**VHDL special project report**

**Important files.**

Ram.vhd: This is the standard RAM File, with parameters modified to take into account the current use case: RAM\_WIDTH := 8, RAM\_DEPTH := 16, RAM\_ADD := 4. This allows for indexing 1 byte words starting address ranging 0 to 15. An important modification is done by adding two external output ports for testing purposes: FIRST and LAST, both std\_logic\_vector of 7 downto 0 handled by a dataflow assignment that links them to the first word in the ram and the last (size-1) word of the ram. Used in the testbenches.

Memory.mem: The file used as initialized for the Ram. This is a standard memory file issued with the pattern given by the assignment pdf

FSM.vhd: Main controller unit. This is a 2 process HLSM handling the logic for the sorting algorithm implemented, asynchronous Reset.

TopLevel2.vhd: this is the toplevel (TopLevelDownUp) entity used by the second testbench for evaluating normal behaviour. A top level wrapper for both the RAM and the FSM controller

TopLevelDownUp\_tb.vhd : this is one of two testbenches, that checks the general behaviour of the TopLevelDownUp entity

tb\_sorting\_system.vhd: this testbenches uses asserts to check if ram contains sorted values.

IMPORTANT: my simulations requires 5,470.000 ns to effectively sort the list. The button “Run for 1000 us” is pressed once to allow the design to complete.

Configuration\_final.wcfg contains all most important waveform to check. Apart from the Clk,Reset, Start signals it contains also the current\_state logic, the inner\_counter, the output data\_out signal and other important waveforms. Used in tb\_sorting\_system.vhd

State\_diagram.png: in case this one is not correctly displayed.

***Extra (Not relative to the assignment. This is a mistake I made but I wanted to underline the learning process)***

Controller.vhd: On my first read of the document, I thought the assignment required to implement a full bubblesort implementation using the “standard” two cycle approach. Since in vhdl dynamic loops are not able to be synthesized, I approached to problem similarly to the “real” assignment, using two signal indexes and implement a FSM approach with multiple states. The only difference was that the inner loop index was still spanning already processed couple of items but since they were already sorted, no swap was necessary. I’m including the controller that implements a full bubblesort algorithm control unit just in case, even though it wasn’t required for the assignment.

Toplevel.vhd: the Top Level entity including both the control unit and the ram

TopLevel\_tb.vhd: Testbench for the Toplevel entity.

**FSM** **Description**

This FSM implements a bubble-sort-like procedure over a memory array. Apart from the standard HLSM architecture, this design relies on an inner\_counter signal that counts from o to 14 to allow for checking the couples in pair, by checking the current N element of the ram and the next N+1 element of the ram if direction is = ‘1’. When reaching 14, direction is inverted, the last iteration is completed and we start from inner\_cycle 15 up until 1, to allow for all the items to be sorted.

The sequential logic handles the asynchronous reset and the normal-operation cycle, updating the state, direction, inner\_counter, two\_counter and the registers, both reg1 and reg2.

The two\_counter signal is used to count 2 CC to allow the RAM to correctly output the value in the DOUT port, and proceeding only when done so. Reg1 and reg2 hold the temporary value of the processed items in the ram. Reg1 holds N while reg2 holds N+1 if dir = ‘1’ and vice versa when dir = ‘0’.

The logic starts in the idle state until start signal is asserted high. The inner\_counter, two\_counter are initialized to zeros. On load1, we perform a read operation on the Ram at the first address and wait until the signal has been correctly assigned to reg1.

Moving to Load2, we perform another load but this time the operation depends on the current iteration of the algorithm, if I’m going upwards the element load in reg2 is the next one from the current, otherwise the previous one.

The compare stage checks for contiguous non-ordered elements in the list. If one of the two conditions is true(Current element is higher than next when going upwards or current element is lower previous when going backwards) we then move to the swap stage, otherwise the pair is ordered and no additional operations are required.

If the swap flag is set to 1, the next swap1 stage performs a write in the next position of the current one when going upwards or in the previous going downwards. This way I’m still keeping the temporary value in reg2 so that it isn’t lost.

Switching to the swap2 stage, we perform the remaining operation of writing to the current address the switched value from before.

Increment stage checks:

