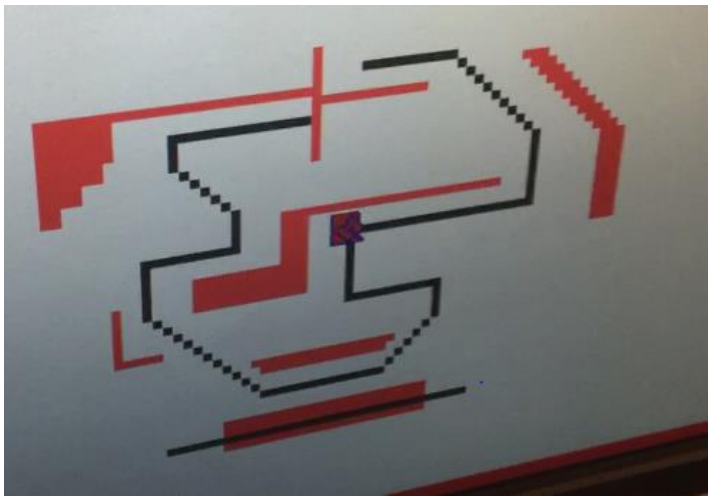
Hai Dang Hoang
Brendan Ball
SoC Design ECE 540
Project 2

Theory of Operation: VGA Simulated Line Following Rojo-Bot

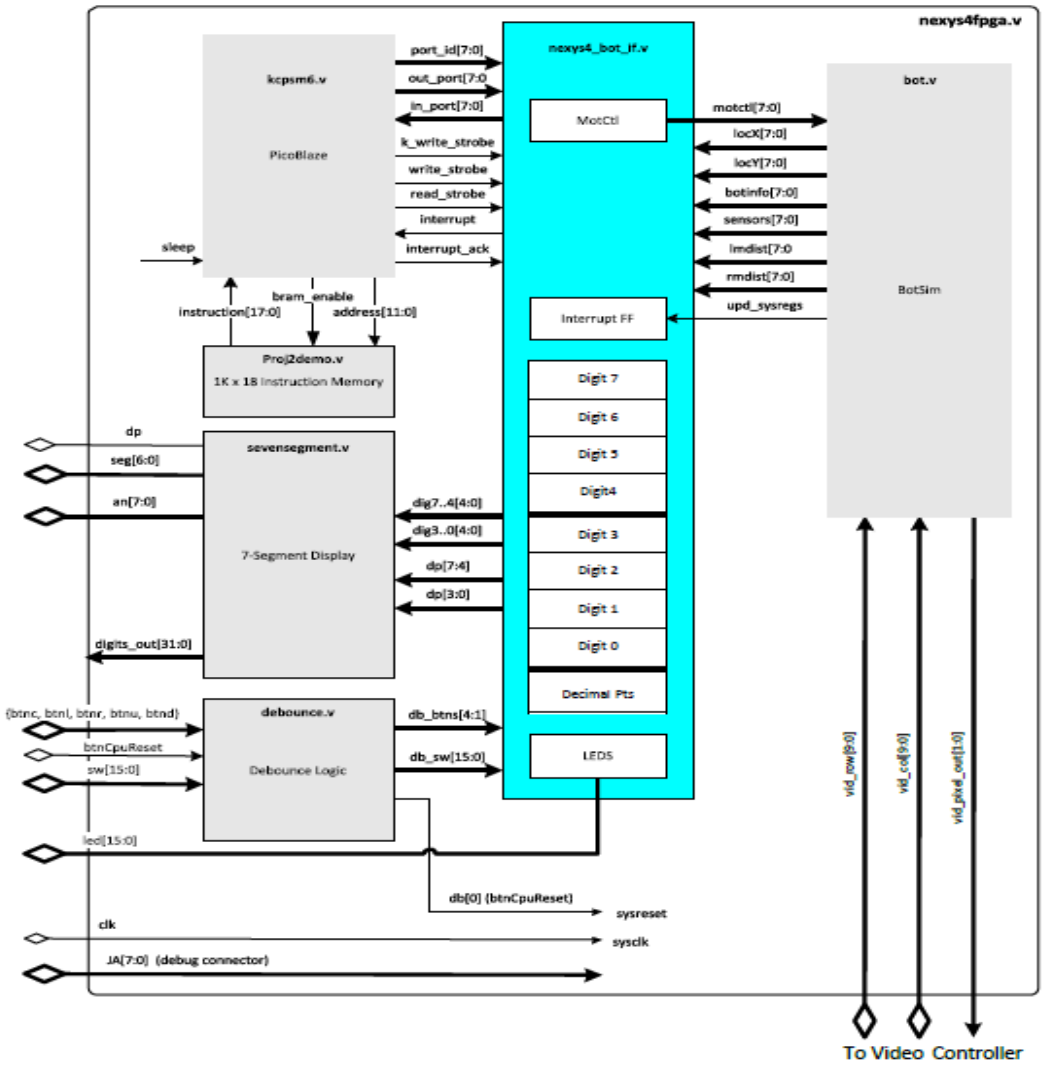
I. Introduction

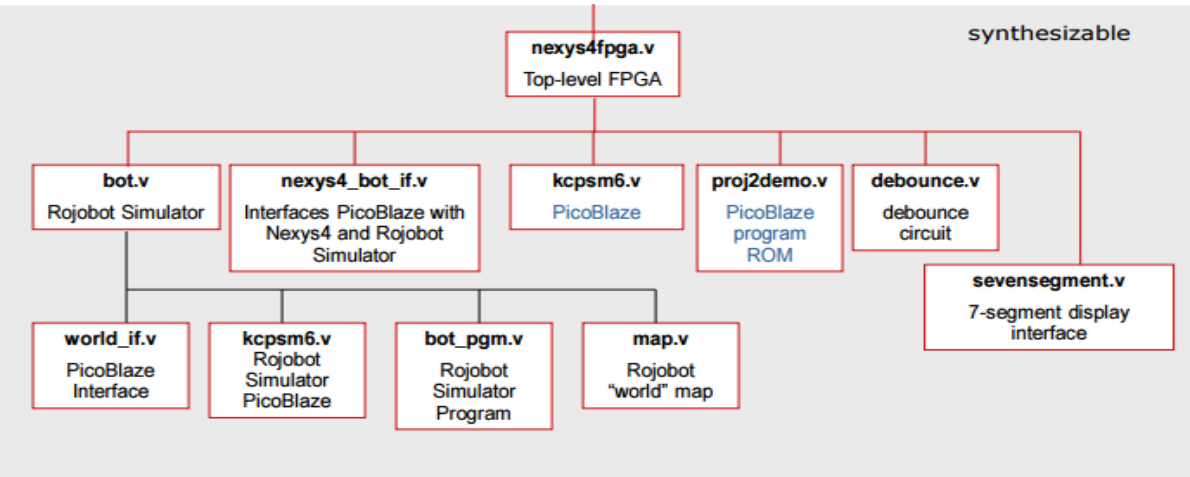
The purpose of this project was to utilize various FPGA constructs to interface blocks of Verilog logic, an embedded CPU executing PicoBlaze Assembly Code and a VGA video controller in order to generate an onscreen virtual robot capable of following different lines through a computer generated world. There was also an intermediate step where a demo program was instantiated, run and displayed on the NEXYS 4 DDR Board. The push buttons were also utilized to control the robot. This was done as a stepping stone towards the final goals of the project and helped the team to better understand the control and flow of information between modules. The main tasks performed included: creating an I/O block for interfacing the BotSim PicoBlaze Program and its program memory; instantiating, synthesizing and running the demo system; modifying the Demo Picoblaze Program to implement the wall avoiding, black line following program; and Designing, implementing and debugging the Rojo-Bot world video controller. The last part included creating colorizer and icon modules to generate the expected images of the Bot successfully navigating the track.



II. Procedure

The block diagram below shows the design hierarchy for the project:





(Reference: Professor Roy Kravitz)

As mentioned above the first step towards completion of the project was the integration and instantiation of the included robotic simulation. For this purpose an I/O block was created to route the data coming to and from the PicoBlaze design block that contained the assembly code logic necessary to run the demo simulation. The logic contained in the I/O block essentially works like a register or memory. It takes in port address data and uses it to determine what information must be placed on the data in and data out buses. All of this information is received from a module named Kcpsm6.v, also included are interrupt, interrupt acknowledge, read and write strobe and constant write strobe bits, for handshaking purposes.

