**Goals**

Over the Summer of 2023, I was looking for a project that could help me practice reading datasheets, implement more complex state machines, and provide a useful module to help me take advantage of other FPGA projects. After spending some time thinking of ideas, I landed on using Digilent’s 96x64 OLEDrgb display PMOD. This display uses the SSD1331 OLED controller, and the PMOD is wired to be used as a write only SPI interface.

To utilize this PMOD, my goal was to create an array of 8x8 pixel ASCII characters, that could be stored or updated as ASCII values. This could be converted from binary, hex, bcd, or scanned through something like UART. It would be very straightforward to convert the values into ASCII, as long as I could keep that ASCII array as a top level input and treat the OLED controller as a hands off display after it is tested. I would also like to be able to use inputs to change the color of the text and background, to make the displays more dynamic based on different states of the logic modules it works with.

The following requirements are to be followed to interface with the OLED controller:

* The Master driven clock SCK shall have a frequency less than 6.66 MHz
* The Chip Select pin CS to the OLED slave controller must be pulled low before transmission
* Data on the MOSI line will be read on the positive edge of SCK on the OLED Controller
* A command code will force D/C to 0, while a data byte will force D/C to 1
* VCCEN and PMODEN must be set to 1 after an appropriate delay to turn on the OLED controller
* The command code 0xAF must be sent to power ON the display, followed by a 100ms wait

The above requirements are determined based on the reference manual for the SSD1331 OLED display controller and the schematic for the OLEDrgb breakout PMOD from Digilent. This would dictate the turn on sequence for the OLED controller, and lead me to designing a standard write only SPI module, which would handle driving the CS and MOSI pins.

The following requirements were determined by myself to achieve the goal of a fully controllable ASCII display:

* The full range of the display will be utilized, being 96 columns and 64 rows of pixels
* Each ASCII character will have 8 columns and 8 rows, or 64 total pixels
* The ASCII values will be stored as ASCII types, and converted to the 8x8 bit array through a case statement
* The ASCII values will be displayed one at a time, modifying the start and end addresses to display each 8x8 bit block in succession
* The ASCII characters will be displayed row by row, starting at column position 0 and incrementing to the right
* The ASCII text shall be one set color, and the ASCII background shall be another color, as inputs to the OLED Interface controller

The following set of modules were planned to be implemented, to achieve a fully customizable ASCII display in real time:

* Nbit\_MOSI\_SPI: A shift register will handle transmitting 1 byte of data over MOSI and driving CS, on the negative edge of SCK
* Nbit\_MOSI\_SPI\_Buffer: Another module will load multiple bytes of data and D/C bits to send strings of commands in a row, updating on the positive edge of SCK
* OLED\_interface: An OLED Controller module will be created to handle the turn on, turn off, and display states by writing to the byte buffer
* Ascii\_font\_8x8: Decodes 1 byte ASCII type into 8x8 bit array of pixels, with MSB being the top right pixel, LSB being the bottom left pixel, row by row

**Process**

Like my other previous FPGA home projects I have worked on, I used my Basys-3 Digilent FPGA Development board for this project, which includes an Artix-7 Xilinx chip. I modified my project structure because I realized how much of a mess the premade Vivado structure was when I didn’t define a source folder to reference. After taking senior design and trying to document more work, I also included PDF’s of reference manuals and Word documents of my own notes as I was researching this project. This is something that I think has really improved through school, and it’s great to see how much time it can save me now and in the future.

The first module that was designed and tested was the Nbit\_MOSI\_SPI module, which acted as a shift register to output 1 byte over the MOSI SPI line. The FPGA would act as the master to the OLED controller, meaning it also had to control the CS pin. Once a transmit begun, the CS pin would be deasserted to 0 on the negative edge of the SCK clock cycle. Since the CS pin was active low, this indicated that on the next positive edge, the OLED controller would read the first MOSI bit and the D/C control. In the transmit state, once the second to last bit was transmitted over MOSI, a flag would be raised indicating the next bit would be the final bit sent in the byte. This would give the opportunity for another byte to be loaded while still staying in the transmit state, leading to one less clock cycle for every byte transmitted. Inputs such as i\_START were included to control when the master would transmit, and would be driven by modules upstream. With this module, I was able to satisfy multiple requirements I had to follow and set for myself, including controlling the CS and MOSI pins of the SPI protocol.

