Gabriel Dimas

6 June 2022

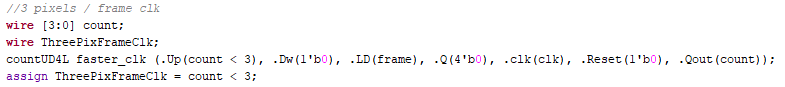
Section C

**Description**

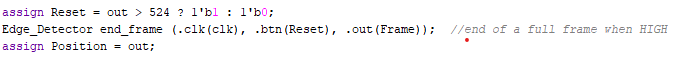
The purpose of this lab was to create a game using VGA wiring, timing, state machines, and point accuracy when displaying data to a screen. Although one of the hardest labs, it did in fact do its duty in teaching how data is displayed to a user and the arithmetic and inequalities that go into game design and hardware management.

**Design**

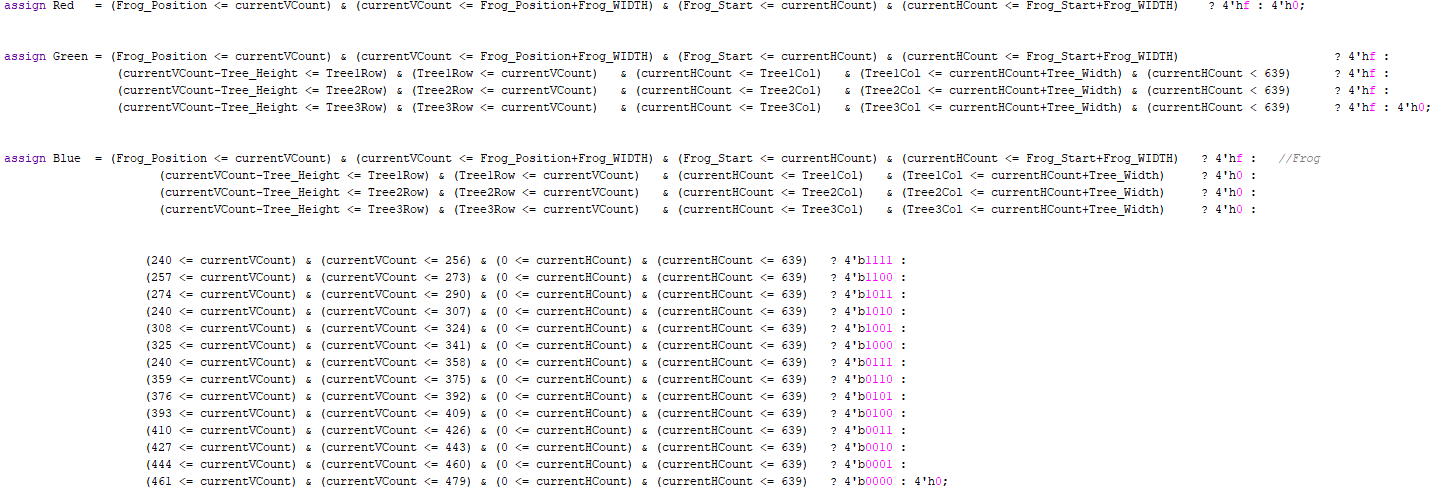
* **Top\_Module\_Main**
  + Inputs: btnC, btnU, btnD, btnR, btnL, [15:0] sw, clkin,
  + Outputs: Hsync, Vsync, [3:0] vgaRed, [3:0] vgaGreen, [3:0] vgaBlue, [15:0] led, [6:0] seg, dp, [3:0] an
  + Implementation: The top module holds the implementation of all of the other modules together. This holds instances of a frog, the VGA values, three trees, the counter (and segment displays), the frame counters, and all the delays for the state machine.

**Figure 1**

* **VSync\_Tracker**
  + Inputs: clk, NextCol
  + Output: Frame, [11:0] Position
  + Implementation: This module uses four 4-bit counters to continuously count upon the clock signal until it is told to reset when the value of the entire counter is greater than 524. When this value is reached, the counter resets, and the output for the Frame would be high for one clock cycle, indicating the end of a frame. Here is a snippet of the code that ensures we have a frame value.

