Alan Coon
Carolinne M. Mesquita
EE 354L
26 April, 2017

EE 354L Final Project Documentation

Visual Bubble Sort Algorithm

Background
For our final project, we created a bubble sorting state machine. The machine processes an inputted array of numbers, and performs the sorting algorithm while displaying the progress in real time – displaying the numbers as columns on a monitor. The larger columns represent bigger numbers.

Implementation of State Machine
The implementation accepts a 32-cell wide array of 8-bit numbers as input, and performs the sorting algorithm while displaying the progress in real time via VGA connection to a monitor. Four push buttons are in use. `BtnC` drives the `reset` signal, `BtnR` drives the `start` signal, `BtnD` drives the `ack` signal, and `BtnU` drives the `step` signal. The states are `INITIAL`, `SORT`, `SWAP`, and `DONE`.

The state machine begins in the `INITIAL` state, where the two counters, `count1` and `count2` are set to `0` and `31` respectively. We also pipe the input values into the array `n`, for easier access during the sorting process. We generated random values using a Python script. When the `start` signal goes high, we move from the `INITIAL` state to the `SORT` state. Bubble sort works by "bubbling" up the largest values. While we iterate over the array, we compare each number with its neighbor and swap them if the left number in the pair is larger than the right number of the pair. This iteration is achieved with the `count1` variable which we increment during the `SORT` and `SWAP` states. When the largest unsorted number reaches the top of the array, we set `count1` back to `0` and restart the process to move the second largest unsorted number to the top, right behind the first. This algorithm is repeated for the rest of the values of `n`. The nature of the bubble sort algorithm ensures that after *i* loops, the top *i* values of the array will be sorted correctly. Thus we do not have to consider them in future comparisons, and we can save clocks. This is why we use the `count2` variable. We start `count2` as the maximum index of the array and decrement it to represent the end of the values we are considering as the *i + 1*th largest number.

Within the `SORT` state, we look at the value of n[count1]. If `n[count1]` is greater than the value of `n[count1 + 1]`, then we progress into the `SWAP` state. If `n[count1]` is not greater than the value of `n[count1 + 1]`, we first check if the entire array is sorted and `count2` has reached the bottom of the array. If this is the case, then we enter the `DONE` state. Otherwise, if the value `count1 + 1` is equal to `count2`, and `count2` is not at the bottom of the n array, then we know that we have reached the section of numbers that has already been sorted and that we have found the *i*th largest number. We reset `count1` to `0` and decrement `count2` and "self-loop" back to the `SORT` state. Otherwise, we increment `count1` and maintain the `SORT` state.

All actions in the SORT state only occur if the step signal is high. This provides us with the interactive sorting experience that we aimed to produce when we planned our project.
The SWAP state uses non-blocking statements to exchange the values of n[count1] and n[count1 + 1]. Also in the SWAP state we check to see if count1 is at the end of the n array, and if it is we reset count1 to 0 and decrement count2. Otherwise we increment count1. This is to prevent us from wasting clock cycles by having to go back to the SORT state, and for simplicity of the program.

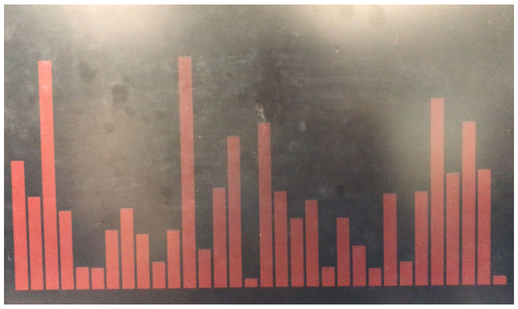
In the DONE state, we wait for the ack signal. Upon receiving the ack signal, we transition to the INITIAL state.

Featured I/O Components
Our project features four push buttons, four LEDs, and a VGA connection to a monitor that displays the visualization of our sorting machine.

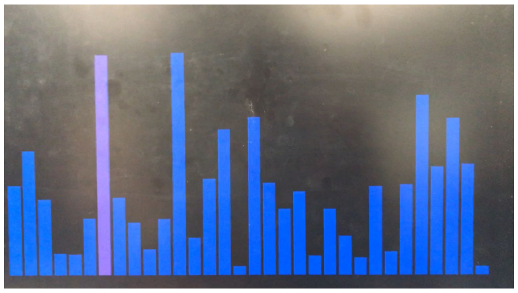
LED State Indicators
Each of the states have a corresponding LED that illuminates when we are in that state. The LEDs are LD0, LD1, LD2, and LD3 for the INITIAL, SORT, SWAP, and DONE state respectively.

Push Buttons for Input and Debouncing
As previously mentioned: BtnC drives the reset signal, BtnR drives the start signal, BtnD drives the ack signal, and BtnU drives the step signal. Each of the buttons are debounced using a debouncing module. The step signal is set up for multiple and continuous enables. This allows the user to hold down the push button and keep the step signal high, quickly advancing the machine and consequently the visualization.

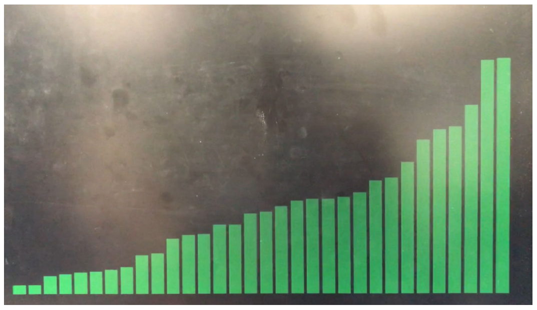
VGA Connection



The visualization of the algorithm is implemented using columns as the representation of the numbers being sorted from the n array. These columns are drawn to a specific location on the monitor for a better visual and interactive experience for our project. We use the *y-axis* as a coordinate starting from the top left point of the VGA mapping, and the down left for the *x-axis* as a starting point.



We created two arrays of numbers, xfrom and xto to help setting the horizontal location of each bar correctly when the n numbers are being sorted. The *y* coordinate defines the size of the columns by its height according to what number is stored the *i*th position of the n array.



All the 32 positions are stored in the `col` array and they are `OR`ed with each other in `columns` to generate the signal for all the bars to be displayed at the same time. This signal is driven to the signal of the corresponding state color of the bubble sort logic (red for `INITIAL`, blue for `SORT` and `SWAP,` and green for the `DONE` state). The `checking` signal is added to the coloring implementation indicating what column on the screen is being analysed by `count1` in the `n` array. The `checking` number bar shows the algorithm process throughout the collection of numbers.