The next module that was implemented and tested was the Nbit\_MOSI\_SPI\_Buffer. This module used a similar counter and state machine to the bit level transfer mentioned before, but instead acted on the positive edge of SCK instead, ensuring timing constraints would be met between the two. This module worked to drive the START signal for the previous module, and would load multiple sets of bytes to continuously transmit over the MOSI output of the FPGA. This module would use the second to last bit flag to determine if another byte should be loaded based on a counter and the input number of bytes to transmit. Again, this was beneficial since it ensured constant output on the MOSI line without losing clock cycles in between waiting. Like before, a flag was added to indicate when the last byte was being transmitted, so the OLED controller module could determine when to load the next set of command or data bytes.

The OLED\_Interface module is where everything came together. This would take in the multiple byte ASCII array of the entire display, and display it pixel by pixel. Before this could occur, the FPGA had to go through a turn on state, where the reset pin was applied, the power enable pins VCCEN and PMODEN were asserted, and the display ON command was set through the Nbit\_MOSI\_SPI\_Buffer. This module included a clock divider to step down the 100MHz clock on the Basys3 to 5MHz, under the maximum frequency for the OLED Display. Another module was included, ascii\_font\_8x8, which was a case statement to decode 1 byte ASCII value to an 8x8 bit array that would contain the pixel display for each respective ASCII code. Each ASCII code would be displayed in the pixel\_display states, where the first state would set the start and end row and columns to an 8x8 bit square, offset by the ASCII column and row counters. At the start of a new ASCII code, while the row and column limits were being set, a 64 bit register would store the converted ASCII code in the 8x8 bit pixel format. A counter and shift register were used to display all 64 pixels concurrently, each holding 1 byte of RGB data, determined by the text and background color inputs.

Finally, to apply the ASCII display to an old project, I decided to use the reaction timer module that I had made around December 2022. This timer was initially used with four seven segment displays, and would require you to press a button after an LED showed up, within one second. If you pressed the button before the LED showed up, or after one second, you would fail the game. This was able to be customized much more with the ASCII display, since I could change the background color to make the reaction easier, or indicate when a success or failure from the game occurred. This required me to create a module to convert hex values that were intended for a seven-segment display to ASCII characters, which could then be updated to the display in real time. I also added additional outputs from the reaction timer for the state of the machine and what the outcome of the game was, being either a success, early failure, or late failure. With this, I was able to use a new interface to enhance an old design.

**Reflection**

Overall, I think this project provided a great sense of challenge and was well scoped for my time over the Summer between taking in depth classes on computer architecture, FPGA development, and embedded applications. While I achieved a more complex project through synthesizing a MIPS processor in my computer architecture class last semester, that was under the guidance of a professor, teaching assistants, and a lab partner. This project helped to teach me where I was at with my own fundamental understanding of digital design, my skills in decomposing reference manuals, and my discipline to follow through a project that gave me trouble. I would like to think I have done pretty good documenting everything I have done since I began this portfolio and my YouTube videos, but I always challenge myself to deliver my thoughts, ideas, and mistakes and a more concise and impactful way. Hope you learned something along the way with me! The following lists are some key points I would like to highlight on this project.

Things that went well:

* SPI Write only interface implementation
* Notes from reference manuals and PMOD schematic
* Flexibility with generic parameters

Things that could improve:

* More documentation for end of project
* Could be more deliberate in state machines to avoid potential latches
* More automated testing would be nice

Things I had fun with:

* Working through the state machine to display multiple ASCII characters
* Using RGB values to update text in real time
* Enhancing an old design with more accessibility and information