Etch A Sketch

Final Report ENGS 31

Liam Jolley and Graham Hazlett

Abstract

For our project, we created an Etch A Sketch. The device receives input from the two rotary encoders, each of which controls a dimension of the cursor movement. Turning the left knob manipulates the horizontal component, while turning the right knob manipulates the vertical component. Additionally, the first three switches control the red, green, and blue components of the cursor color, respectively, and the final three switches do the same except with the background color. Finally, pressing the left knob clears the canvas of all drawings, and pressing the right knob toggles having the pen up or down.

Table of Contents

1. Introduction

Creating an Etch A Sketch involves taking input from two rotary encoders, debouncing those signals, processing the series of signals, moving the cursor as needed, storing the cursor data, and displaying the data on the monitor. Additionally, the program can process other data from the FPGA to assign cursor and background colors, toggle having the pen up or down on the screen, and clear the canvas of drawings.

2. Design Solutions

2.1. Specifications

Taking inputs from the rotary encoders and six switches, our circuit stores a 2-D array mapping positions to colors, which are encoded using four bits each. The VGA functionality of the circuit simultaneously reads this data onto the monitor continuously, updating the color as the user navigates through the array. Turning the left knob clockwise and counterclockwise moves the pen right and left, respectively, while moving the right knob clockwise and counterclockwise moves the pen up and down, respectively. Pressing the left knob clears the canvas, and pressing the right knob toggles the pen up/pen down functionality. Finally, the first three switches control the red, green, and blue components of the cursor color, respectively, while the final three control those of the background color, respectively. The user can combine these components to create eight colors for either entity. Please see Appendix A for a front panel view of the device as well as Appendix B Figure 1 for the top level block diagram of the system.

2.2. Operating Instructions

To use our Etch A Sketch, the user must turn on the FPGA and connect the device to the monitor with the VGA. To start the device with contrast so that he or she can see a drawing, the user should change the color of either the cursor or the background so that they do not match.

To change the cursor color:
-Flip the first switch up to add a red component to the cursor color (Cursor Red Switch in Appendix A)
-Flip the second switch up to add a green component to the cursor color (Cursor Green Switch in Appendix A)
-Flip the third switch up to add a blue component to the cursor color (Cursor Blue Switch in Appendix A)
-Mix and match these colors to create eight different cursor hues
Note: The trail will stay the color in which it was originally drawn in a particular region

To change the background color:
-Flip the third to last switch up to add a red component to the background color (Background Red Switch in Appendix A)
-Flip the second to last switch up to add a green component to the background color (Background Green Switch in Appendix A)
-Flip the last switch up to add a blue component to the cursor color (Background Blue Switch in Appendix A)
-Mix and match these colors to create eight different background hues

To move the cursor on the screen:
-Rotate the left knob clockwise to move the cursor right
-Rotate the left knob counterclockwise to move the cursor left
-Rotate the right knob clockwise to move the cursor up
-Rotate the right knob counterclockwise to move the cursor down

To clear the screen of any marks:
-Press the left knob once

To toggle the pen up/pen down functionality:
-Press the right knob once

2.3. Theory of Operation

We used a few modules combined with a top level shell to implement our design. The debouncers first process the raw, choppy data from the rotary encoders and pass the cleaned signals onto the signal coders, which articulate the data into rotation and enable signals to model the behavior of both the left and right knobs. The cursor update module then processes these signals and moves the cursor accordingly, storing the current cursor position in anticipation for the next move. In the top level shell, this path is wired together to the Random Access Memory (RAM) block write address input. The RAM block holds a 2-D array of every valid place on the screen mapped to a color–or lack thereof. In addition to this serial data path, other paths are also running in parallel. The VGA module constantly reads the RAM block data onto the monitor, starting at the top left corner of the screen with the first index of the RAM and moving down the array in rows. This module reads from the RAM block at a slightly slower 25 MHz frequency compared to the 100 MHz on which the rest of the system runs. In addition to this concurrent display process, other processes in the top level shell translate the data between modules and wait for color, pen up/pen down, and clear signals from the user. Every time a user changes one of the six color switches, the upper level shell reacts by changing the code written into the RAM block on every cursor movement. When a user toggles the pen up/pen down function, the write enable functionality is also toggled in the RAM block to create the concept of picking up or putting down the pen. Finally, when a user clears the canvas, the RAM block writes zeros into every position, leveraging the VGA cycle to cover every index. On the next cycle when the clear signal is not asserted, and the user can begin drawing again. This process occurs quickly enough that a user does not notice the timing. See the block diagram in Figure 2 of Appendix B for a visual depiction of the data flow. See below for more detailed explanations of each module.

