MAPD : Bubble Sort Algorithm in VHDL

AUSILIO Lorenzo, PRODAN George, JAFARPOUR Farshad

Feb/July 2022

Abstract

The goal of this project is to write a Bubble sorting algorithm in VHDL that sorts an array of numbers, then implement it into our FPGA. We measure the performances of this algorithm by running several simulations. Overall we conclude that by making a comparison between the results of VHDL implementation to those obtained from Python it is shown that FPGAs are computationally faster.

1 Introduction

Sorting is one of the most basic operations you can apply to data. You can sort elements using various sorting algorithms like Quick Sort, Bubble Sort, Merge Sort, Insertion Sort, etc. Bubble Sort is the most simple algorithm among all these.

Bubble sort makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. The name of the algorithm comes from the fact that each number "bubbles" up at the location where it belongs.

If there are n items in the list, then there are $n-1$ pairs of items that need to be compared on the first pass. Note that if the largest value of the list is part of a pair, then it will be continually moved along until the pass is complete.

At the start of the second pass, from n items 1 has been sorted, so there will remain $n-1$ left to sort, which means also $n-1$ pairs to compare for that pass. After completing the $n-1$ passes, the smallest item must be in the correct position with no further processing required.

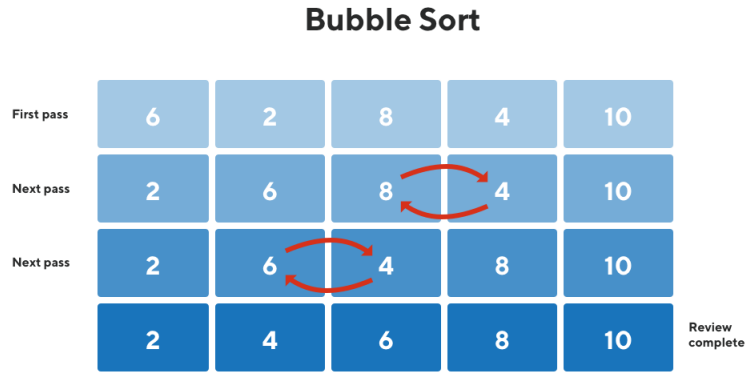

Figure 1: Visualization of Bubble sorting algorithm (image retrieved from [1])

The sum of the first n integers is $12n^2 + 12n$. The sum of the first $n-1$ integers is thus $12n^2 - 12n$. This is still $O(n^2)$ comparisons.On average we exchange half of the time, in the best case scenario no exchanges will be made and in the worst case every comparisons will lead to an exchange.

Of all sorting algorithms bubble sort is considered to be the most inefficient, since we do some exchanges before the final location of each number is known. Those exchanges can be considered as "wasted" and are costly in computation time, which explains the inefficiency of the algorithm.

Next we'll introduce the role of **UART** in data transmission-reception. UART stands for universal asynchronous receiver transmitter. It is a hardware device for asynchronous serial communication. It sends data bits one by one, from the least significant to the most significant, with a start and stop bit. The UART controller is responsible for receiving parallel data from host and transmitting that in serial fashion [2]. For communication between two devices, the system is composed of two units that both have a receiver (RX) and transmitter (TX). Each unit works with a certain clock frequency [3]. Both receiver and transmitter have internal clock signals that govern the changing logic levels. As implied by the name "universal asynchronous receiver/transmitter," the UART interface does not use a clock signal to synchronize the TX and RX devices. So how does the receiver know when to sample the transmitter's data signal? The clock generators of the two units are physically different and separated so that the actual frequencies of both will be a little different. Thus, the communication between the two units must be considered asynchronous [4].

A **finite state machine (FSM)** is a mechanism whose output is dependent not only on the current state but also on past input and output values. To create a time-dependent algorithm in VHDL or implement a computer program in an FPGA, this can usually be done by using a FSM. State machines in VHDL are clocked processes whose outputs are dictated by the value of a state signal. The state signal plays the role of an internal memory for what happened in the previous iteration.