Our top level module and a constraint file containing pin location data allowed for this data exchange between our program and the real world. To make all of these connections work properly all modules had to be instantiated in the top level module. This set up allowed inputs and outputs from one module to be shared with others and also the outside world. Top level inputs and outputs contain just the variables that are going to be sent and received from outside the chip. All other data passed from module to module is internal and given the data type of wire in this module. Although this sounds like a lot of work it really wasn't. Understanding how everything was organized was the most time consuming part this meant identifying what each piece of information represented, where it was needed and which blocks were responsible for sending and receiving it. Once this was established it was just a matter of copying port addresses from the PicoBlaze program to the I/O block and instantiating the lower level modules into the top level. Everything else regarding the programming logic was already done.

Completing the demo did not exactly make the next portion simple but it did make the task of completing the RojoBot world video controller a lot less difficult. Much of the data including map and sensor information from the bot.v program could be re-used, however, instead of using the push buttons to control the Bot and the seven segment decoder to "see" it, we now had to come up with a way to have it algorithmically move through the map and check if it was in fact following the determined path on a VGA controlled video screen. To do this we had to create a computer generated representation of the Bot that for display on the screen, this image had to be superimposed on a world background that would contain the map information, this included a black line path, obstacles or walls and open spaces. A colorizer program needed

to be created to display the open space (background), wall and black line colors on the screen. It was also necessary for this colorizer program to display the bot icon over the top of these images. The VGA image had to be updated as the bot moved. Since the clock speed used by the VGA controller is slower than the 100 Mhz clock used for the previous project adjustments had to be made.

In summation the main work done had to include writing IO interface, creating colorizer and Icon modules, editing the PicoBlaze assembly demo code in order to generate the black line following algorithm and generating a clocking structure that would allow the VGA video controller to obtain pixel data at the write speed.

III. Implementation

a) Picoblaze I/O interface (module nexys4_bot_if):

To interact with the external environment, a regular microcontroller chip consists of a variety of built-in I/O peripherals, such as buttons, switches, 7-segment decoders, PicoBlaze etcetera; in short, it provides a simple generic input and output structure for an I/O interface. I/O peripherals are added as needed and thus are customized to each application. PicoBlaze uses the input and output instruction to transfer data between its internal registers and I/O ports, and its interface consists of the signals shown on the table below.

(Reference: FPGA Prototyping by Verilog Examples By Pong P. Chu

Format template I/O interface from professor Roy Kravitz)

I/O Register Map:

Port ID	Op	Nexys4	Name parameter
0x1	Rd (I)	slide switches	PA_SLSWTCH
0x2	Wr (O)	Leds	PA_LEDS
0x3	Wr (O)	Digit 3	PA_DIG3
0x4	Wr (O)	Digit 2	PA_DIG2
0x5	Wr (O)	Digit 1	PA_DIG1
0x6	Wr (O)	Digit 0	PA_DIG0
0x7	Wr (O)	Decimal point	PA_DP
0x8	Wr (O)	*Reserved	PA_RSVD
0x9	Wr (O)	Motor control	PA_MOTCTL_IN
0xA	Rd (I)	X Coordinate	PA_LOCX
0xB	Rd (I)	Y Coordinate	PA_LOCY
0xC	Rd (I)	Rojobot Information	PA_BOTINFO
0xD	Rd (I)	Sensor register	PA_SENSORS
0xE	Rd (I)	Left motor dist	PA_LMDIST
0xF	Rd (I)	Right motor dist	PA_RMDIST

0x11	Rd (I)	Slide switch 15:8	PA_SLSWTCH1508
0x12	Wr (O)	Led (15:8)	PA_LEDS1508
0x13	Wr (O)	Digit 7	PA_DIG7
0x14	Wr (O)	Digit 6	PA_DIG6
0x15	Wr (O)	Digit 5	PA_DIG5
0x16	Wr (O)	Digit 4	PA_DIG4
0x17	Wr (O)	Decimal Point 7:4	PA_DP0704
0x18	Wr (O)	*Reserved	PA_RSVD_ALT

Data from the bot.v program encoded in these data in, data out values using the I/O block as a reference included a single byte motor control output, two byte long location inputs, one for the x coordinate and one for the y coordinate, a robot information register output (more on this later), a one byte virtual sensor data input and some inputs indicating the distance traveled by the Bot which were not utilized. This information was encoded to display outputs on, and receive inputs from, the Nexys4 board. The Bot's location emanating from the map data contained in the bot.v instruction memory had to be displayed on the board's seven segment digits, the motor control data which was coming in from the board's push buttons was utilized to update the location and direction of the virtual Bot.