Debouncer:

The debouncer module takes in an input signal, and one 20-bit wide signal, indicating number of clock cycles over which debouncing should be evaluated. These signals input into a state machine controller, which interfaces with a counter and comparator to evaluate debouncing. When the input signal switches from being in a steady state on or off to the opposite signal, a state change is triggered within the state machine. In this state, as long as the signal is in this new position, it increments a counter each clock cycle, until this counter reaches the value of the 20-bit wide input signal, in which case the counter resets, and the output signal is changed to the current input signal. However, if, while the counter is still less than the number of clock cycles desired, the input switches back to what it was before, then the counter resets, and it reverts back to its initial state, without changing the output signal. In this way the debouncer module takes in the input signal and desired number of clock cycles, and effectively outputs a debounced signal on its output terminal. See Figure 1 in Appendix C, and Figure 4 in Appendix B for a visual representation of the debouncer module's operation.

Signal Encoder:

The signal encoder module takes in two signals, A and B from a rotary encoder, and outputs two signals: enable, and direction. Using a state machine, this module senses changes on the A input, and responds accordingly. If the A input changes, it then looks at the value of the B input, and depending on this combination of values, determines whether or not the encoder has turned clockwise or counter-clockwise. For example, if the A signal goes from 0 to 1, and the B signal is currently low, it knows the encoder has turned counter-clockwise. When the module has sensed that the rotary encoder has turned (ie that the value of A has changed), it outputs enable for one clock cycle, and asserts the direction signal as 1 for one clock cycle if it senses the encoder has turned clockwise, or asserts it as 0 if it senses the encoder has turned counterclockwise. See Figure 2 in Appendix C for a visual description of the functionality of this module.

Cursor Logic:

The cursor logic module takes and processes the horizontal and vertical rotation and enable signals coming from the signal encoder. Before interpreting the rotation signals, the process checks each enable signal to understand if either knob is turning. If the a knob turns, the process checks the rotation signal (1 for clockwise, 0 for counterclockwise) and the appropriate out of bounds condition using the modulus function and other comparators. If the cursor does not fall out of bounds on that certain movement, the cursor position is updated as an unsigned variable and is outputted back to the shell. Although the horizontal knob logic comes before the vertical knob logic in the process, both are updated on a clock tick so as to allow for diagonal cursor movement. See Figure 3 in Appendix B for a visual depiction of this process.

VGA:

The VGA module takes in a slowed clock and cycles through the specified boundaries, skipping over the vertical borders. We covered this module design in class, so we will not specify the algorithmic details. The module outputs a vsync signal, which indicates when the process reaches the bottom of the scan; a hsync signal, which indicates when the process reaches the rightmost boundary of the scan; a video_on signal, which is not very important to our functionality; and a graphical pixel location using pixel_x and pixel_y signals. The shell uses these signals to cycle through the monitor pixels and update the screen accordingly. See Figure 5 in Appendix B for a visual representation of the VGA dataflow.

Top Level Shell:

The most fundamental functionality of the top level shell is passing data between other modules. Responsible for the data flow from the rotary encoders to the monitor, the shell links the 4 FPGA signals coming from the Rotary Encoders (See Appendix A) to the debouncers. Next, the shell connects these outputted, debounced signals to the signal encoder. Finally, the shell passes this data into the last processing module of this sequence: the cursor update module. The shell links the cursor update output–the new cursor location–to the write address of the RAM block. Having explained the rotary encoder to RAM datapath, we'll now go through the concurrent processes running on the shell. The VGA module runs on a slower frequency than the rest of the program and is constantly cycling through the monitor's pixel indices to read from the RAM onto that monitor. While the RAM indexes memory addresses from 0 to (HEIGHT * WIDTH) –

1, the VGA deals with a graphical representation of the memory–indexing x and y positions instead. To make this conversion, we used simple arithmetic functions with the pixel_x and pixel_y vectors to generate the new RAM address signal from which to read. We decided to implement a “fat pixel” display with a factor of four, meaning that each index in the RAM represents a four by four block of pixels on the monitor. To do this, we ignored the two least significant bits of both the pixel_x and pixel_y vectors while operating on them. The final vital process for the data flow is that which writes to the pixel_data output. Responsible for manipulating all of the previously discussed data into an visual user interface, this process assigns the concatenation of the FPGA signals from the Background Switches (See Appendix A) to pixel_data if that certain RAM data has not been visited (contains a zero as the least significant bit of the 4-bit RAM output). Otherwise, the process assigns the concatenation of the raw_color signal, which comes from the first three most significant bits of the RAM block memory. We made translations from the 3-bit stored colors to 12-bit outputted colors by assigning the first four most significant bits of the output color to the red bit value, which is the most significant; the next four to the green bit, which is the second-most significant; and the final four to the blue bit–the third-most significant. These processes suffice to implement the basic functionality of the Etch a Sketch.