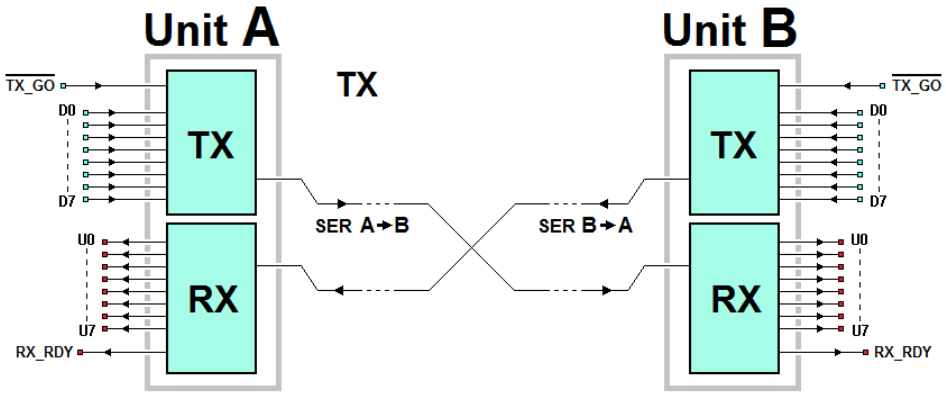

Figure 2: Asynchronous Serial Communication system. Image retrieved from [5]

The FPGA we used is the Arty-A7-100T [6].

2 Experimental Setup

In our project we use Vivado 2018.3 [7] to implement bubble sorting in VHDL [8]. We instantiate three components: the transmitter, the sorter and the receiver. They all are declared in top source file as:

```
component uart_transmitter is
    port (
      clock        : in  std_logic;
      array_sorted : in  ourArray;
      sorting_valid   : in  std_logic;
      busy         : out std_logic;
      uart_tx      : out std_logic);
  end component uart_transmitter;

  component sorter is
    port (
      clock        : in  std_logic;
      data_to_sort :
        in  std_logic_vector(7 downto 0);
      data_valid   : in  std_logic;
      sorting_valid : out std_logic;
      sorter_busy  : out std_logic;
      array_sorted  : out ourArray);
  end component sorter;

  component uart_receiver is
    port (
      clock         : in  std_logic;
      uart_rx       : in  std_logic;
      sorter_busy   : in std_logic;
      valid         : out std_logic;
      received_data :
        out std_logic_vector(7 downto 0));
  end component uart_receiver;
```

The sorter component performs the bubble sorting algorithm. A schematic of the VHDL project is presented in Figure 3.