General Purpose Input Ports:

The inputs connect via a pipelined multiplexer. For optimum implementation, the input selection control of the multiplexer is limited to only those signals of 'port_id' that are necessary. Note that 'read_strobe' only needs to be used when whatever supplying information to KCPSM6 needs to know when that information has been read.

General Purpose Output Ports:

Output ports must capture the value presented on the 'out_port' based on the value of 'port_id' when 'write_strobe' is High.

Interrupt interface:

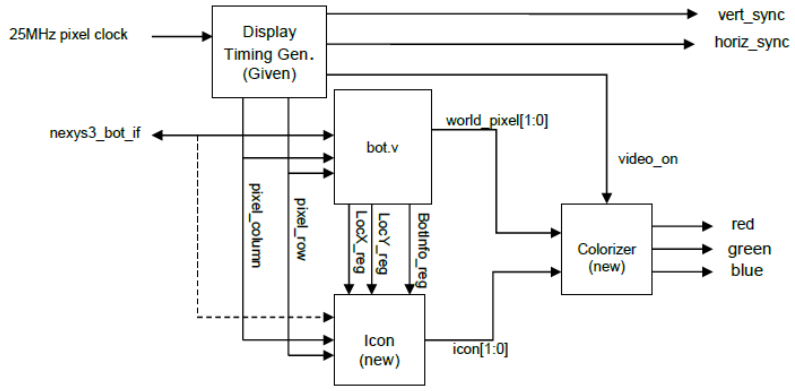
Interrupt becomes active when 'int_request' is observed and then remains active until acknowledged by KCPSM6

b) *RojoBot World Video Controler: (module Colorizer and module icon)*

One of the primary outcome for this task is to design and implement video. In particular, we will implement an animated and colorful graphical display for the RojoBot world that displays an image of the RojoBot virtual world with the Bot moving around on it. The icon moves through the RojoBot world based on the current {X,Y} coordinates of the Bot.

These coordinates are returned by BotSim in the LocX and LocY registers. The orientation is returned in the BotInfo register. A high-level block diagram for the display subsystem is shown below.

Display Subsystem Architecture

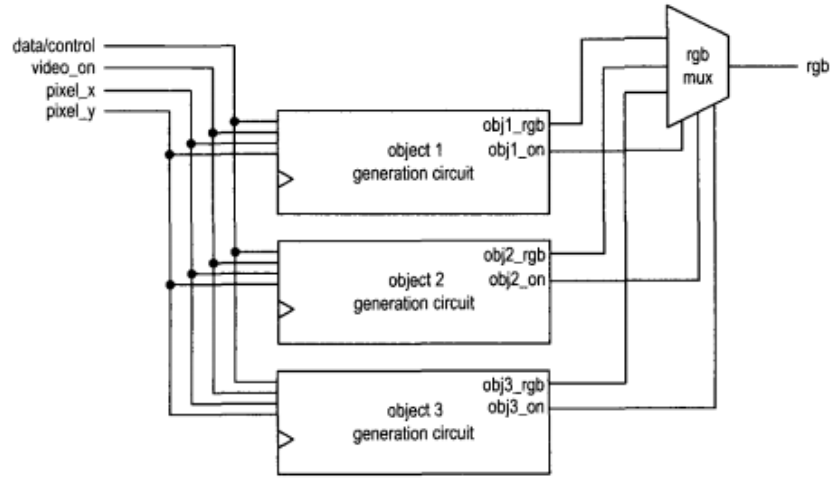


Reduced Clock Speed Generator:

The clock speed for the VGA control module was 25 Mhz this needed to be accounted for since the original demo program was running at 100 Mhz. Our team chose to use an IP block called clock wizard to generate the slower clock. Since nothing in the design required such a fast clock we chose to use the 25 Mhz clock to synchronize our entire design. The clock wizard made this very easy. All that had to be done were a few mouse clicks in Vivado which pulled up some menus regarding how fast out new clock should be. A clock module was automatically created by this procedure. This module was then instantiated in the top level with the original clock as inputs to the module and the new clock as outputs from it.

Colorizer:

A high-level block diagram for the display subsystem is shown below.