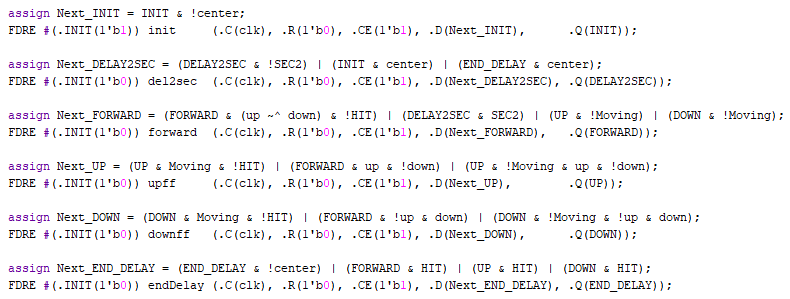
**Figure 2**

* **HSync\_Tracker**
  + Inputs: clk
  + Outputs: RowFinish, [11:0] Position
  + Implementation: This module uses four 4-bit counters to continuously count up on the clock signal until it is told to reset when the value of the entire counter is greater than 798. When this value is reached, the counter resets and the output for the RowFinihs is high for one clock cycle to indicate that a row is finished and that the **VSynch\_Tracker** should move onto the next column. Below is the snippet of code that makes the colors and different shapes appear on the screen to the user.

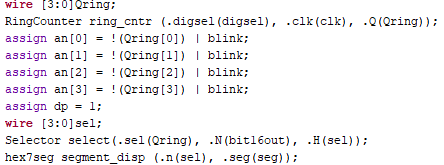


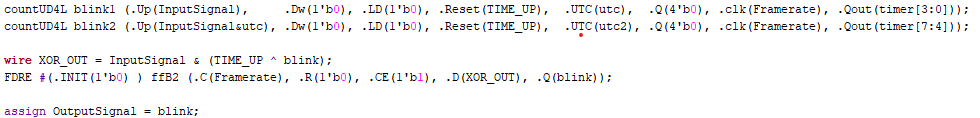
**Figure 3**

* **StateMachine**
  + Inputs: Up, Down, Center, SEC2, Moving, HIT, clk
  + Output: Rungame, INITSTATE, TimerStart2Sec, Frog\_Up, Frog\_Down, Frog\_Blink, Reset
  + Implementation: This module consists of a six-state state machine. The states include an initial state, Delay for two-second states, a forward state, a frog move upstate, a frog moves downstate, and an end delays state. During the initial state, the frog is stationary and nothing is moving or blinking. During the delay for two seconds state, the fog and the seven segment displays are blinking two times for a total of two seconds. The forward state is a state where the trees move (not the frog) forward until the action happens. During the upstate, the frog is given a signal to move up for the entire motion. This state also includes the motion for the frog to move back down to the original state. During the downstate, the frog is given a signal to move down for the entire motion. This state also includes the motion for the frog to move back up to the original state. During the end delay state, the plants stop moving in their original position, the frog stops moving, and the frog and the seven segment display all blink in unison until the btnC button is pressed. When the btnC button is pressed, the frog resets to the middle position, the plants move back to their original positions, and the seven-segment displays reset to their zero values. Here is a snapshot of the code of all the states.