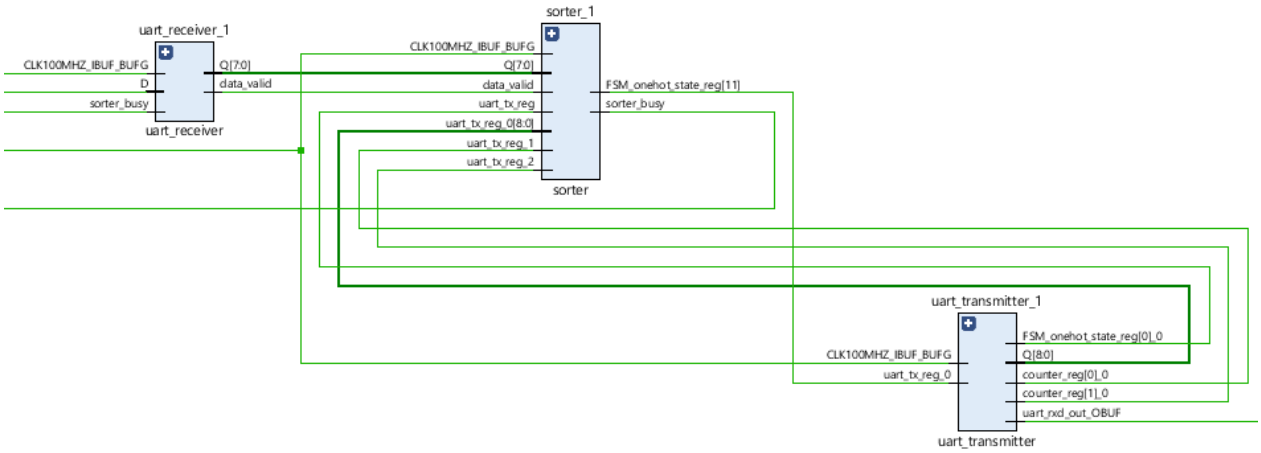

Figure 3: Synthesized Design - Schematic

To perform tests, we write a testbench file to simulate the process. A number is sent to UART receiver in this way:

```
wait until rising_edge(CLK100MHZ);
    uart_txd_in <= '0'; -- START    -- pentru 43
    wait for 8680 ns;
    uart_txd_in <= '1'; -- b0
    wait for 8680 ns;
    uart_txd_in <= '1'; -- b1
    wait for 8680 ns;
    uart_txd_in <= '0'; -- b2
    wait for 8680 ns;
    uart_txd_in <= '1'; -- b3
    wait for 8680 ns;
    uart_txd_in <= '0'; -- b4
    wait for 8680 ns;
    uart_txd_in <= '1'; -- b5
    wait for 8680 ns;
    uart_txd_in <= '0'; -- b6
    wait for 8680 ns;
    uart_txd_in <= '0'; -- b7
    wait for 8680 ns;
    uart_txd_in <= '1'; -- STOP
    wait for 8680 ns;
```

The receiver - transmitter system is working at a rate of 8 bits. This means that we are sorting only integer numbers smaller than 256. When programming the FPGA, the numbers can be received from a serial port opened using pyserial Python library:

```
import serial
ser = serial.Serial('COM5', baudrate=115200)
```

3 VHDL and Python Implementation

By using UART we receive the bits arrays for each number of the unsorted array. Then, they are stored in an array of *std_logic_vector*, which we defined it as:

```
package our_array_type is
    constant nb_size : integer := 8;
    constant array_size : integer := 4;
    type ourArray is array(0 to array_size - 1)
    of std_logic_vector (nb_size - 1 downto 0);
end package our_array_type;
```

We define the constant *nb_size* as the size of the bits array for each number, this size should correspond to the transmitter - receiver rate. Another constant is *array_size*, which is the length of the array that is sorted. The packages we use for our implementation are the following:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.our_array_type;
```

The sorter entity is responsible for handling the new numbers received by UART and the sorting. The numbers are stored in the signal array A. The *data_valid* signal will be triggered for each new number received. Anyway, the transmitter will remain in the *idle_s* state as long as the unsorted array is not completed and/or sorting is not needed. We monitor the sorting process by the signal *sorting_on*, which is '1' if the array A is still unsorted (even if the array is not complete yet). If sorting must be done, then the state machine jumps to the *sorting_state*. We use a counter to check if the array is completed or not. If the array is completed and sorting is not needed, then the machine will jump into the state *data_sorted* and its job is done for the moment.
Another signal, called *sorter_busy* is used to make sure that the sorter does not receive any number if the sorting is not done yet. While sorting, we check if the sorting is done by verifying if the numbers are in correct order. The sorter architecture is listed on the next page.

```
entity sorter is
  port (
    clock        : in  std_logic;
    data_to_sort :
    in  std_logic_vector(7 downto 0);
    data_valid   : in  std_logic;
    sorting_valid : out std_logic;
    sorter_busy: out std_logic;
    array_sorted  :
    out our_array_type.ourArray);
end entity sorter;
```

```
architecture Behavioral of sorter is

    type state_t is (idle_s, sorting_state, data_sorted, stop_s);
    signal state : state_t := idle_s;
    signal A: ourArray;
    signal sorting_on: std_logic;
    signal counter: integer := 0;
    signal sorting_done: std_logic := '0';
    signal new_data: std_logic := '0';

begin

  -- State Machine of the Sorter
  main_state_machine : process (clock) is
  begin  -- process main_state_machine
    if rising_edge(clock) then          -- rising clock edge
      case state is
        when idle_s =>
          sorter_busy <= '0';
          if data_valid = '1' then
                if counter < array_size - 1 then
                    counter <= counter + 1;
                else
                    counter <= 0;
                end if;
                A(counter) <= data_to_sort;
                new_data <= '1';
          end if;

          if new_data = '1' then
            sorting_on <= '1';
            sorting_done <= '0';
            sorter_busy <= '1';
            new_data <= '0';
            state <= sorting_state;
          else
            if sorting_done = '1' then
                if counter = 0 then
                    state <= data_sorted;
                else
                    state <= idle_s;
                end if;
            else
                state <= sorting_state;
            end if;
          end if;

