FPGA BUBBLE SORT ALGORITHM IMPLEMENTATION AND TIMINGS

Group 4: Jake Jackson, Daniel Jordan, Walter Martemucci,
Mariam Hergnyan and Khadijatou Trawally.

ABSTRACT. This project aims to show how an Arty A7 FPGA can be utilized to sort data sent through a UART receiver. The optimized bubble sort algorithm is used as a test algorithm so we can gauge the efficacy of the FPGA through a series of timed array sorts over various array sizes. The FPGA performed comparably in speed to the same algorithm running in python on a medium range laptop. This is a significant result considering the smaller amount of computational resources, lower power and slower clock speed available to the FPGA. This result shows FPGA processors are worth implementing on efficiency critical operations.

Keywords: FPGA, VHDL, Algorithms, Bubble sort

1 Introduction

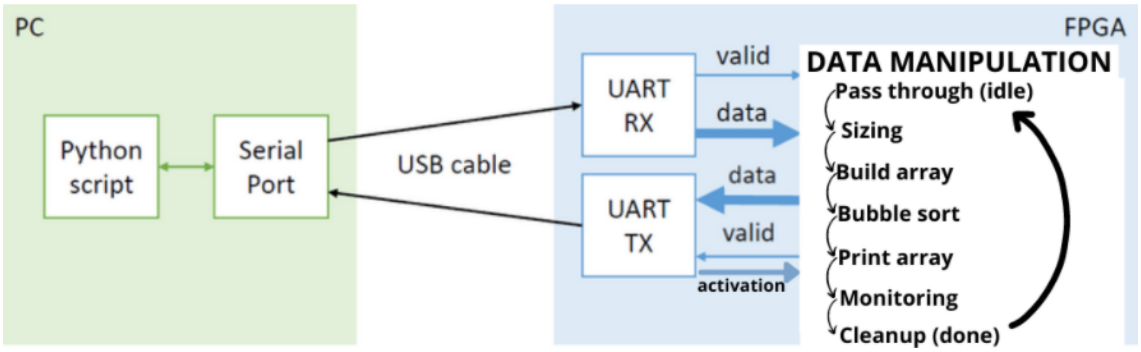

Figure 1: Project overview schematic

The goal of the experiment is to compare computational run times of the bubble sort algorithm on an FPGA device and an average laptop running python. The data to be used for the bubble sort will be transferred to the device using a standard serial port. The results of the bubble sort will be serially transferred back to the computer. These results include the sorted array and the number of clock cycles taken to complete the sort. The following sections will explain in detail the inner workings of each component programmed onto the FPGA. The performance of the FPGA is then compared to other sorting algorithms performed with python.

2 VHDL Implementation of UART and Bubble Sort

2.1 Top file

The top level entity implements the hierarchical design and the instances of the components. Also it defines port elements, each element listed in a port interface provides a channel for dynamic communication between a block and the environment. Signals are defined before the begin statement in the architecture structure of top and can be used in multiple processes.

2.2 Constraints file

The file contains physical and timing restrictions for the implementation. The first and most common are the pin assignments, a declaration to which physical FPGA pins the top level entity signals must be directed. Timing constraints set the boundaries for the propagation time to one logic element to another, an example is the clock constraint. It is of the utmost importance to specify the clock frequency in order to know how much time the system has to work with between clock edges.

2.3 Baud Rate Generator

The baud rate generator creates a pulse waveform (clocks) at desired baud rate (timing) for data transmission across (that sends data between) the transmitter and receiver wires. A frequency of 100MHz for a period of 10 ns was set as a constraint. We chose a desired baud rate of 115200 (bit/s), this was based on our receiver or transmitter clock speed. The clock cycle value was obtained from the divider equation:

$$Divider = \frac{frequency}{baud\ \ rate}$$

From the divider equation, we obtained 868 as our maximum clock cycles. Thus, for every 868 (clk) cycles at a delay of 10 ns, a single clock (i.e. uart clk) waveform of either '1' or '0' states is generated from the system clock.

2.4 Receiver

The receiver is designed to receive 8 bits of serial data, one start bit as 0, one stop bit as 1, and with no parity bit. It has i_clk and i_Rx_serial (serial data stream) as input pins, o_Rx _dv (data value pulse) and o_Rx_byte as output pins. When receive is complete o_Rx_dv will be raised high for one clock cycle. Once a bit is detected the i_Rx _serial goes low and clk count starts. The byte index will count from 0 up to 7 and the sampling line is incremented by one after every clk cycle until all the bits are received (i.e bit index of 7). This will assert o_Rx _dv to one for one clock cycle. The o_Rx_byte will contain the parallel received data in the form of a standard logic vector.

