6.111 Final Project
Soil-Water Characterization Unit

Benjamin Mussi
12/14/2005

Abstract
A portable unit for measuring soil-water levels was designed and constructed. The system is comprised of an LCD, a GPS, an FPGA, ADC circuitry, and a new sensor that measures the resistance of soil at specified depths. The resulting design handles a variety of functions beyond water detection, including automation of position recording using a GPS and ability to configure the sensor for use in any soil type.

Table of Contents

List of Figures

List of Tables

Project Overview

Many communities in the southern United States mandate water distribution sampling when sprinkler systems are installed in lawns. After installation, a technician places small cups around the lawn and runs the system through a watering cycle. The level of water collected in each cup is recorded along with each cup's location. Figure 1 shows a planning chart for use when recording cup locations. The obtained data is first used to compute an average water distribution figure and next to detect outlying measurements. System watering times and individual sprinkler heads are adjusted accordingly to achieve optimal distribution uniformity.

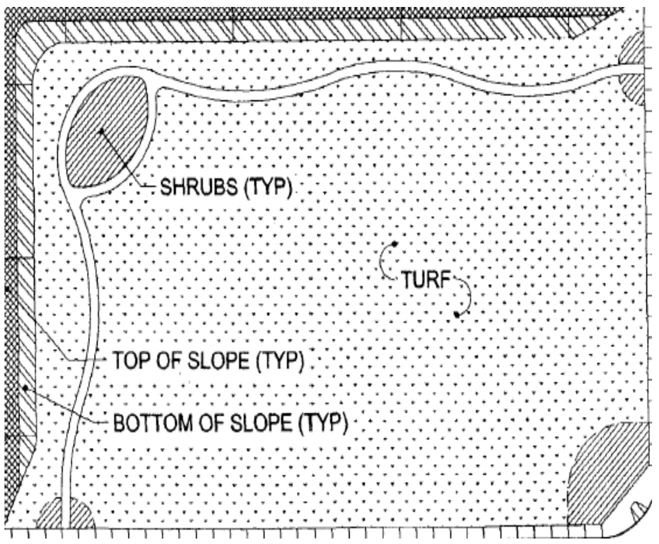

Figure 1 – Planning chart for current method of water distribution sampling

Although the current method of sampling water distribution produces healthier lawns and reduces sprinkler runoff, it has several shortcomings. First, because the process of placing cups and recording the necessary data is fairly labor-intensive, a typical distribution sampling may require as much as a day for a technician to setup and compile. Secondly, cup locations and the water levels in each collection cup are measured manually, introducing human error. Finally, although the current method accurately describes the water distribution above ground, it can only be used as an indirect measure of water penetration into the soil. Penetration of the water into the soil is a more important measurement of water distribution for lawns because soil types vary widely and have different porosities and permeation rates.

The soil-water characterization unit seeks to improve upon the current process of water distribution sampling in soil. The battery powered unit is capable of electronically sensing the depth and level of water saturation in any soil type. In addition, a GPS installed inside the unit allows for automation of recording measurement position information. In this manner, human error in the measurements is greatly reduced. Also, the original sensor installed in the unit allows for a direct measurement of water permeation into soil instead of relying on empirical formulae. In addition, a convenient user interface displayed on a five-inch color LCD uses four buttons to respond to user inputs and control the unit. These improvements dramatically reduce the total time required for completing water distribution jobs. Figure 2 illustrates a conceptual overview of the soil-water characterization unit.

