available quantity (but this isn't a huge percentage). More precisely, the system uses 12 input pins and 8 output pins. Considering all the resources, the overall utilization percentage is $0,95\%$

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 235 | 17600 | 1.34 |
| FF | 245 | 35200 | 0.70 |
| IO | 20 | 100 | 20.00 |

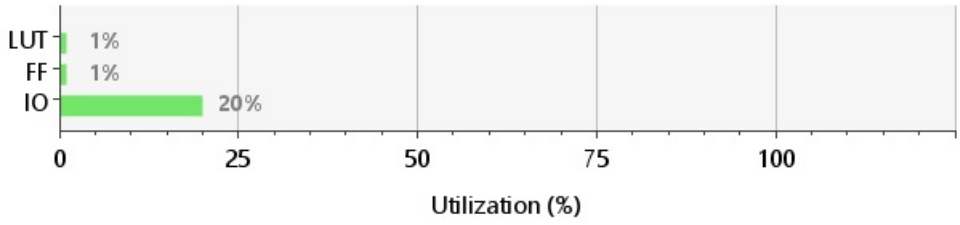


**Figure 6.5:** Overall utilization

We can see the amount of resources allocated for each stage on Figure 6.6. Values are very similar but we can see that the $7^{th}$ stage uses the highest number (46 total resources) and the $4^{th}$ the lowest (only 22). The particular aspect is that every slice LUT is used for logic (according to the Vivado default implementation), no one for memory.

| Name | Slice LUTs (17600) | Bonded IOB (100) | BUFGCTRL (32) | Slice Registers (35200) | Slice (4400) | LUT as Logic (17600) |
|---|---|---|---|---|---|---|
| SortingCell | 235 | 20 | 1 | 245 | 92 | 235 |
| sorting_pipeline[0].first_stage.frist_stage (sorting_stage) | 27 | 0 | 0 | | 14 | 27 |
| sorting_pipeline[1].normal_stage.regular_cells (sorting_stage_0) | 24 | 0 | 0 | | 10 | 24 |
| sorting_pipeline[2].normal_stage.regular_cells (sorting_stage_1) | 25 | 0 | 0 | | 10 | 25 |
| sorting_pipeline[3].normal_stage.regular_cells (sorting_stage_2) | 17 | 0 | 0 | | 10 | 17 |
| sorting_pipeline[4].normal_stage.regular_cells (sorting_stage_3) | 16 | 0 | 0 | | 6 | 16 |
| sorting_pipeline[5].normal_stage.regular_cells (sorting_stage_4) | 16 | 0 | 0 | | 7 | 16 |
| sorting_pipeline[6].normal_stage.regular_cells (sorting_stage_5) | 32 | 0 | 0 | | 14 | 32 |
| sorting_pipeline[7].normal_stage.regular_cells (sorting_stage_6) | 24 | 0 | 0 | | 11 | 24 |
| sorting_pipeline[8].normal_stage.regular_cells (sorting_stage_7) | 25 | 0 | 0 | | 13 | 25 |
| sorting_pipeline[9].last_stage.last_stage (sorting_stage_8) | 17 | 0 | 0 | | 8 | 17 |

**Figure 6.6:** Utilization for each pipeline

Finally, the Figure 6.7 shows the primitives used by the system. We can see that Flop and latch are the greatest number (245), followed by the LUTs (which are by different types). Not negligible is also the number of primitives for carry (26)

**Primitives**

| Ref Name | Used | Functional Category |
|---|---|---|
| FDRE | 245 | Flop & Latch |
| LUT4 | 176 | LUT |
| LUT6 | 42 | LUT |
| LUT3 | 29 | LUT |
| CARRY4 | 26 | CarryLogic |
| LUT5 | 23 | LUT |
| LUT2 | 17 | LUT |
| IBUF | 12 | IO |
| OBUFT | 8 | IO |
| LUT1 | 2 | LUT |
| BUFG | 1 | Clock |