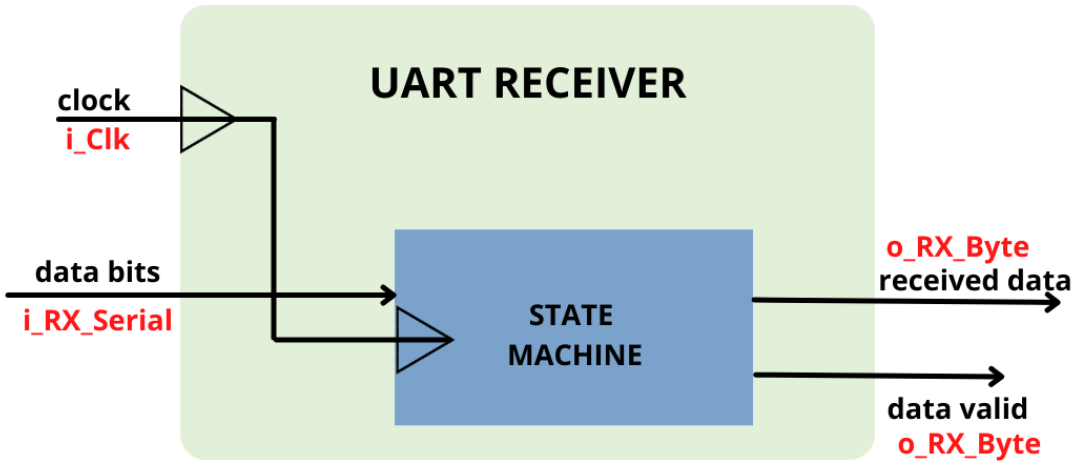

Figure 2: Uart receiver schematic

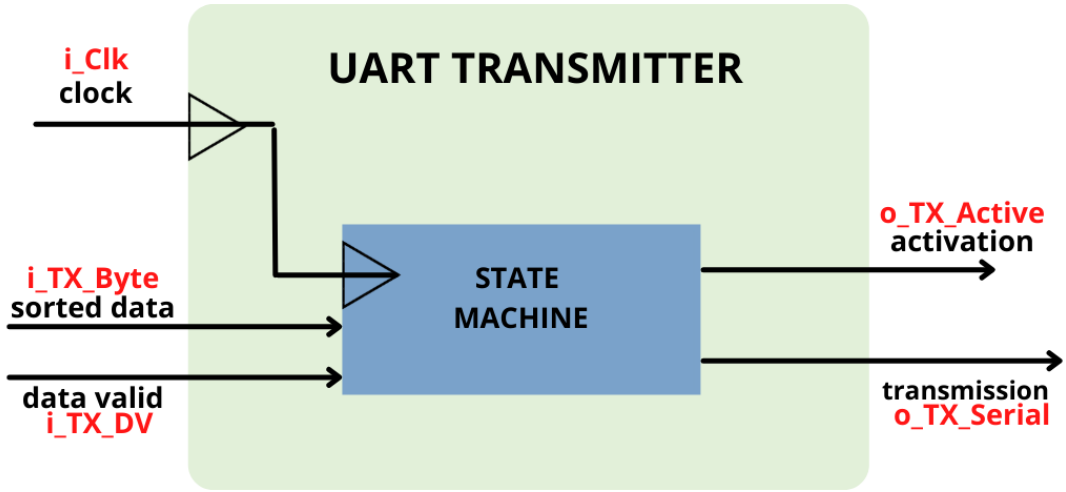

Figure 3: Uart transmitter schematic

2.5 Transmitter

The transmitter is designed to transmit 8 bits of serial data, one start bit as 0, one stop bit as 1, and with no parity bit. It has i_clk, i_Tx_dv and i_Tx_byte as input pins and o_Tx_active o_Tx_serial and o_Tx_done as output pins. The transmitter is activated when a rising edge of i _Tx_dv is detected. The o_Tx_active and the o_Tx_serial are initialized at 0 and 1 respectively. Once a bit is detected the o_Tx_active goes high and clk count starts. The byte index will count from 0 up to 7. The sampling line is incremented after every clk cycle until all the bits are transmitted (i.e bit index of 7). This will assert o_Tx_done high for one clock cycle and o_Tx_active becomes low again. The o_Tx_byte will contain the serial transmitted data in the

form of a standard logic vector.

2.6 Bubble Sort Algorithm

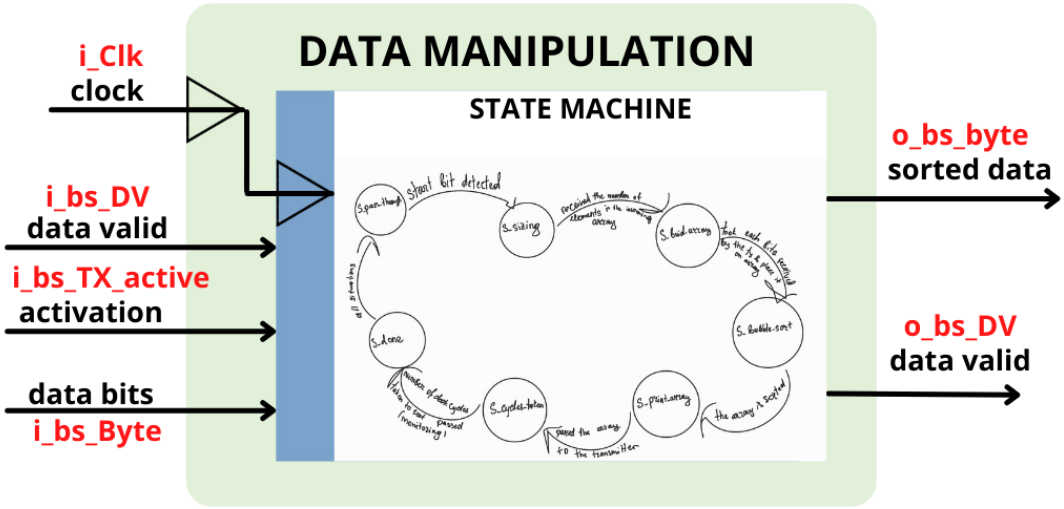

Figure 4: Bubble sort overview schematic

The bubble sort algorithm is used to sort an array of 8 bit unsigned integers. It has i_clk, i_bs_dv and i_bs_byte, i_bs_Tx_active as input pins and o_bs_byte, o_bs_dv, o_bs_LED0, o_bs_LED, o_bs_LED2 and o_bs_LED3 as output pins. The LED's are used to signal which state the board is in for debugging purposes. Our module consists of two operations. The operation that compares the two adjacent numbers in the array and the operation that does the swapping in memory when the condition is true. It loops through the array to push the largest number in memory to the highest index in the array. If the number at index [i] in memory is greater than the number at [i+1] memory the two numbers are swapped. The sampling line is incremented after every pass until all the array is sorted in ascending order.