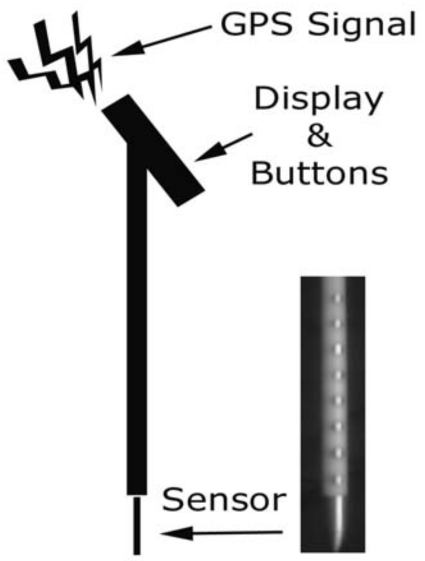

Figure 2- Conceptual overview of the soil-water characterization unit

When powered up, the unit's LCD displays the graphical user interface. Settings are controlled from the labeled box in the top right corner. The first item listed in the settings box is GPS Com., an abbreviation for GPS Communication. Red symbols are displayed to indicate the state of the GPS communication status. A symbol guide is shown in Table 1. At the bottom of the settings box, the active sample number is displayed. The user may press the next sample button and the active sample number is incremented. A new sample may be stored to any of the 32 sample numbers with unstored data. Alternatively, a measurement may be reviewed for a previous sample by selecting that particular sample number. The settings box also displays the active zero setting that calibrates all measurements to the soil type being tested. To calibrate the unit and update the zero setting, the probe must be completely inserted into a dry sample of the soil to be sampled. By pressing the zero button, the user activates the sensor, which measures the resistance of the dry soil and uses this as its basis for saturation measurements.

Table 1 - Default timing parameters

GPS Com. Display	GPS Status	Activity
>>	Idle	GPS Connected
>>>>	Receiving Start of Transmission	Initiating update of position, date, and time
>>>>>>	Recognize date and time packets	Updating date and time
>>>>>>>>>	Recognize position packets	Updating position

The remainder of the display provides results and records of individual samples. The top left corner of the display supplies position information and the bottom of the display gives individual measurement results in a bar graph form. If an empty sample number has been selected and the sample button is pressed, the GPS updates the current date, time, and latitude and longitude position in the position box. Similarly, the bar graph on the bottom updates with seven bars representing the saturation level of the soil at seven evenly spaced depth intervals. If, however, a recording has already been made on a particular sample number, the display provides the stored position and measurement information. Pressing the sample button while viewing a stored measurement will replace the old position and measurement information with new data.

The sensor uses digitally controlled switches and an analog-to-digital converter to make seven measurements of the soil at half-inch increments to a depth of 3.5 inches. The switches are controlled to sequence the transmission of a voltage across a pair of pads on the sensor when activated by the zero or sample buttons. When water is not suspended in the soil, the soil will act as a large resistor and allow for a large voltage difference to exist across a voltage divider. If water is present, the soil will allow some current to pass proportionally to the amount of water present and will act as a smaller resistor. With a smaller resistance, a small voltage difference is present across a voltage divider. These voltage differences are measured with an ADC that converts the analog voltages to a digitally proportional number. The soil is sampled from the bottom of the sensor to the top in an incremental manner and is sampled multiple times per user request to average several measurements and obtain precise results.

Implementation

The soil-water characterization unit was implemented using the Spartan 3 FPGA as the digital controller. The controller was programmed in Verilog and was stored into the Platform Flash non-volatile memory for loading at initial power up. The controller drives a five-inch color LCD in 640 by 480 resolution, receives GPS data in NMEA format, and generates control signals for the sensor circuitry. Four buttons allow for complete control of the device.

Hardware Description

The soil-water characterization unit is comprised of five main hardware components as shown in Figure 3. Any battery or voltage source between 7.5 and 24 volts supplies power to the device, which is converted to five and 7.5 volts using diodes and power-resistors.

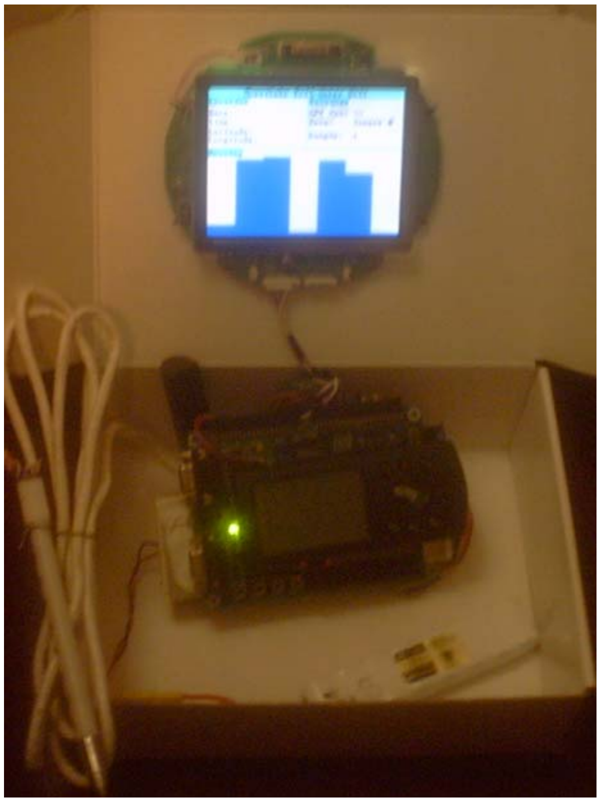
Figure 3 – Soil-water characterization unit powered by battery