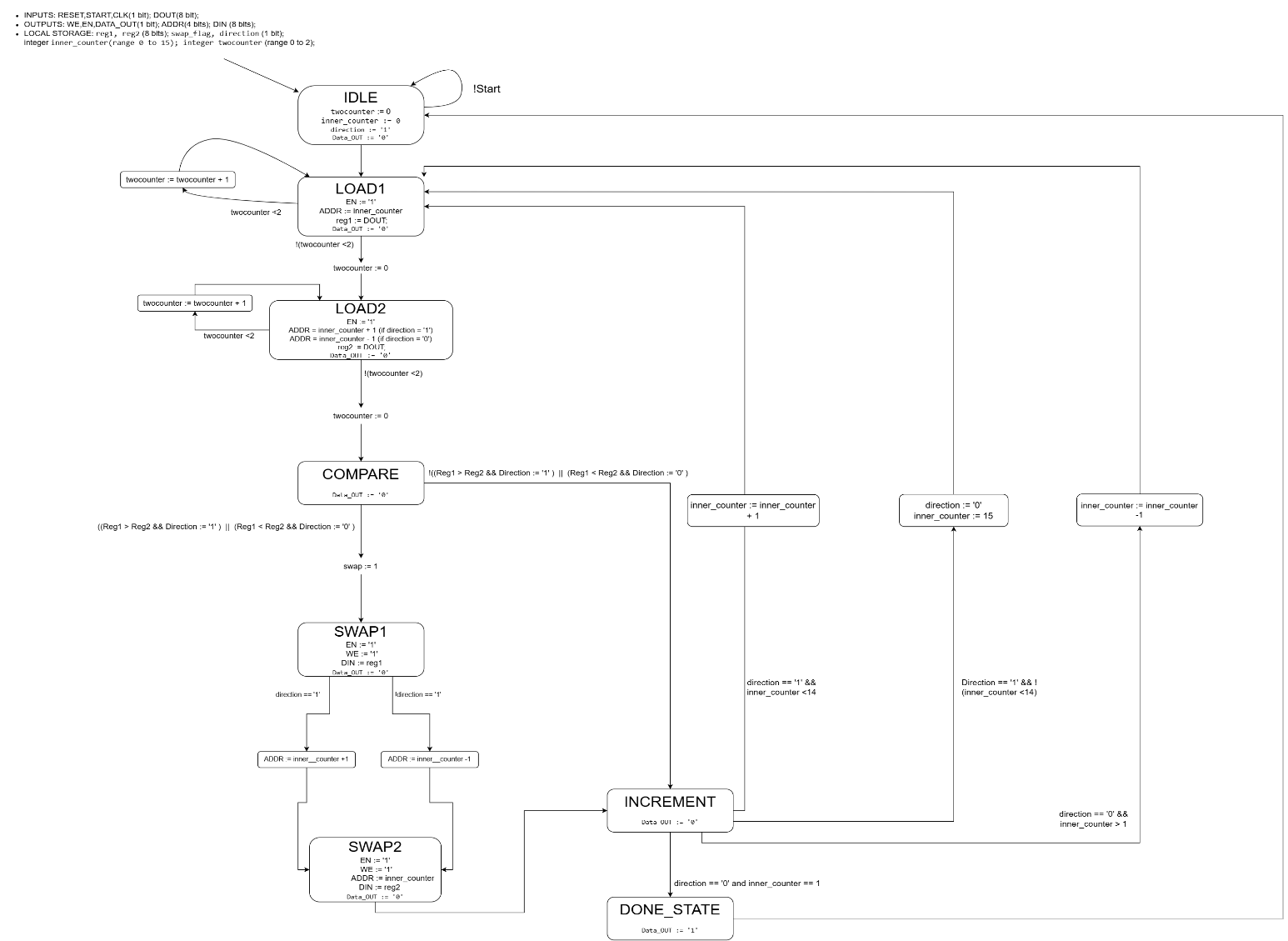
* If direction is ‘1’ (upwards) and the inner\_counter is less than 14, a sign I’m still in the first iteration of the algorithm
* If direction is ‘0’ (downwards) and the inner\_counter is higher than 1, meaning I’m in the second iteration of the algorithm moving downwards.

When sorting in the forward direction (direction = '1'), it increments the pointer (inner\_counter) until it reaches 14, this is because comparisons are done in pairs, so address 14 compares with 15. Once inner\_counter reaches 14, it switches direction to backward (setting next\_direction to '0') and sets the pointer to 15 to start a reverse pass. In the backward direction, the pointer is decremented until it reaches 1; when inner\_counter is no longer greater than 1, the FSM concludes that the sorting is complete and transitions to DONE\_STATE.

In DONE\_STATE, we assert DATA\_OUT to one to signal the end of the logic.

Looking it more accurately, my design could have been better implemented by splitting the load stages into an address decoding stage and an effective load stage, that way removing the need of the two clock cycle signal to wait for the correct value to be loaded, that way improving the overall quality of the design and align better to what the professor explained.

This state diagram could have greatly benefitted from such a choice, while this way the signals assignment for both the addresses are assigned in the LOAD2 stage. There should have been an additional stage for determining the correct address,



**Testbench cases**

TopLevelDownUp\_tb.vhd : This testebench checks for the general behaviour of the TopLevelDownUp entity.

tb\_sorting\_system.vhd: this testbenche uses two final asserts to check that after Data\_out == ‘1’ the RAM is effectively sorted: the two debug ports of the RAM, FIRST and LAST, are checked for the smallest value and largest value of the input pattern.

The testbench waits for the DATA\_OUT signal to indicate that the sorting operation is complete.

A process called clk\_process generates the clock signal, toggling between '0' and '1' every half period of the specified CLK\_PERIOD (20 ns). The FSM entity is instantiated and connected to the testbench signals. The test\_process begins by applying a reset pulse (RESET <= '1' for 40 ns) to initialize the system. After reset, the START signal is triggered to begin the sorting process.

Data\_out is asserted to one when the sorting is complete, I expect to find at ADDR: 0000 the value 0xaa and at ADDR: 1111 the value 0xff. The assert returns an error otherwise.

Access to the RAM allows for checking that the ram gets updated during the sorting.

End of file.