2.6.1 The State Machine for the Bubble Sort

For the bubble sort we have the state machine approach just like we had for the transmitter and the receiver. We define a new type t_SM_Main here as well. In this case we have 7 states:

1. s_pass_through - this acts like the idle state
2. s_sizing - here we receive the number of elements in the array that we are going to get
3. s_build_array - here we create the array byte by byte
4. s_bubble_sort - the sorting of the array is being done here

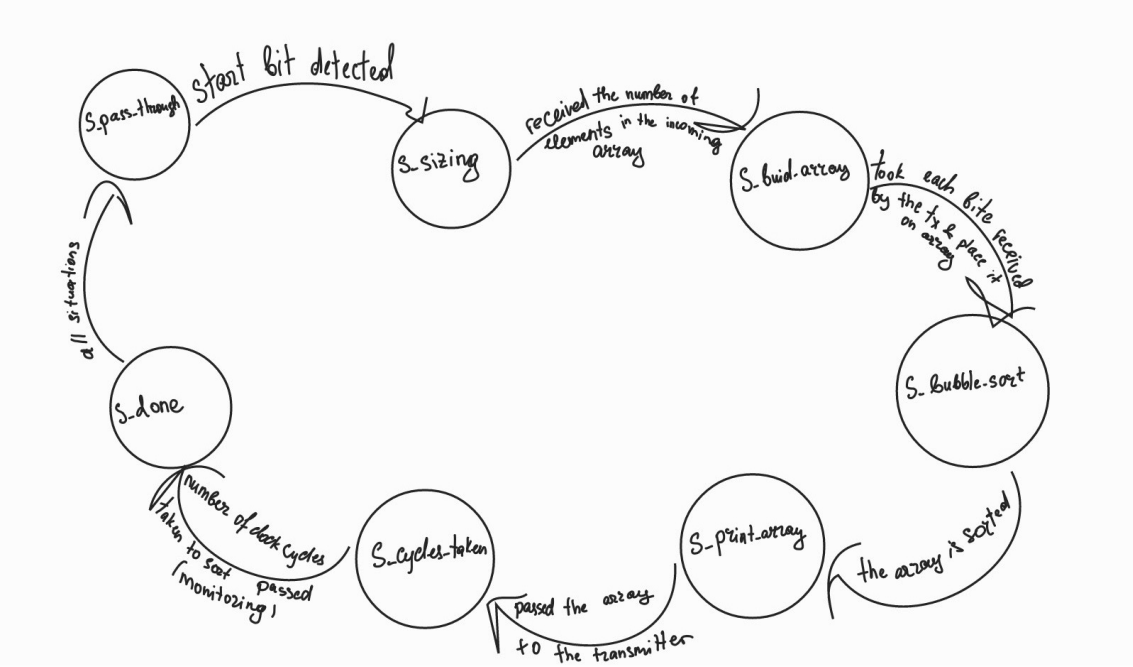

Figure 5: Bubble sort state machine schematic

5. s_print_array - we print the array that we have
6. s_cycles_taken - we use this state for performance monitoring
7. s_done - All LED's are lit to signal that all the states have been traversed

In the picture above we can see how this state machine functions. First, we start in the idle - s_pass_through state. Then we check if the start byte is detected (unsigned integer 105).If it is detected we move to the next state. If it is not, we stay in the s_pass_through state until it actually is detected. Once the start byte is detected, we go to the s_sizing state, where we receive the number of the elements of the incoming array. Then we move to the s_build array. Here we receive the data and build the array byte by byte. We check if all of the data is detected. If it is not, we stay in this state until it is. If it is all detected, we move to the next state, which is s_bubble_sort. Here we implement the bubble sort algorithm. We stay there until all of the elements are sorted. We also count the number of elements sorted and left to sort. We leave this state when there are 0 elements left to sort and the amount of sorted elements is equal to the number of all the elements. We move to the next s_print_array state once everything is sorted. Here we pass the sorted array to the transmitter byte by byte. Then we check to see if the transmission is done and move to the s_cycles_taken state where we pass the number of clock cycles taken to sort to the transmitter. Afterwards we move to the final s_done. As mentioned, this lights up all of the LEDs as a signal we have traversed every state. Then we move back to the s_pass_through and wait for the start byte again to signal the incoming of another array to sort.

2.7 Test Bench

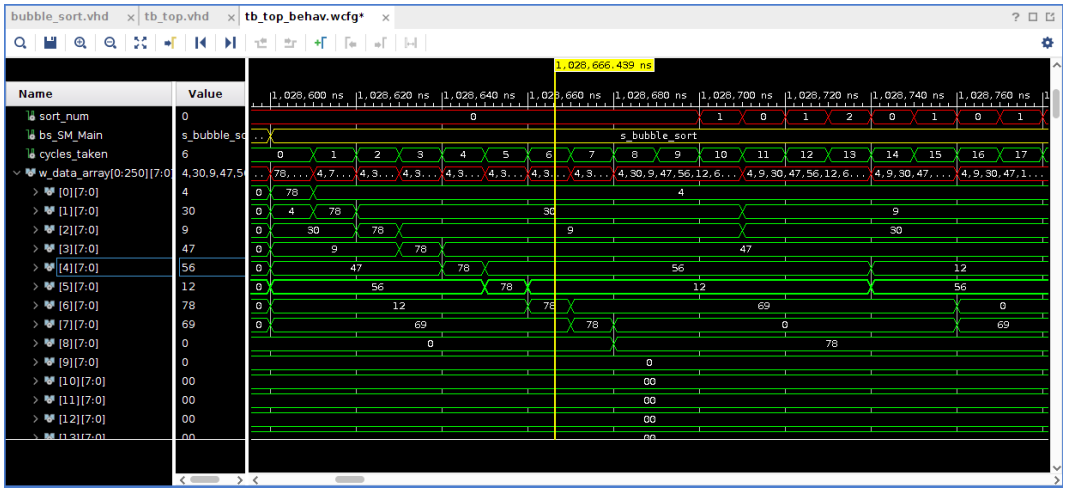

Figure 6: Simulation Example

We made a test bench and made a simulation. So, we made 10 variables with 10 bits each. The whole data to be tested out was composed of those 10 variables. We made a for loop which did 10 iterations and gave the receiver an initial "1" value for one baudrate cycle to simulate the idle state. We made another for loop inside the previous one, so that we can iterate through each bit for each variable. Once we exit the loops we give the receiver another "1" value to bring it back again to the idle state.We then imported many of the internal signals so we could monitor exactly what was taking place during the simulation. This helped significantly for debugging.