Five volts DC are supplied to a Spartan 3 FPGA running at 50 MHz. The Spartan 3 board also includes four buttons and ample expansion ports for connection with other hardware components. A Garmin GPS II is provided with five volts and its data output is connected to an expansion port on the Spartan 3. The sensor controlling circuitry is also supplied with five volts and interfaces the sensor to Spartan 3 expansion ports. The circuitry consists of four CD4066BCN switches used to control the routing of voltages sent to the sensor pads as well as an ADC0841 for converting analog measurements to digital measurements.

The 5-inch color LCD is provided with 7.5 volts. It operates in 24-bit color in an interlaced manner with a resolution of 640 by 480 pixels. The LCD receives composite-sync and RGB VGA signals. The LCD is backlit with a fluorescent lamp controlling brightness and contrast. Figure 4 shows the graphical user interface that is displayed on the LCD.

Figure 4– GUI implemented on soil-water characterization unit

Controller Description

The controller is implemented on the Spartan 3 board using the Xilinx XC3S200 and populates over half of its 200,000 gates. The controller was loaded onto the Platform Flash memory and is loaded automatically at startup. The design requires about one megabit of the two megabits of Flash memory available. All 216K bits of the BRAM are loaded with graphical font information for displaying characters to the LCD.

The controller uses several inputs. The one bit GPS data stream is received in NMEA format at a rate of 4800 bits per second.   The ADC data input is an eight-bit number between zero and 255 that is valid when the one bit ADC done input is active. Four buttons are used to control the reset, next sample, sample, and zero functions of the controller. The 50 MHz clock on the Spartan 3 board is to control the synchronous design.

The controller outputs signals to several devices. The LCD receives 24 bits of RGB data and a composite sync signal at 12.5 MHz. Three bits of the ADC control output are used to sequence the ADC through successive read and write operations to take measurements from the sensor. This operation is sequenced to allow a guarantee of valid measurements from the ADC. In addition, sixteen bits of control information are used to manage the switches that sequence the routing of voltages on the sensor.

The controller implements four main modules to accomplish its function. A sensor module manages communication with the ADC and the switches. When requested, the sensor module sequences the switches, takes measurements from the ADC, and provides a calibrated sensor measurement to the requestor. A GPS module receives eight-bit ASCII data transmitted between one start and one stop bit. It samples the data stream at 4800 bits per second, decoding an updated position, date and time every two seconds. A video module creates the GUI interface and generates the appropriate LCD control signals for the RGB and composite-sync outputs. A top level module handles communication between each of the modules and the Spartan 3 board. Figure 5 illustrates the inputs and outputs of the controller, as well as each of the main modules controlling a major function. The complete code for the implemented controller is found in Appendix A.