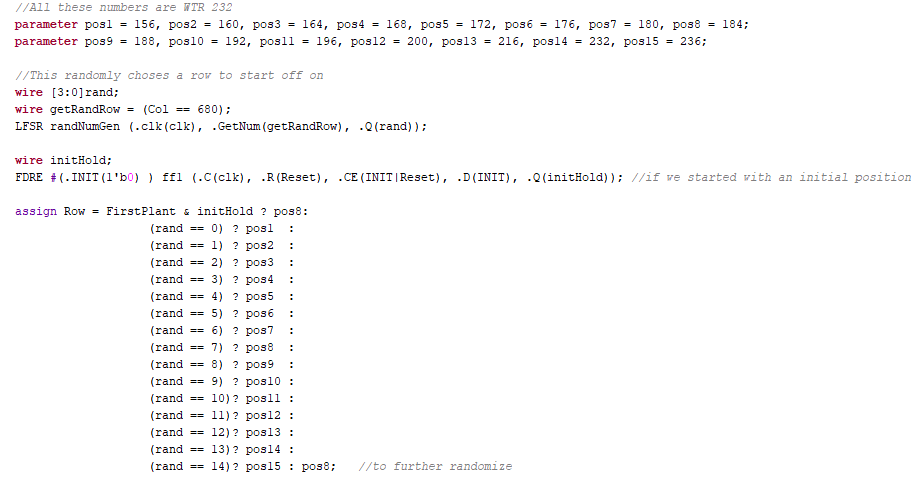
**Figure 4**

* **Blink**
  + Inputs: InputSignal, Framerate, clk
  + Output: OutputSignal
  + Implementation: This module uses two 4-bit counters that measure the number of times the counter counts upon the frame clock cycle. When the timer reaches 64 (about one second) the flip flop is triggered to go to the opposite value using the XOR logic. This ‘0’ and ‘1’ action repeats as long as the InputSignal is high. The output is the current flip flop value. Here is a snippet of the code that makes this action happen. **Figure 5** is the code that makes the seven-segment displays blink, while **Figure 6** is the code that does the actually blinking logic.

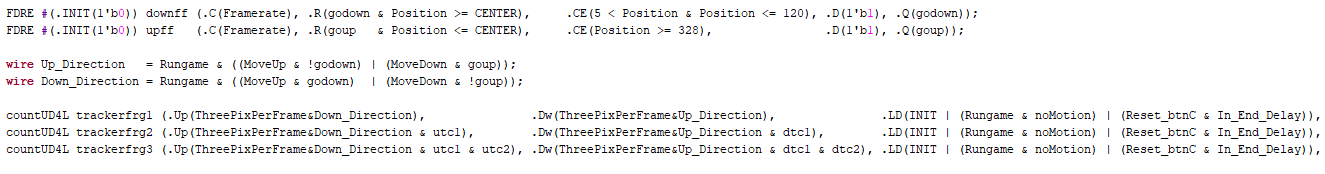
**Figure 5 (Left) and Figure 6 (Below)**



* **Plants**
  + Inputs: Rungame, Framerate, INIT, [11:0] INITPosition, FirstPlant, ThreePixFrame,Clk, ResetPlant, clk
  + Output: [11:0] Row, [11:0] Col
  + Implementation: This module controls the position of the plants. When an instance of a plant is created, the inputs determine if this is the first plant in the sequence of plants. There should only be one of these values HIGH for any string of **Plants** call. The two 4-bit counters count down from their INITPosition values (loaded at the delay state of the game) until the counters reach zero value. When this happens, the counters will then reset to the “12'b001011010101” value, which is used as the reset column value. The row value is randomly chosen using the LFSR values. This module chooses a 4-bit pseudorandom number and selects the index value of that number from a pool of parameters created in the **Plant** module. (see **Figure 7** below).

**Figure 7**

* **Frog**
  + Inputs: MoveUp, MoveDown, In\_End\_Delay, INIT, Reset\_btnC, Framerate, clk, ThreePixPerFrame, FrogBlink, Rungame
  + Output: Moving, [11:0] Position
  + Implementation: This module controls the position of the frog. The MoveUp and MoveDown inputs are implemented in a way that the entire movement of the frog happens in one continuous motion. For example, when the MoveUp signal is high (and stays high), the Moving output is high (signal for the state machine that the frog is not in the resting state and is out of idle position) and the counters count down towards zero. When the counters reach a value of less than 120, the godown flip flop goes high and the counter begins to count back up towards the frog’s idle value position. When the frog reaches its idle position, the counters stop counting, the initial position is loaded into the frog to ensure no movement, and the Moving output goes LOW. This signals to the state machine that the frog has returned to its original position and can continue with the forward state. Without loss of generality, this process is the same when the frog moves in the MoveDown state. Here is a snippet of the code used to make the frog move up and down