2.8 Python for Performance Testing

A python script is used to send unsigned integers to the FPGA board via the serial interface. The pyserial library is used. It will send 150 arrays from a size of 5 elements to 250 elements. The integers range from 0 to 99. The scripts will receive the sorted array and this can be used to check that the FPGA has sorted the array correctly. It will also receive the number of 10ns clock cycles the board took to sort the data. This is then used to calculate the sort time for each array size. The exact same arrays are passed through python versions of bubble sort, optimized bubble sort as well as numpy.sort(). The sort times of each array size is measured for each python algorithm using python's timeit library. These are then plotted and the results can be seen in section 3. The full python code is available in the appendix.

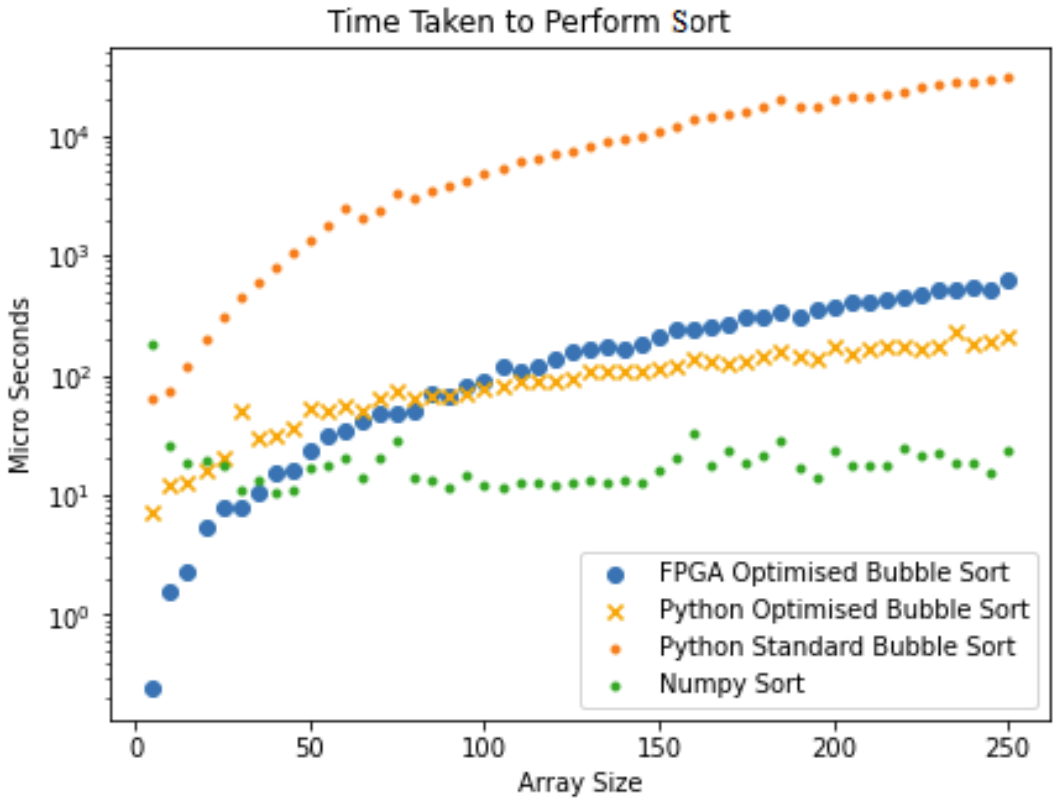

Figure 7: Time to perform sort against array size for the various bubble sorts and reference numpy sorting, with log scaled time axis.

3 Results

The FPGA optimised bubble sort performed favorably on small array sizes and was comparable to it's mid range laptop[1] python implementation on larger array sizes as shown by fig 7. For larger array sizes the numpy sort performed better as a expected. This is due to the fact even a optimised bubble sort is a relatively inefficient algorithm.

Figure 8 shows the general trend of our optimised bubble sort algorithm. It increases with the array size squared. This is a stark contrast against the python implementation that appears to be linear. As these algorithms are supposed to be identical their speed should also scale following the same trend. This graph shows that our version of bubble sort isn't behaving as expected. The most obvious conclusion is that the sort is behaving like the non optimised bubble sort which is

[1]Laptop Hardware: Processor: AMD® A10-5745m apu with radeon(tm) hd graphics × 4 Clock Speed: 2.1Ghz Python Version: 3.6 RAM: 8GB
FPGA Board: Model: artix 7-xc7a100t csg324-1 Clockspeed: 166.667 MHz