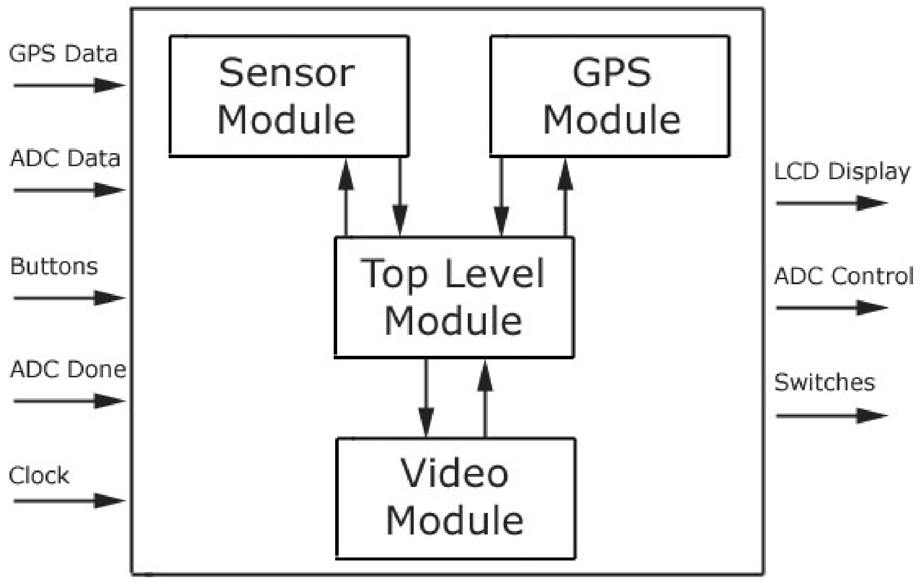

Figure 5 – Inputs, outputs, and main modules of the controller

Top Level Module

The top level module is used as a manager for the other main modules and also provides access to the resources connected to the Spartan 3 board. In addition to instantiating the main modules, the top level module provides the appropriate connections between system level and module level inputs and outputs. The top level module also creates a 25 MHz clock that drives each of the modules.

A constraints file is associated with the top level module to associate the names used for the ports in the top level module with the actual resources on the Spartan 3 board. In this manner, the top level module serves as an interface between the Spartan 3 resources and the instantiated modules.

Synchronize Module

The synchronize module is used by the top level module to synchronously handle the button inputs. Without synchronizing the button inputs, a button pressed near the rising edge of the system clock may result in unpredictable behavior. The clock input is the 25 MHz clock generated within the top level module. The button input is connected to the in port. The output of the module is a signal named out that is nominally zero and becomes one in a synchronous manner while a button is pressed. These inputs and outputs are shown in Figure 6.

Figure 6 – Inputs and outputs of the Synchronize Module

The synchronize module uses the clock signal to synchronize the input to a register. Every clock cycle, the register is set to the state of the button. Another register connected to out is updated with the value of the one clock cycle delayed input. Given this implementation, the synchronize module achieves its function as previously described.

Sensor Module

The sensor module generates control signals for the switches and ADC. The inputs to the module are the 25 MHz clock, the synchronized reset request from the reset button, and a start sensor signal. When a one is received on the start sensor input, the module simultaneously begins to generate the switching signals on sensor_sel and ADC control signals on cs, rd, and wr. When the intr input from the ADC becomes active, the ADC has placed an individual eight-bit measurement on the sensor_in input. These signals are sequenced appropriately to sample the sensor from the bottom to the top in an incremental manner. Seven measurements from the sensor_in input are combined to form the contents of a single 56-bit register. The sensor module repeats this process for the length of the sensor eight times and generates an average measurement for precision. The ssa_ready output will become one when an average measurement is ready, notifying the requestor that the average measurement is updated on the 56-bit sensor_display output. If a zero request is generated by the synchronized zero button, the sensor module will follow a similar process, generating an average of eight sets of measurements and signaling the success with a one on ssa_ready for a clock cycle. This will notify the requestor that the data on sensor_display is the updated zero reading. The sensor module uses the sensor clock, sensor control, sensor decoder, sensor communicator, and sensor configuration modules to achieve its function. Figure 7 illustrates the inputs and outputs of the sensor module.

