Code Explanation

This is a basic overview of how I completed my project and what my code does; more detailed explanations are in the comments of my code. Initially, I planned to display a square through HDMI, assuming it would be straightforward. After running into several issues, I pivoted to making a Tug-of-War game, which I later presented in class for extra credit. In my Tug-of-War module, we detect whether switch 0 or switch 15 was flipped. Each switch flip moves the LED (flag) one position in the direction of the flip, making sure to save the data of the previous switch flips to accurately place the “flag” in the row of LEDs. One important detail is that with 16 LEDs, the flag cannot start perfectly centered, there is no single LED with equal numbers of LEDs on both sides. This could be fixed either by using a two-LED wide flag or by ignoring one LED on one side so the flag stops early. Both solutions work, but I chose to leave the asymmetric look. I think it adds another level of complexity to the game, where the previous winner must start with a one LED disadvantage at the beginning of the game.

When the game ends, a win celebration will begin, lighting up all 7-segment displays with the player number that won (all 1’s if player 1 won, and vice versa for player 2), while leaving the flag-LED frozen on the winning side.

After completing the game logic for Tug-of-War, I revisited the HDMI display. My first attempt was to write a VGA-to-HDMI converter myself, but it never worked correctly. After reviewing the Boolean Board reference manual, I saw a tutorial that guided me through the correct process. As described in my hardware setup, the first step was downloading the Clocking Wizard IP. This allowed me to generate two signals at two different frequencies. One is at 25 MHz for the pixel clock, and one is at 125 MHz for the HDMI. The 25 MHz clock updates the pixels, basically saying, “Here is the next pixel to display”. The 125 MHz clock runs roughly five times faster in order to manage internal resources and convert the generated pixel into a signal we can transmit to the display (TMDS format used by HDMI). In Verilog terms, this means we are moving pixel bits across a 125 MHz wire.

This faster clock is necessary because HDMI uses a 10-bit encoded symbol format (TMDS) to represent red, green, blue, etc., instead of an 8-bit signal like the VGA. However, the FPGA system clock operates at 100MHz, not 125MHz. This means we also have to use the FPGA’s clock management tile (CMT) to generate all the required clock signals. Luckily, we don’t have to make all the code that does this; the download clocking wizard from Real Digitals' website gives us a template that we can use to make our converter. Knowing all of this, it makes sense why trying to write this myself didn’t work.

Once the clocking system was correct, I set up the HDMI timing parameters. Even though modern LCD, LED, or OLED displays are not CRTs, HDMI still operates under the old idea of cathode ray tube displays (CRTs). This is where we have an amplitude-modulated moving electron beam (or cathode ray) to display information on a phosphor-coated screen. Which sounds really complicated, but all it is is an electron gun that fires electrons at different parts of the screen very quickly (left to right, top to bottom) to make an image. Although this is very interesting, it unfortunately gives us physical constraints like the time it takes for the electron gun to reposition itself to fire a new beam of electrons at the screen after resetting. To counteract this, we must set up buffer zones that act as an area of the screen for the electron gun to reset. These buffer zones are called front porches and back porches, and are used with a retrace to ensure our signal is synced and tells the monitor that the scanline is finished. So, even though we no longer physically sweep an electron beam, the timing standard remains.

I used a 660x480 active display area and predefined display resolution parameters for front/back porch and retrace: a 16-pixel back porch and a 48-pixel front porch horizontally, along with a 96-pixel clock horizontal retrace, resulting in the standard 800-pixel horizontal line length. Vertical timing works the same way, resulting in 525 total lines. As the "virtual beam" scans across the screen, horizontal and vertical counters track the current pixel location. The counters reset the beam 1 pixel before reaching the maximum value in the horizontal and vertical directions. Lastly, we need to actually create the "virtual beam" in the x and y directions to light up the display. To do this, we need to sync our vertical and horizontal pulses and generate a VDE (Video Data Enable) signal, which is high only when the counters are within the visible display region. This makes sure everything is displayed when and where we want it to be.

Now that we have made our timing system for the "virtual beam", we can start having fun with our display. To do this, I made an image creator module to create the images we want to display. The pixel coordinates (x and y) come directly from the counters, making it simple to draw shapes or make patterns. For example, the checkerboard pattern is generated by XOR-ing bits of the coordinates ((x >> 4) ^ (y >> 4)), switching between black and white based on the result. Most of the other shapes were decently straightforward to make, but the on-screen "flag", sometimes labeled as "triangle" or "upside-down_triangle" in my code, was much more difficult. I generated it using what I call the "chiseled square" method, where each scanline chisels off a pixel early on each side based on the previous line of the triangle, forming a triangular outline from a square.

Since I originally wrote the game logic and display logic as separate modules. I now needed to connect everything in order to move the "flag" on screen and make the "flag" blink when the game was over. To do this, I made wires like "game_over" to communicate what was happening in one module to another. Once linked, I mapped the LED flag's position to the screen so the display matched the Boolean Boards LEDs. I could now also display an "end game" effect when a player won, since I noticed it was hard to tell when the game was over on the HDMI display.

I wasn't entirely sure how to do this since I'm somewhat limited by how complex my code can be in Verilog, so I chose a simple endgame effect suggested by James Ferguson: blink the on-screen flag. I first implemented this by using a counter that toggles the blink state on and off every half second, and then I added a condition that determines whether to draw the flag based on the blink state and whether the game is over. If the game is still in progress, the flag is always displayed (since the blink counter continues to run in the background), and if the blink counter is active and the game is not, then the flag is on; if neither condition is true, the flag is not drawn. I also decided to only update the game logic every 100Hz. This helped get rid of odd glitches in my display and also makes sure that any sort of switch or button debounce isn't an issue. I also thought it was an interesting application of different clocking abilities that the FPGA has that I didn't get to play around with earlier when making the clock for the HDMI display.

Hardware Setup

In order to get my code to work, you must first download a VGA to HDMI clocking wizard IP off of Real Digitals website: https://www.realdigital.org/doc/715356000ec89fbfd26a44cd2444659b and follow the instructions to set it up. After that, you must also setup another Clocking Wizard using the default IP catalog under the “clocking” tab. Instructions on how to do this are on Real digitals website: https://www.realdigital.org/doc/ae6ce4fbf81065307776fd9d0911ec7d
For the clocking wizard, clock ones output (clk_out1) must have the output frequency requested set to be 25.000 (MHz) and for clock twos output (clk_out2) you must set the output frequency requested to be 125.000 (MHz). Also make sure to open all the IP catalogs in the sources tab and let them generate templates as well as show their hierarchies. After that, click on the main Verilog file and generate synthesis. It may give “Spawn errors” for your first synthesis, if it does this, just ignore it and try again.