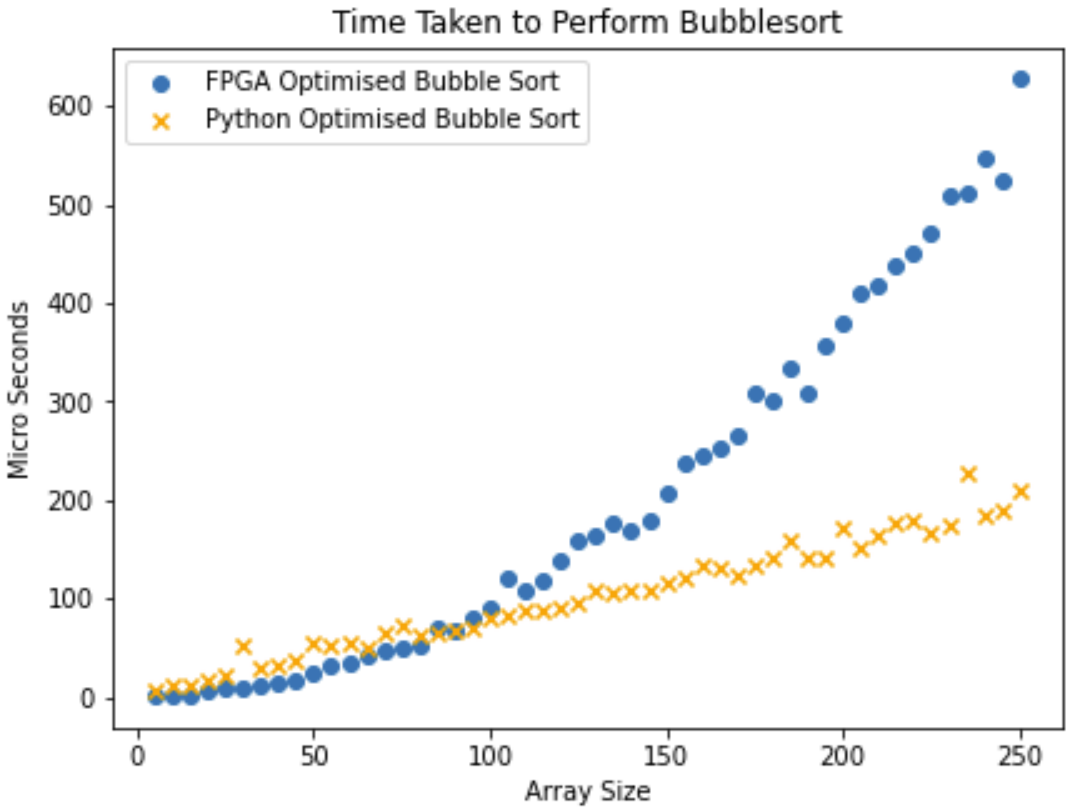

Figure 8: Direct comparison of optimised bubble sort between laptop running Python and Arty A7 FPGA

always quadratic in nature. In other words the sort is not being halted when the array is sorted but rather must iterate through the array the same number of times as the array size every time.

4 Conclusion

The FPGA optimised bubble sort performed remarkably well given its low power usage. It had compairable sort times to the python optimised bubble sort that was ran on a more resource heavy and power demanding machine. This result is in line with the growing trend to implement FPGA on efficiency critical operations such as cryptocurrencies, neural networks and in particle physics.

In addition to this, the project was successful in its primary objective of establishing a means to transmit, receive and process data via 8-bit UART serial connection.

Whilst there was success in the speed of the FPGA optimised bubble sort algorithm. This implementation is severely limited by the speed of the serial connection. To be of practical use it is recommended that faster channels such as Ethernet should be used. On the Arty A7 Ethernet can support connection speeds of 10/100Mbs.

5 Appendix

5.1 Debugging and practical working in VHDL

As this was our first substantial project written in VHDL it was important to optimize our work flow through extensive debugging.

1. Syntax and readability of code
2. Test bench Simulation
3. Pyserial
4. LED debugging on the hardware

1 Syntax and readability of code
We used a naming syntax of in which all input and output signals were prefixed with i for input or o for output. As well as this everything defined has a name that matches its use case.

2 Test Bench Simulation
The test bench simulated the start bit been sent to the board followed by a sequence of 8 non ordered numbers. Whist this test is much smaller that that of the intended use case following the various signals as a function of time turned out to be an invaluable tool for finding the origin of any unexpected behaviour.

3 Pyserial/Pass through state
Our idle state was wrote so that it would transmit any received data back to the transmitter as a pass through. This was useful as it enabled verification that the board was still receiving and sending properly.It also enabled a check on various things such as does the behaviour change with the start key byte has been received

4 LED Debugging
We utilized the LED's on the board to let us know what state the board was in at a given moment in time. Usually the only visible states (due to the speed of the board) are idle and the completion state if the LED completion display is set on. However, the board is stuck somewhere else its a good indication that there is an error the most often of which is not sending the same number of bytes you told it to expect so the FPGA remains in the build array state.

5.2 bubble_sort.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity bubble_sort is
6    generic (
7      MAX_DATA_SIZE : integer := 250; --max number of array elements sortable
8      LED_FIN_WAIT : integer := 500000000      -- 5s
9      );
10
11     port (
12       i_bs_DV     : in std_logic;--signal that i_bs_Byte is ready to read
13       i_Clk       : in  std_logic;
14       i_bs_TX_Active  : in  std_logic;--when 1 the transmitter is sending
15       i_bs_Byte   : in  std_logic_vector(7 downto 0);--data coming from receiver
16       o_bs_Byte   : out  std_logic_vector(7 downto 0);--data sent to transmitter
17       o_bs_DV     :  out std_logic;--signal for transmitter that o_bs_Byte is
         ↪  updated for sending
18       --led indicators for in the case the fpga is stuck in a state
19       o_bs_LED0    : out  std_logic :='0'; --pass through
20       o_bs_LED1    : out  std_logic :='0'; --build array
21       o_bs_LED2    : out  std_logic := '0'; -- bubble sort
22       o_bs_LED3    : out std_logic  :='0'); -- array write
23       -- ALL ON IF GETS THOUGH 5s
24
25 end bubble_sort;
26
27 architecture bs of bubble_sort is
28
29       --Type declarations, state machine and data array type
30       type t_SM_Main is (s_pass_through, s_build_array, s_bubble_sort,
         ↪  s_print_array, s_done, s_sizing, s_cycles_taken); --STATE MACHINE
31       type t_data_array is array (0 to MAX_DATA_SIZE) of  std_logic_vector(7
         ↪  downto 0);
32
33       --Type usage
34       signal bs_SM_Main : t_SM_Main := s_pass_through;
35       signal recived_data_array : t_data_array := (others => (others => '0'));
36       signal w_data_array : t_data_array := (others => (others => '0')); --tried
         ↪  with and without this
37
38       --Indexes
39       signal r_DV_Count : integer range 0 to MAX_DATA_SIZE:= 0;
40       signal bs_index : integer  range 0 to MAX_DATA_SIZE:= 0;
```