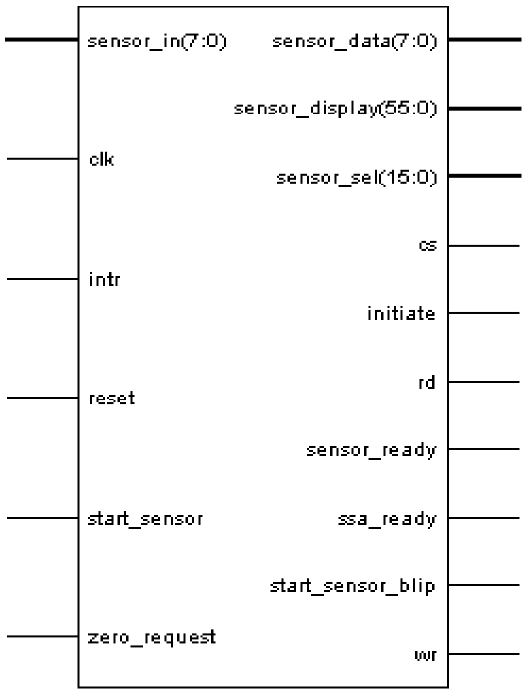

Figure 7 – Inputs and outputs of the Sensor Module

The sensor module converts a long button press of the synchronized start_sensor and zero buttons into a single clock cycle delayed pulse. This is accomplished using registers to detect the onset of a button press and the response to any change is delayed one clock cycle to guarantee synchronous operation.

Sensor Clock Module

The sensor clock module is used by the sensor module to properly time the sequencing of the switches and the ADC. The input clock is the 25 MHz clock. Nominally, the active input and the enable output are zero. When the active input is one, the enable outputs a one at a 50 KHz rate. When the reset input is one, the enable output is set to zero regardless of the state of the active input. The inputs and outputs of the sensor clock module are illustrated in Figure 8.

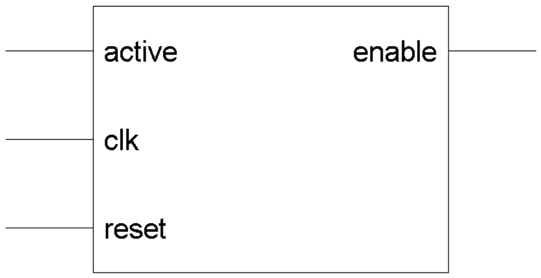

Figure 8 – Inputs and outputs of the Sensor Clock Module

The sensor clock module uses the clock input to increment a counter register from zero to 500. When the counter register reaches 500, the sensor clock module produces a one on the enable output for one clock cycle. On the positive edge of the clock, the module first checks for a reset button press, sensing a zero on the reset input. If a reset is sensed, the counter and the enable output are set to zero on the next clock cycle for the duration of the button press. Given this construction, the sensor clock module performs its function as described earlier.

Sensor Control Module

The sensor control module is used to properly control the ADC operation and obtain individual measurements from the sensor. The input clock is 25 MHz and the module accepts a reset input. When a one is sensed on the initiate input, the module proceeds to sequence the ADC through a sequence to take an individual measurement. The active output starts the sensor clock when active, generating a 50 KHz enable signal. The cs and wr outputs to the ADC are set active for the necessary durations, notifying the ADC to start converting its analog voltage input to a digitally scaled equivalent. After the conversion is completed, the intr input from the ADC signals the success. The cs and rd outputs to the ADC are set active for the necessary durations while the sensor_data output is set to the eight-bit sensor_in measurement from the ADC. The sensor_ready output will be set to one for one clock cycle, notifying the requestor that the single measurement on the sensor_data output is updated. Also, the ADC acknowledges the successful read of the measurement by resetting the intr input. If the reset input is enabled, the outputs are set to their nominal states and the internal sequencing is reset. The inputs and outputs of the sensor control module are shown in Figure 9.