(Reference: FPGA Prototyping by Verilog Examples By Pong P. Chu, Chapter 13)

The colorizer module was fairly straight forward to create. Its purpose was to take information regarding what color needed to be displayed from the icon.v and world.v modules and send it to the "external" VGA video controller. The block diagram illustrates the idea about object-mapped pixel generation (Colorizer) with object 1: world map, and object 2: tank icon. The rgb mux circuit performs multiplexing according to an internal prioritizing scheme. The prioritizing scheme prioritizes the order of the displays when multiple object signals are asserted at the same time. It corresponds to selecting an object for the foreground. Its function is defined in the following table:

World[1:0]	Icon[1:0]	Color
00	00	Background
01	00	Black Line
10	00	Obstruction
11	00	Reserved
x	01	Icon color 1
x	10	Icon color 2
x	11	Icon color 3

In addition, in order for the proper value to be sent to the VGA controller the module dtg.v was utilized to keep track of which pixel on the 480 x 640 pixel display's turn it was to be "printed" on the screen. This module which is also known as the display timing generator is like the conductor. It uses the same clock as the VGA controller and keeps track of when and at what location the next nibble and byte of information needs to be provided to the VGA video controller. The dtg module was included with the release of the project. It provided a horizontal and vertical location of the pixel being drawn and a bit that contained information as to whether the "external" video module was on or off.

The VGA controller is built to send data to the display three-color pixels at a time from right to left and top to bottom. These values arrive as a 12-byte combinations creating a four bits code for each color. The combination of these three, four-bit color-containing pixels generates what the human eye sees as the actual single color on the screen. This means that there are 4,096 color possibilities per pixel. Each of these pixel values provided by the colorizer was based off the information it received indirectly from the dtg.v module and directly from world.v and icon.v. In both instances this information was received as a two bits value. From the world.v module these values indicated if the pixel should represent the black line Bot path, a wall, or an open space, the last bit pattern was reserved and not used for this project. From the icon.v module came two bits that either represented one of three colors or a double zero for

transparent. If the coordinate that was about to be placed on screen at that time was not one that would contain the bot icon the map bits would be used to print the color that was representative of that space. To determine whether or not that was the case the icon program's output was tested to see if it was zero for transparent or another value. Because there were only six different possibilities of what value could be sent to the VGA controller our design was limited to just six colors. We chose red, white, blue, black and green. The shades of which were varied of course.

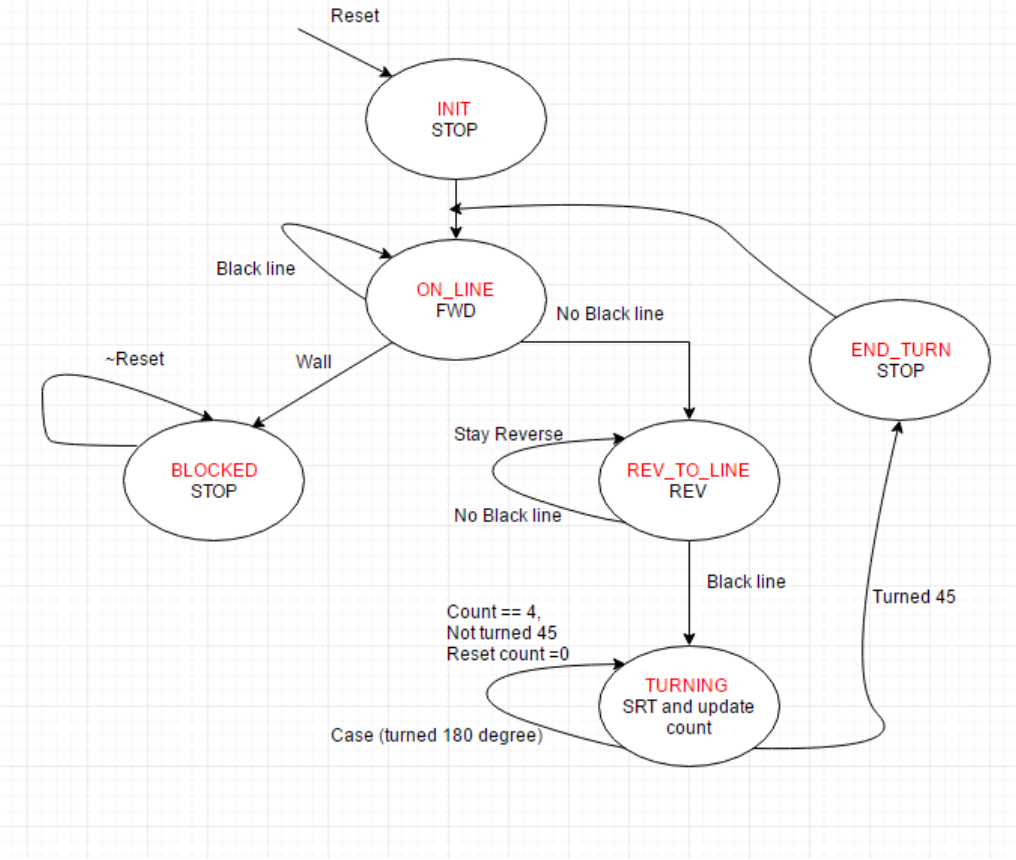
Icon Generator:

As stated above the icon.v module was used to send data to the colorizer module as to whether or not a pixel of our virtual bot was located on the pixel to be printed. This was done by taking the X,Y location data of the bot from the bot.v program and comparing it to the horizontal and vertical "pixel to print" data coming from dtg.v. The icon was designed to be 16x16 pixels. Since the pixels are displayed on screen from left to right the bot icon provided the its pixel color data in sequences of four. The world.v data would be read in until the bot location minus two in both x and y directions (to keep it centralized) at which point the bot's first four two bit combinations of pixel data would be read out. Then another 456 pixels of the map would be read out to the colorizer followed by the next set four of the Bot's pixel values from the next row. This process was repeated until the entire Bot icon was finished being read. We chose as our icon a tank. Our first choice was Super Mario, but the image became too distorted when attempts were made to indicate he was going diagonally. This leads us to the next portion of the inner workings of icon.v. Since our goal was to be able to visualize the bots direction and progress a different icon needed to be generated to represent the different orientations of our Bot. Because it was able to turn 45 degrees at a time this meant a maximum of eight different pixel patterns could be created. In fact there was some discussion of proceeding in this manner. The original plan was to have Mario go east and west and the tank go the other directions. In the end it was determined it would be all tank. Attempts were made to obtain picture online and convert them with Matlab to they're constituent pixel values. Unfortunately none of those attempts came to fruition. There was a point at which it was determined that failing to start hard coding the pixel data by hand due to an overly optimistic belief that the "easy" way could work might cause the deadline to be missed. Much time was spent hard coding all eight directions for the tank. Some time was save by reversing, translating and flipping rows and columns of our design, however it was still a fair amount of data entry since each icon was composed of 256 points. In the end it turned out that only two shapes, diagonal and straight were really needed. This was because the location data could be altered such that the pixels were read in in different

orders. For example if the tank was meant to be facing downward the location data for the bot could be flipped and read in in the sequence where the x and y locations were flipped.

c) Black Line Following Algorithm

A flowchart show black line following algorithm (Right turns only)



At the beginning, Rojobot is at initial state (STOP). It changes to state (go forward) without condition. On the state (go forward), Rojobot first needs to check that it is still on a black line. If it is still on black line → Stay forward, it also checks whether it hits the Wall.

If it hits the Wall → go to state blocked (STOP). It changes states until reset is on.

Else → Keep staying forward

Else (Not on the black line) → Go to state Reverse to line (REV). It needs to check that after reversing it is still on a black line.

If it still is not on the black line → Keep staying reverse until get on the black line.

Else (On the black line after reversing) → Go to state turning. In this state, it update counts to detect whether Rojobot turned 180 degree by checking the count. If count == 4 (It turned 180 degree) turning 45 degree without checking black line, and then reset the count. Else (count != 4) → go to state end turn, and then going back to state forward.

IV. Conclusion

Overall this project was quite successful, despite the amount of documentation there was to wade through it was not as complicated as it seemed on the surface. Our team was able to learn some more important lessons about FPGA programming and interfacing PicoBlaze assembly code into a design. It was important to see an entire system working together despite being written with both firmware and software procedures. It was also important to gain experience writing a comprehensive I/O block and instantiating it along with those which were needed to run the PicoBlaze software. We also learned a lot about how video images are processed and produced on a screen. This will definitely be a useful skill as we continue our educations.

Reference:

FPGA Prototyping by Verilog Examples By Pong P. Chu

Lectures, slides, template, design, and example code from professor Roy Kravitz