```
41        signal sort_num : integer range 0 to MAX_DATA_SIZE:= 0; --just seeing if
          ↪  can loop all the way with no swaps then done
42        signal print_index : integer range 0 to MAX_DATA_SIZE:= 0;
43        signal led_wait_index : integer range 0 to 500000000:= 0;
44        signal print_wait_index : integer range 0 to 868:= 0;
45
46        --Used to measure performance
47        signal cycles_taken : integer range 0 to MAX_DATA_SIZE**2 := 0;
48
49        --Used to wait for a byte to finish sending before updating o_bs_byte
50        --i_bs_Active is only high while sending but not while a byte is waiting to
          ↪  be sent
51        --This leads to some bytes being skipped. byte_sent resolves the issue.
52        signal byte_sent : integer range 0 to 1:= 0;
53        --The number of array elements expected to be received.
54        signal DATA_SIZE : integer := 0;
55  begin
56
57    p_bubble_sort : process (i_Clk)
58
59    begin
60       if rising_edge(i_Clk) then
61
62
63
64       case bs_SM_Main is
65         --pass though will return what is sent to it.
66         --Used as an idle state and a testing state
67         when s_pass_through =>
68          o_bs_LED0 <= '1';
69          if i_bs_DV = '1' then
70            --105 is the key, i.e. the signal from the computer we are about to
              ↪  receive data.
71            if i_bs_Byte = std_logic_vector(to_unsigned(105, 8)) then
72              o_bs_DV <= '0';
73              bs_SM_Main <= s_sizing;
74            else
75              o_bs_Byte <= i_bs_Byte;
76              o_bs_DV <= i_bs_DV;
77              o_bs_LED0 <= '0';
78              bs_SM_Main <= s_pass_through;
79            end if;
80          end if;
81
82         --sizing is used to receive the number of elements expected in the
           ↪  incoming array.
```

```
83          when s_sizing =>
84             if i_bs_DV = '1' then
85               DATA_SIZE <= to_integer(unsigned(i_bs_Byte));
86               bs_SM_Main <= s_build_array;
87             else DATA_SIZE <= DATA_SIZE;
88             end if;
89
90          --This state takes each byte received by the transmitter and places it in
            ↪  an array.
91          when s_build_array =>
92          o_bs_LED1 <= '1';
93            if r_DV_Count < DATA_SIZE then -- we are using the  data valid pulse to
              ↪  know when to move to next byte
94               if i_bs_DV = '1' then  -- got data
95                  r_DV_Count <= r_DV_Count +1; --rising edge cant be used twice but
                    ↪  this does work
96                  recived_data_array(r_DV_Count) <=  i_bs_Byte;
97               end if;
98
99            else
100              r_DV_Count <= 0;
101              w_data_array <= recived_data_array;
102              o_bs_LED1 <= '0';
103              bs_SM_Main <= s_bubble_sort;
104           end if;
105
106         --Implementation of the optimized bubble sort algorithm on the built
            ↪  array
107         when s_bubble_sort =>
108         o_bs_LED2 <= '1';
109           cycles_taken <= cycles_taken + 1;
110           if bs_index < DATA_SIZE - 1 then
111
112              if w_data_array(bs_index) >  w_data_array(bs_index+1) then --swap
113                 w_data_array(bs_index) <= w_data_array(bs_index+1);
114                 w_data_array(bs_index+1) <= w_data_array(bs_index);
115                 sort_num <= 0;
116              else
117                 sort_num <= sort_num+1;
118              end if;
119              bs_index <= bs_index+1;
120           else -- now at full data size
121
122               if sort_num = DATA_SIZE - 1 then  -- can only happen if no swaps
                  ↪  occured as due to reset of sort_num
123                  sort_num <= 0; --reset for next time
```

```
124                 bs_index <= 0;
125 
126                 bs_SM_Main <= s_print_array;
127             else
128                 sort_num <= 0;
129                 bs_index <= 0;
130                 o_bs_LED2 <= '0';
131                 bs_SM_Main <= s_bubble_sort;
132             end if;
133           end if;
134 
135         --print array will pass each byte of the sorted array to the transmitter
136         when s_print_array => --tested and working
137         o_bs_LED3 <= '1';
138               if print_index < DATA_SIZE then
139 
140                   o_bs_DV <= '0';
141 
142                   if i_bs_TX_Active = '0' then -- dont send when transmitter is
                      ↪  sending
143                      -- byte_sent stops index increasing in the lag before the
                         ↪  transmitter starts sending
144                      if byte_sent = 0 then
145                          o_bs_DV <= '1';
146                          o_bs_Byte <=  w_data_array(print_index);
147                          print_index <= print_index +1;
148                      end if;
149                      byte_sent <= 1;
150                   else
151                     byte_sent <= 0; --can only be switched off when transmitter
                        ↪  has started
152                   end if;
153 
154               else
155                   o_bs_DV <= '0';
156                   print_index <= 0;
157                   byte_sent <= 0;
158                   o_bs_LED3 <= '0';
159                   bs_SM_Main <= s_cycles_taken;
160               end if;
161 
162         --cycles taken is used for performance monitoring.
163         when s_cycles_taken =>
164         --This state will repeatedly send the number to the number of clock cycles
            ↪  the bubble sort took to complete.
```

```
165         --This method was the simplest way of overcoming the 255 number size
            ↪ restriction of an 8 bit integer.
166            if cycles_taken > 0 then
167               if i_bs_TX_Active = '0' then -- dont send when trasmitter is busy
168                     if byte_sent = 0 then --stops index increasing in the lag
                        ↪ before the transmitter starts sending
169                           o_bs_DV <= '1';
170                           o_bs_Byte <= "00000010";
171                           cycles_taken <= cycles_taken - 1;
172                        end if;
173                     byte_sent <= 1;
174                  else
175                    byte_sent <= 0; --can only be switched off when transmitter
                       ↪ has started
176                  end if;
177               else
178                     o_bs_DV <= '0';
179                     byte_sent <= 0;
180                     bs_SM_Main <= s_done;
181               end if;
182
183         --done is a final light show to confirm the states have been fully
            ↪ traversed.
184         when s_done =>
185            if led_wait_index < LED_FIN_WAIT then
186              led_wait_index <= led_wait_index + 1;
187              o_bs_LED0 <= '1';
188              o_bs_LED1 <= '1';
189              o_bs_LED2 <= '1';
190              o_bs_LED3 <= '1';
191            else
192              o_bs_LED0 <= '0';
193              o_bs_LED1 <= '0';
194              o_bs_LED2 <= '0';
195              o_bs_LED3 <= '0';
196              led_wait_index <= 0;
197              bs_SM_Main <= s_pass_through;
198            end if;
199
200         when others =>
201            bs_SM_Main <= s_pass_through;
202
203        end case;
204
205      end if;
206    end process p_bubble_sort;
```