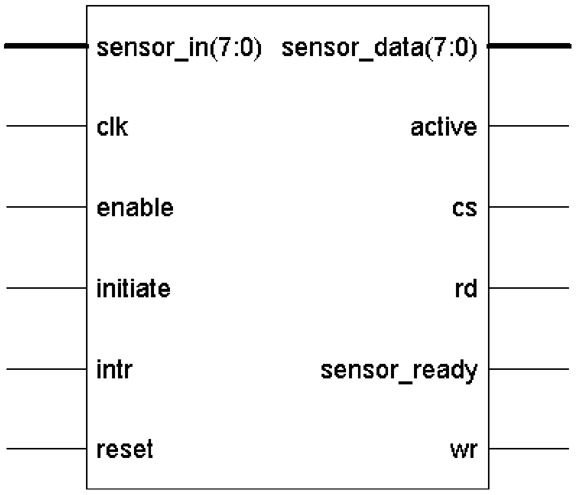

Figure 9 – Inputs and outputs of the Sensor Control Module

The sensor control module uses the clock signal to synchronously walk through a finite state machine and control sampling. In the idle state, the module waits for a request on the initiate input. Once sensed, the active output is set to one, generating a 50 KHz signal on enable. When the first signal is received on the enable input, the sensor control module activates the cs output and later the wr output. This requests a conversion to the ADC. The intr input is activated by the ADC, signaling that it is converting. When the intr input is sensed, the cs and wr outputs are reset to zero. As the ADC has completed its conversion, it resets intr. The module reactivates the 50 KHz enable signal by generating a one on active. When the first signal is received on the enable input, the sensor control module activates the cs output and later the rd output. This requests a read from the eight-bit data output of the ADC. The ADC will acknowledge the read request and reactivate intr. Once this signal is detected, the necessary delays are provided and the converted measurement on sensor_in is placed onto the eight-bit sensor_data. At the same time, a one is placed on sensor_ready for one clock cycle. If the reset button is pressed, all registers are reset to their initial states and the FSM is set to the idle state. Given this implementation, the sensor control module operates in the manner described earlier.

Figure 10 depicts the timing diagram for a conversion of zero volts to the eight-bit number zero. The measurement request can be seen on the initiate input. Next, the write, conversion, and read processes are seen. Finally, the one present on sensor_ready denotes the successful conversion and the output on sensor_data is valid during this time.

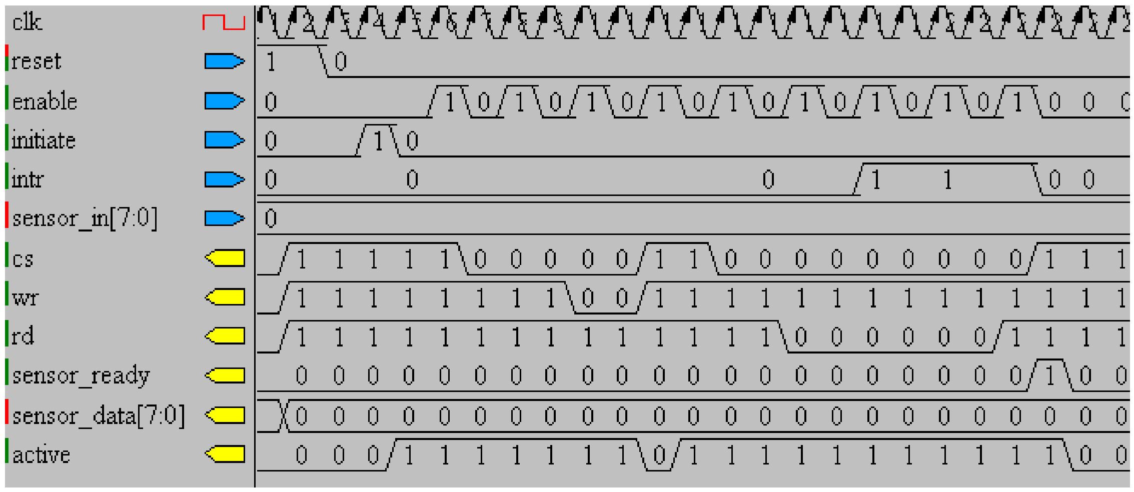

Figure 10 – Timing diagram for the Sensor Control Module

Sensor Decoder Module

The sensor decoder module is used by the sensor module to properly sequence the switches and to buildup a set of seven measurements from the ADC. The input clock is 25 MHz and the module accepts a reset input. When requested by the sensor communicator module on the start_sample input, the sensor decoder module selects the input to the switches on the sensor_sel output. The initiate output is enabled, taking a measurement from the ADC and enabling sensor_ready when a measurement is ready. This process is repeated for a total of seven measurements and packaged into a 56-bit output called sensor_stream_raw. When the sensor_stream_output is updated, the stream_ready output is one for a clock cycle. By this method, the sensor decoder requests individual samples in an incremental manner and outputs the seven measurements on sensor_stream_raw when completed. The inputs and outputs of the sensor decoder module are shown in Figure 11.