**Figure 6.7:** Used primitives

## 6.3 Power Consumption

Our implementation, as we can see in Figure 6.8, has a power consumption of $99mW$, of which $6mW$ of *dynamic* power and $93mW$ of *static* power. The bigger part of dynamic power is consumed by the IO ports (41%). Notice that the overall confidence level of the computations is low, so numbers can be different from those. The Figure 6.9 shows that this is manly due for the contribute of IO computations (remember that we haven't assigned pins but it is done by Vivado from default settings)
This small value for the dynamic power can be explained by the low number of elements for the vector (the value M) used for experiments.

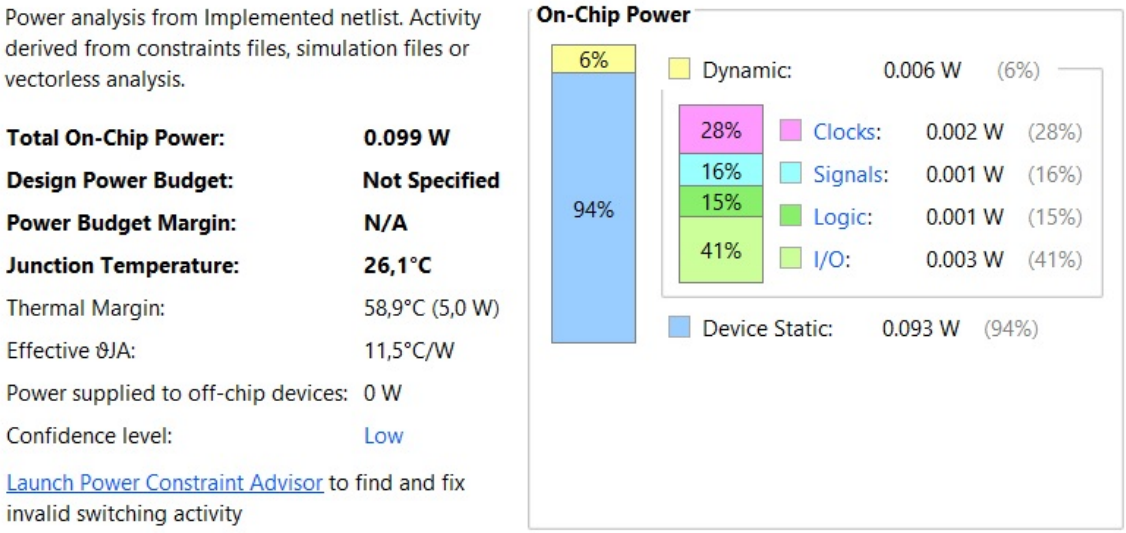


**Figure 6.8:** Power Consumption Report

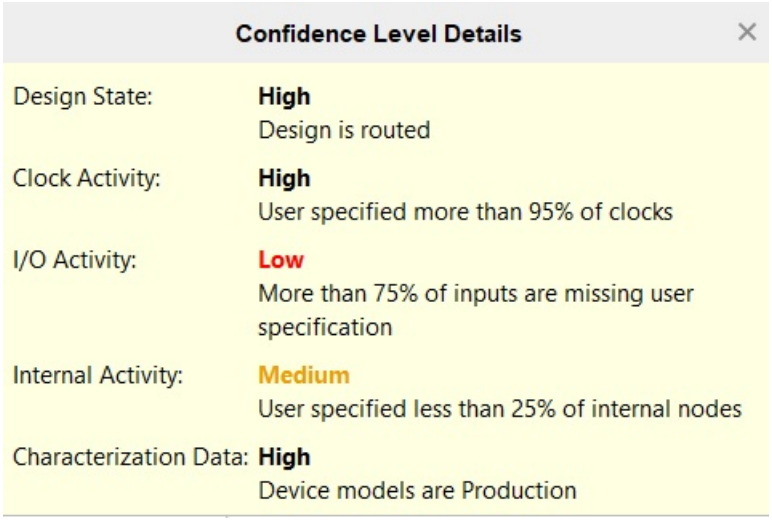


**Figure 6.9:** Confidence Levels of the Estimations

## 6.4 Warning Messages

During the synthesis and implementation processes, VIVADO has reported some warnings. We will now explain the reason for each one of them.