The remaining processes are triggered by other FPGA signals changing. The simplest is the pen up/pen down functionality, which is processed simply by monopulsing and toggling an input signal, which changes the write enable signal of the RAM block to the opposite of its current state. This is the only process that changes this write enable signal, so there is no conflict in the program. We implemented the clear functionality with a process that diverts the VGA cycling addresses into the RAM memory write address to make sure every index is cleared. Simultaneously, the color_in signal is reset to zeros for all of these blocks. This completely clears the memory so that the monitor will be completely reset when clear is deasserted. To implement the changing cursor color, a process concatenates the FPGA signals from the Cursor Switches (See Appendix A) into a four-bit color_in signal with the final bit representing whether or not the index represents a cursor color (1) or a background color (0). This final, least significant bit contributes to the monitor update process in the top level shell, which controls the VGA color output as we discussed earlier. Lastly, there is a process that uses flip flops convert the 100 MHz system clock to a 25 MHz clock for the VGA clock to run. See Figure 2 in Appendix B for a detailed visual representation of the data flow.

2.4. Construction and Debugging

We began by implementing and testing the cursor update and signal encoder modules, thinking that we would modularize first and then combine. Flying through these modules, we then moved on to implementing the VGA code with the provided test pattern. Thinking that we would use the code directly from class, we tested for a while before realizing that the in-class solution was incomplete and deciding to rewrite the fundamentals of the code. Because we covered the main ideas of this module in class, we did not extensively document the VGA module. Having written these three vital modules for the data processing path, we began writing the shell to tie all of these parts together. Initially, we implemented the Etch A Sketch on a small screen to make the fat pixel data manipulation easier. We ran into debugging problems when we incorrectly tried to store the cursor position in the top level shell as well as in the cursor update module. This contradiction created a checkerboard pattern on the monitor as well as a weird cursor position fluctuation. We were finally able to catch the issue when we created a testbench for the top level shell and saw this inconsistency in the waveform. After achieving this basic functionality, we added on functionalities in this order: making the program take up the entire screen, creating cursor colors, creating background colors, making a clear functionality, and allowing for the user to toggle pen up/pen down. We did not run into any more significant errors or debugging. See Appendix H for all waveforms used to test and debug each module.

3. Evaluation

Having pretty seamlessly modularized and integrated our code, we are proud of our design and implementation process. Making modules for the cursor update, signal encoder, and debouncing functionalities made our combination task much easier because we had already tested these modules. After achieving the basic functionality, we started adding on auxiliary processes for the add-on functionalities, which we ideally would've modularized if we had more time. Additionally, we were satisfied by our data flow through the top level shell. Having three switches represent each color of the RGB scale for both the cursor and the background made concatenation and color updating a lot easier. Also, our idea to have the final bit in each RAM index address represent the "visited" status of that certain location allowed for us to have an easy way to know if the process even needed to deal with the other three bits. We also believe that the user controls have good learnability and are easy to manipulate.

4. Conclusions

Having finished the project with complete functionality, we are ecstatic with our final result and satisfied with our construction and debugging processes. In the future, we would aim for a little more modularization with the add-on functionalities, but we're glad we kept the top level shell pretty sparse of extraneous processes. We overhauled our RAM block to extend the data to the entire monitor, but that transition went down without a fault. If we were to do this project again, we would look to understand all of the concepts equally well before jumping into the implementation because, while we flew through the cursor update and signal encoder modules, the debouncer and VGA modules did not go as quickly due to our lack of comprehensive planning. Overall, we're super proud of our progress.

5. Acknowledgements

Thank you to Professor Hansen for his guidance and teaching throughout the term leading up to the project. We really felt rewarded building from simple logic gates in week one to a full working Etch A Sketch this past week. Thank you also to Dave, Nick, and Matt for their support during all of the labs. They knew when to let us work out problems on our own and when to lend a helping hand.

We started the project by divvying up the work into modules. Liam designed, implemented, and tested the cursor update and the VGA modules, while Graham dealt with the signal encoder and debouncing modules. Both of us came together to collaborate on the top level shell with Liam

doing the majority of the initial wiring and debugging and Graham creating extension processes to improve on our baseline functionality. We debugged our tough cursor logic bug together with a top level testbench, which was probably our highlight of the project. Although this was the darkest time for us, we also came together as a duo for the first time in the project and finally fixed the bug. While Liam did the bulk of the report writing, each of us did our own code documentation and commenting.

6. References

We took all of our information from past lectures and labs as noted in file headers. Our most significant reference was taking and altering Professor Hansen's VGA controller code. Otherwise, the top level shell clock processes were from labs 5 and 6.