        when sorting_state =>

            sorting_on <= '0';
            L1: for j in ourArray'LEFT to ourArray'RIGHT - 1 loop
                L2: for i in ourArray'LEFT to ourArray'RIGHT - 1 - j loop
                    if unsigned(A(i)) > unsigned(A(i + 1)) then
                                A(i) <= A(i + 1);
                                A(i + 1) <= A(i);
                                sorting_on <= '1';
                                exit L1;
                    end if;
                 end loop L2;
            end loop L1;

            if sorting_on <= '0' then
                sorting_done <= '1';
            end if;
            state <= idle_s;

        when data_sorted =>
          array_sorted <= A;
          sorting_valid <= '1';
          state <= idle_s;
        when others => null;
      end case;
```

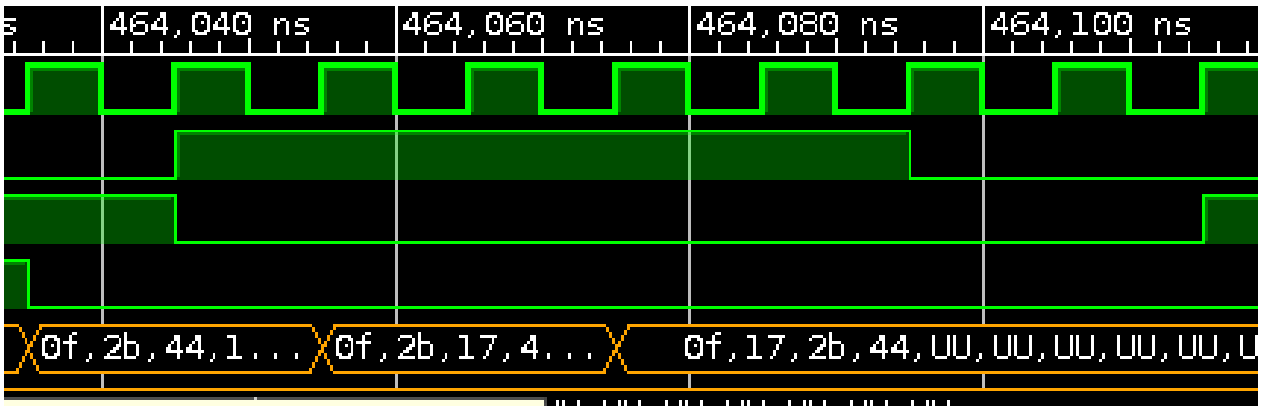

Figure 4: Sorter signal for time measurement

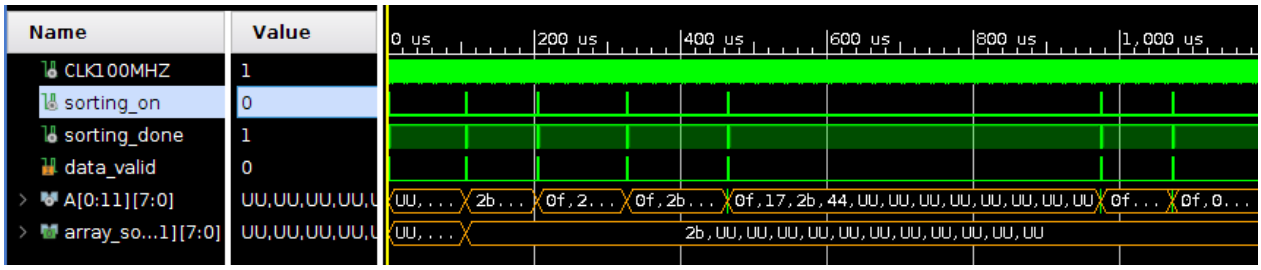

Figure 5: Overview of the simulation

```
    end if;                                                    74
  end process main_state_machine;                              75
end Behavioral;                                                76
```

The Python implementation of the same algorithm is the following:

```
                                                               1
                                                               2
sorted_array = unsorted_array.copy()                           3
for j in range(n - 1):                                         4
   for i in range(n - 1 - j):                                  5
       if sorted_array[i] > sorted_array[i + 1]:               6
           temp = sorted_array[i]                              7
           sorted_array[i] = sorted_array[i + 1]               8
           sorted_array[i + 1] = temp                          9
```

4 Performance evaluation

We run different simulations for arrays having the length between 2 and 32. In Table 1 we present the results of some of the simulations stating the array size, the clock cycles we counted and the total time taken for sorting. We measure the sorting time by using the *sorting_on* signal, which is turned on during the sorting state of the sorter FSM.
In Figure 4 one can observe the interval in which the sorting state of the sorter is active and, in the case presented, there can be counted 5 clock cycles (50 ns). Also, one can see how A is changing during sorting. An overall visualization of the simulation is presented in Figure 5.

We run the same tests but using a Python bubble sorting algorithm. We measure the time with *time* library. Finally, we plot the time measurements of both VHDL and Python implementations in the plot shown in Figure 6. We can observe that VHDL implementation run faster than the one in Python.

array length	clock cycles	time (ns)
4	10	100
6	18	180
8	32	320
12	66	660
20	139	1390
24	221	2210
32	346	3460

Table 1: Simulation results in VHDL

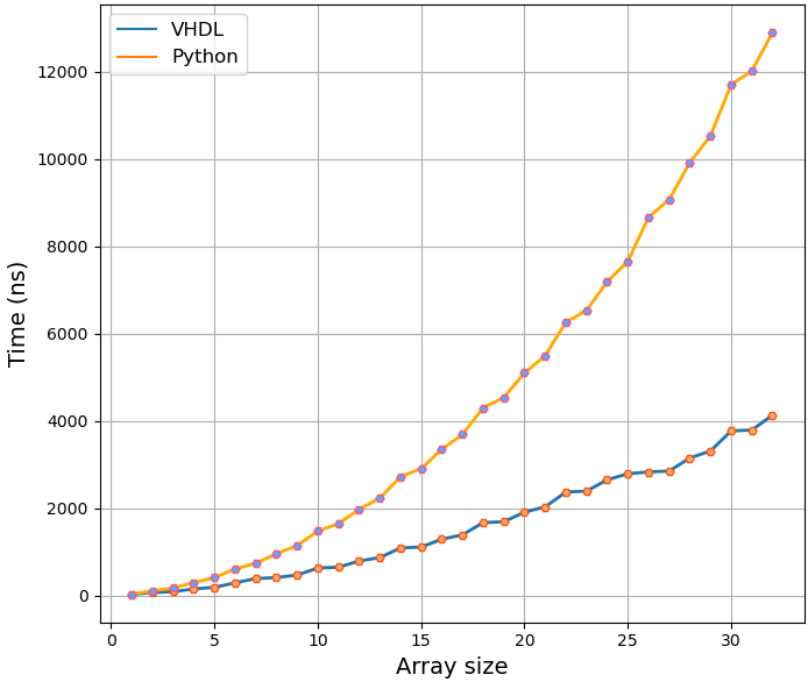

Figure 6: A comparison between VHDL and Python performances

References

[1] productplan.com.
`https://www.productplan.com/glossary/bubble-sort/`. Accessed: 2022-02-12.

[2] nandland.com.
`https://www.nandland.com/`. Accessed: 2022-02-18.

[3] microcontrollerslab.com.
`https://microcontrollerslab.com/uart-communication-working-applications/`. Accessed: 2022-02-19.

[4] allaboutcircuits.com.
`https://www.allaboutcircuits.com/technical-articles/back-to-basics-the-universal-asynchronous-receiver-`
Accessed: 2022-02-19.

[5] digitalelectronicsdeeds.com.
`https://www.digitalelectronicsdeeds.com/learningmaterials/LM/T050/050430_AsynchSerialTXRX_On_FPGA/Index.htm`. Accessed: 2022-02-19.

[6] J. Pazzini. G. Collazuol, A. Triossi. *MAPD Laboratory, academic year.* 2021 - 2022.

[7] Sanjay Churiwala. *Designing with Xilinx FPGAs: Using Vivado.* Springer Publishing Company, Incorporated, 1st edition, 2016.

[8] Mark Zwolinski. *Digital System Design with VHDL*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 2000.