```
207  end bs;
208
```

5.3 top.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity top is
5
6     port (
7       CLK100MHZ    : in  std_logic;
8       uart_txd_in  : in  std_logic;
9       uart_rxd_out : out std_logic;
10      led0          : out std_logic;
11      led1       : out std_logic;
12      led2       : out std_logic;
13      led3       : out std_logic);
14
15  end entity top;
16
17  architecture str of top is
18    signal i_Clk        : std_logic;
19    signal i_TX_Byte    : std_logic_vector(7 downto 0);
20    signal i_bs_Byte    : std_logic_vector(7 downto 0);
21    signal i_bs_TX_Active : std_logic;
22    --signal i_TX_Byte    : std_logic_vector(7 downto 0) := X"61";
23    signal i_TX_DV      : std_logic;
24    signal o_TX_Active  : std_logic;
25    signal o_TX_Serial  : std_logic;
26    signal o_TX_Done    : std_logic;
27    signal i_bs_DV      : std_logic;
28    signal o_bs_DV      : std_logic;
29    signal o_bs_LED0    : std_logic;
30    signal o_bs_LED1    : std_logic;
31    signal o_bs_LED2    : std_logic;
32    signal o_bs_LED3    : std_logic;
33    component UART_TX is
34      port (
35        i_Clk        : in  std_logic;
36        i_TX_Byte : in  std_logic_vector(7 downto 0);
37        i_TX_DV   : in  std_logic;
38        o_TX_Active         : out std_logic;
39        o_TX_Serial      : out std_logic;
40        o_TX_Done   : out std_logic);
41    end component UART_TX;
42
43    component bubble_sort is
44      port (
45        i_bs_DV     : in std_logic;
46        i_Clk       : in  std_logic;
47        i_bs_TX_Active  : in  std_logic;
48        i_bs_Byte   : in  std_logic_vector(7 downto 0);
49        o_bs_Byte   : out  std_logic_vector(7 downto 0);
50        o_bs_DV     :  out std_logic;
51        o_bs_LED0    : out  std_logic;
```

```
52        o_bs_LED1    : out  std_logic;
53        o_bs_LED2    : out  std_logic;
54        o_bs_LED3    : out std_logic);
55
56    end component bubble_sort;
57
58    component UART_RX is
59      port (
60        i_Clk         : in  std_logic;
61        i_RX_Serial       : in  std_logic;
62        o_RX_DV         : out std_logic;
63        o_RX_Byte : out std_logic_vector(7 downto 0));
64    end component UART_RX;
65
66  begin  -- architecture str
67
68     UART_RX_1 : UART_RX
69      port map (
70        i_Clk         => CLK100MHZ,
71        i_RX_Serial       => uart_txd_in,
72        o_RX_DV         => i_bs_DV,
73        o_RX_Byte => i_bs_Byte); -- this is where wee are setting loop
74
75
76    bubble_sort_1 : bubble_sort
77      port map (
78        i_bs_DV   => i_bs_DV,
79        i_bs_TX_Active => i_bs_TX_Active,
80        i_Clk         => CLK100MHZ,
81        i_bs_Byte         => i_bs_Byte,
82        o_bs_DV  => i_TX_DV,
83        o_bs_Byte       => i_TX_Byte,
84        o_bs_LED0 => led0,
85        o_bs_LED1 => led1,
86        o_bs_LED2 => led2,
87        o_bs_LED3 => led3
88        ); -- this is where wee are setting loop back
89
90    UART_TX_1 : UART_TX
91      port map (
92        i_Clk        => CLK100MHZ,
93        i_TX_Byte => i_TX_Byte,
94        i_TX_DV   => i_TX_DV,
95        o_TX_Done => o_TX_Done,
96        o_TX_Active  => i_bs_TX_Active,
97        o_TX_Serial      => uart_rxd_out
98        );
99
100 end architecture str;
```

5.4 tb_top.vhd

```
1   --RUN SIMULATION FOR 1200us
2   library IEEE;
3   use IEEE.STD_LOGIC_1164.ALL;
4   use ieee.numeric_std.all;
```

```
5
6   entity tb_top is
7   --  Port ( );
8   end tb_top;
9
10  architecture Behavioral of tb_top is
11  type a_std_v is array (0 to 9) of std_logic_vector(9 downto 0);
12
13  component top
14  port (
15      CLK100MHZ    : in  std_logic;
16      uart_txd_in  : in  std_logic;
17      uart_rxd_out : out std_logic
18      );
19  end component top;
20
21  signal CLK100MHZ : std_logic;
22  signal uart_txd_in :  std_logic;
23  signal uart_rxd_out :  std_logic;
24  signal value_bin : std_logic_vector(7 downto 0);
25  signal uart_txd_in_vec : std_logic_vector(7 downto 0);
26
27  begin
28  DUT : top port map(CLK100MHZ => CLK100MHZ, uart_txd_in => uart_txd_in, uart_rxd_out =>
    ↪  uart_rxd_out);
29
30  main : process --set data size in the top in 8 when runing the sim
31      variable a : std_logic_vector(9 downto 0) := '1' & std_logic_vector(to_unsigned(105, 8)) & '0';
32      variable b : std_logic_vector(9 downto 0) := '1' & std_logic_vector(to_unsigned(2, 8)) & '0';
33      variable c : std_logic_vector(9 downto 0) := '1' & std_logic_vector(to_unsigned(78, 8)) & '0';
34      variable d : std_logic_vector(9 downto 0) := '1' & std_logic_vector(to_unsigned(4, 8)) & '0';
35      variable e : std_logic_vector(9 downto 0) := '1' & std_logic_vector(to_unsigned(30, 8)) & '0';
36      variable f : std_logic_vector(9 downto 0) := '1' & std_logic_vector(to_unsigned(9, 8)) & '0';
37      variable g : std_logic_vector(9 downto 0) := '1' & std_logic_vector(to_unsigned(47, 8)) & '0';
38      variable h : std_logic_vector(9 downto 0) := '1' & std_logic_vector(to_unsigned(56, 8)) & '0';
39      variable i : std_logic_vector(9 downto 0) := '1' & std_logic_vector(to_unsigned(12, 8)) & '0';
40      variable j : std_logic_vector(9 downto 0) := '1' & std_logic_vector(to_unsigned(69, 8)) & '0';
41      variable data : a_std_v := (a,b,c,d,e,f,g,h,i,j);
42
43  begin--begin main
44
45  value_bin <= std_logic_vector(to_unsigned(105, 8));
46  for i in 0 to 9 loop
47      uart_txd_in <= '1'; wait for 8680 ns; --idle
48      for j in 0 to 9 loop
49          uart_txd_in <= data(i)(j);
50          uart_txd_in_vec <= data(i)(1) & data(i)(2) & data(i)(3) & data(i)(4) & data(i)(5) &
            ↪  data(i)(6) & data(i)(7) & data(i)(8);
51          wait for 8680 ns;
52          report(std_logic'Image(uart_txd_in));
53      end loop;
54      uart_txd_in <= '1'; wait for 8680 ns; --idle
55  end loop;
56  wait;
57  end process main;
58
59  clk : process
60      begin
```

```
    CLK100MHZ <= '1'; wait for 5 ns;
    CLK100MHZ <= '0'; wait for 5 ns;
end process clk;

end Behavioral;
```