**Constraints 18–5210: No constraints selected for write**
As said during laboratory lectures, this warning can be ignored.

**ZPS7–1 Warning PS7 block required**
This warning arise because no pin placement specifications have been provided to the tool (VIVADO set himself the default ones)

# 7 | Behaviour Using Different Parameters

In the previous example we have analyzed the behaviour of the systems using as parameters N = 8 and M = 5. In this section we show how values for timing, utilization and power change when we increase those parameters. We can divide the experiments in three different section, represented in Figure 7.1:

- **Variable N and fixed M = 10:** changing the number of bits for the representation of values we can see that, as we can expect, there is an increasing in an almost linear way of maximum delay, power consumption (especially the dynamic one), utilization and a lose of performance. Notice that the Zybo board used doesn't allow 64 bit representations (IO ports are not enough)
- **Variable M and fixed N = 8:** changing the dimension of the vector, we can see that the behaviour of other experiments is the expected one, with an increasing in a way which is almost linear of maximum delay, power consumption (only the dynamic one) and utilization. The maximum clock is decreased. A quite different behaviour can be seen with the configuration M = 500, because the maximum delay is less than the one of M = 100 and the difference between the two clock frequency is very small. This can be caused by the selected implementation made by Vivado, which probably is very different with respect to the other configurations.
- **Analysis with maximum experiment's values:** In this final section, we first try a configuration with 32 bit and 500 elements, but the Zybo Board doesn't provide the necessary number of LUTs or Flip Flops, so we try to decrease the vector dimension to 100. This shows that the application works in an acceptable way also for big configurations (into the limits of the board), in fact we have a maximum clock frequency of 136,407 MHz and a power consumption which isn't so huge.

| N | M | Max CLK Frequency | Max Delay | Static PC | Dynamic PC | IO Utilization | Other Utilization |
|---|---|---|---|---|---|---|---|
| 8 | 10 | 212,26 MHz | 4,227 ns | 93 mW | 6 mW | 20% | 0,91% |
| 16 | 10 | 202,75 Mhz | 4,634 ns | 95 mW | 10 mW | 36% | 1,52% |
| 32 | 10 | 176,08 MHz | 5,338 ns | 95 mW | 17 mW | 68% | 2,63% |
| 64 | 10 | ND | ND | ND | ND | 132% | ND |
| 8 | 20 | 186,95 MHz | 4,667 ns | 93 mW | 10 mW | 20% | 1,59% |
| 8 | 50 | 159,74 MHz | 5,67 ns | 93 mW | 16 mW | 20% | 3,66% |
| 8 | 100 | 139 MHz | 7,04 ns | 93 mW | 33 mW | 20% | 7,61% |
| 8 | 500 | 137,55 MHz | 6,592 ns | 95 mW | 147 mW | 20% | 35,24% |
| 32 | 100 | 136,4 MHz | 7,173 ns | 96 mW | 99 mW | 68% | 24,73% |
| 32 | 500 | ND | ND | ND | ND | 68% | 121,24% |

**Figure 7.1:** Results of Experiments

# 8 | Conclusions

To conclude we make some consideration on our results.
Into the limits of the Zybo board used, we can see that the performances are good enough both with little and big configuration values. The power consumption is acceptable and especially the dynamic power is very reduced with a little number of array elements.
We have done the synthesis in a fully automatized way, so the tool has the full control. What can be done to improve the results obtained is to use other synthesis strategies provided by the tool (not the default one), for example we can use a strategy for high performance, or reduced power consumption, depending on the constraints of the system that want to use this hardware sorting accelerator.