**Figure 8**

* **Delay2sec**
  + Inputs: Start, FrameClk, clk
  + Output: Signal
  + Implementation: This module is a daily simple implementation of a 2-second delay. This uses three 4-bit counters to count up to a safe value of greater than 250 and then resets to zero when the value is greater than 250. This number is used because 64 frames are about ½ a second. So 64 frames times 4 half seconds = 250 on the counter value. See the snippet below of how this was implemented.



**Figure 9**

* **Ring\_Counter**
  + Inputs: digsel, clk
  + Outputs: [3:0]out
  + Implementation: See **Design:** Ring\_Counter *Lab 4: Multiplexers, Full Adders, and Seven Segment Displays*
* **Selector**
  + Inputs: [3:0]in
  + Outputs: [3:0]out
  + Implementation: See **Design:** Selector *Lab 4: Multiplexers, Full Adders, and Seven Segment Displays*
* **hex7seg**
  + Inputs: [3:0]in
  + Outputs: [6:0]out
  + Implementation: See **Design:** hex7Seg *Lab 4: Multiplexers, Full Adders, and Seven Segment Displays*

**Testing & Simulation**

During the beginning of the lab, all portions of the lab were able to be tested. As soon as the VGA and the display-only states came into play, it was very difficult to try and test the display as a simulation. For the parts that *can* be tested, there needed to be exact states for every part of the state. Certain inputs were chosen for reasons that every possible state needed to be tested and considered. For example, the state machine in this lab was very similar to that of Lab 5 because the current input of the state machine depended on its own outputs that needed a feedback signal. It was also important to note that some modules used both a frame clock and a system clock. This was later relinquished when using which clocks needed to go where and how they were going to be used (i.e which clock should a counter or flip flop be in sync with: the frame or the system clock?). This is very important when developing because one clock is significantly slower than the other.

**Results**

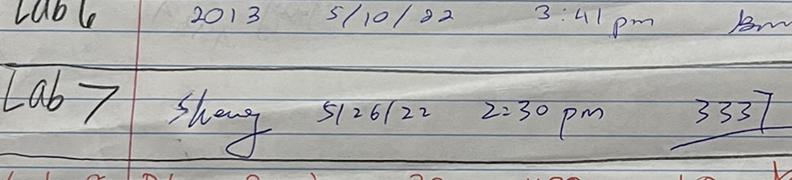
The final result of this lab was a success! The final and initial check off was one of the most important parts that was able to be done successfully. It is important to note that the state machine hardly gave any types of issues. It was mainly the coloring (VGA outputs) that was the issue. For example, the frog and the plants had moments when the green color would disappear. This issue was trying to be resolved for nearly six hours, and would finally be solved. This issue was the result of enabling color where color shouldn’t be enabled off the screen. Because there are certain Hsync and Vsync areas where the RGB values should be LOW, but there are also places on-screen where those same RGB values should be HIGH. This resulted in the RGB values trying to be HIGH and LOW simultaneously, which resulted in a brief discoloration of the frog and the plants.

As this was solved, another issue came up with the randomness of the plant row. This was quickly resolved by enabling the flip flops in the LFSR module, as they weren’t done beforehand. As a result, the flip flops were not active and the LFSR value was the same throughout the program run.

**Conclusion**

This lab, although high in its difficulty, created many new paths to learning. This also brought to light the amazing aspects of VGA and real world applications to a VGA-enabled screen.

**Appendix**



**LAB 7 Signed Off: 5/26/22 2:30PM. CODE:** 3337

Below are the Lab 5 Schematics and Verilog Code.