5.5 Python Test

```
import numpy as np
from timeit import timeit

def nps(array):
    return np.sort(array)

def bubble_sort(array):
    for b in range(0,len(array)):
        for i in range(0,len(array)-1):
            if array[i]>array[i+1]:
                bs_mem = array[i+1]
                array[i+1] = array[i]
                array[i] = bs_mem

def bubble_sort_opt(array):
    sort_num = 0
    for b in range(0,len(array)):
        for i in range(0,len(array)-1):
            if array[i]>array[i+1]:
                bs_mem = array[i+1]
                array[i+1] = array[i]
                array[i] = bs_mem
                sort_num = 0
            else:
                sort_num += 1

            if sort_num == len(array)-1:
                return array

bs_times = []
bs_opt_times = []
np_sort_times = []
runs = 10
#generate and sort random arrays of different sizes and store the number of
#clock cycles taken to complete the bubble sort.
for i in range(noOfArrays):
    rd_array  = arrays[i]
    bs_t = timeit('bubble_sort(rd_array)',"from __main__ import bubble_sort, rd_array", number =
    ↪  runs)
    bs_o_t = timeit('bubble_sort_opt(rd_array)',"from __main__ import bubble_sort_opt, rd_array",
    ↪  number = runs)
    np_s_t = timeit('nps(rd_array)',"from __main__ import nps, rd_array", number = runs)
    bs_times.append(bs_t/runs)
    bs_opt_times.append(bs_o_t/runs)
    np_sort_times.append(np_s_t/runs)

```

5.6 Python Board Test

```python
#change for recieving the number or clock cycles to sort.
import serial
import numpy as np
import time
from timeit import timeit
from datetime import datetime as dt
np.random.seed(456345)

ser = serial.Serial('/dev/ttyUSB1',baudrate = 115200, bytesize=8, timeout = 10)

def fpgaSort(array):
    cyc = 1#number of clock cycles taken to complete the sort
    key = 105#the indication the fpga should move out of its idle state
    size = len(array)#
    #write the key to move the fpga out of the idle state
    ser.write((key).to_bytes(1, byteorder = 'big'))
    #send the size of the array we want to sort, requrired for sorting.
    ser.write((size).to_bytes(1, byteorder = 'big'))

    for j in range(size):
        #send the elements one by one via serial communication
        ser.write(int((array[j])).to_bytes(1, byteorder = 'big'))
        #immediatly read the sorted array

        if j==size-1:
            d = ser.read(size)
            '''The fpga will repeatedly send the number two
            The number of 2's recieves is the number of clockcycles taken
            to complete the bubble sort'''
            two = ser.read()
            r = two
            while r == two:
                cyc += 1
                r = ser.read()
        sort = []
        for a in d:
            sort.append(a)
    return cyc sort

maxArraySize = 250
step = 5
noOfArrays = int(maxArraySize / step)
array_sizes = np.arange(step,maxArraySize+1,step)
cycTaken = []
arrays  = []
sortedArrays
#generate and sort random arrays of different sizes and store the number of
#clock cycles taken to complete the bubble sort.

for i in range(noOfArrays):
    rd_array  = np.random.randint(low = 0, high = 99, size = array_sizes[i])
    arrays.append(rd_array)
    cycles, sortedData = fpgaSort(rd_array)
    cycTaken.append(cycles)
    sortedArray.append(sortedData)
ser.close()
```

```
print(cycTaken)
```

5.7 Python Graph Plotting

```
import matplotlib.pyplot as plt

figal, (al, nopsd) = plt.subplots(nrows = 1, ncols = 2, figsize = (15,5))

timeTaken = [x/100 for x in cycTaken]#change number of 10ns cycles to microseconds
#timeTaken_s = [x*10*10**-9 for x in cycTaken]#change from 10ns cycles t

#change from seconds to microseconds
bs_times_us = [x*10**6 for x in bs_times]
bs_opt_times_us = [x*10**6 for x in bs_opt_times]
np_sort_times_us = [x*10**6 for x in np_sort_times]
al.scatter(array_sizes,timeTaken, label = 'FPGA Optimised Bubble Sort')
al.scatter(array_sizes,bs_opt_times_us, label = 'Python Optimised Bubble Sort', marker = 'x', c =
↪  'orange')
al.scatter(array_sizes,bs_times_us, label = 'Python Standard Bubble Sort', marker = '.')
al.scatter(array_sizes,np_sort_times_us, label = 'Numpy Sort', marker = '.')
al.legend()
al.set_xlabel('Array Size')
al.set_yscale('log')
al.set_ylabel('Micro Seconds')
al.set_title('Sort Time Comparison')

nopsd.scatter(array_sizes,timeTaken, label = 'FPGA Optimised Bubble Sort')
nopsd.scatter(array_sizes,bs_opt_times_us, label = 'Python Optimised Bubble Sort', marker = 'x', c =
↪  'orange')
nopsd.legend()
nopsd.set_xlabel('Array Size')
nopsd.set_ylabel('Micro Seconds')
nopsd.set_title('Trend of FPGA Sort Time')
```