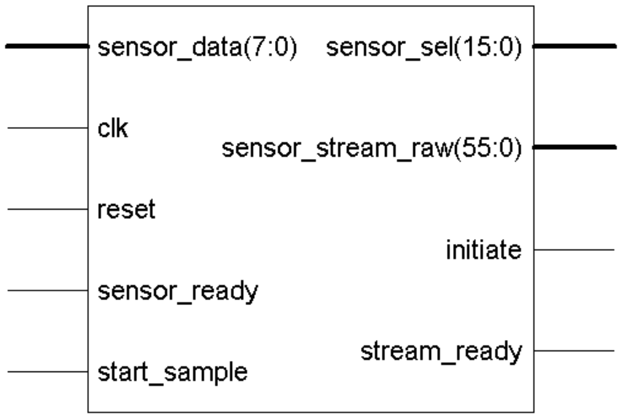

Figure 11 – Inputs and outputs of the Sensor Decoder Module

The sensor decoder module uses the clock signal to synchronously walk through a finite state machine and control the ADC. In the idle state, the module waits for a request on the start_sample input. Once received, the initiate output is set to one, beginning an ADC measurement. When the eight-bit measurement is ready on the sensor_data input, the sensor_ready input is active. The module saves the first byte of sensor data to a register. The module continues in a similar manner to request and save six additional measurements along the sensor's shaft and also generates the sensor_stream_raw output when completed. At the same time, the stream_ready output is set to one for a clock cycle, notifying the requestor that the measurements for the length of the sensor's shaft are ready. If the reset button is pressed, all registers are reset to their initial states and the FSM is set to the idle state. Given this implementation, the sensor decoder module achieves its function as previously described.

Sensor Communicator Module

The sensor communicator module provides more precise sensor measurements by averaging eight of the 56-bit measurements from the sensor decoder module. The input clock is 25 MHz and the module accepts a reset input. When requested by the sensor module on the start_sensor_blip input, the sensor communicator module responds by requesting a 56-bit measurement from the sensor decoder module. The start_sample output is enabled and a new measurement is received on the sensor_stream_raw input when the stream_ready input is active. This process is repeated for a total of seven measurements and packaged into a 56-bit output called sensor_stream_raw. When the sensor_stream_raw input is updated, the stream_ready output is one for a clock cycle. This process is repeated eight times and the average of the measurements is computed and output on the sensor_stream_averaged output. When the average of eight measurements has been computed, the ssa_ready output is activated, notifying the requestor that the average is updated. In this way, the sensor communicator module increases the precision of the measurements by averaging many samples. The inputs and outputs of the sensor communicator module are shown in Figure 12.

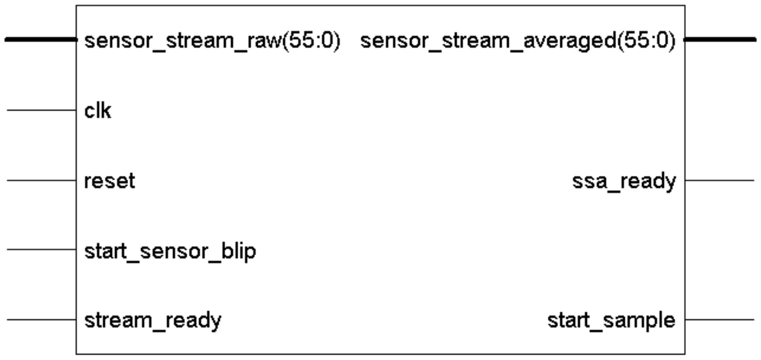

Figure 12 – Inputs and outputs of the Sensor Communicator Module

The sensor communicator module uses the clock signal to synchronously walk through a finite state machine and control the computation of the average measurement. In the idle state, the module waits for a request on the start_sensor_blip input that is controlled by the sample button. When this is received, the start_sample output is set to one, initiating the generation of seven measurements from the ADC. When the 56-bit measurement is ready on the sensor_stream_raw input, the stream_ready input is active. The module saves the first 56-bit measurement to a register. The module continues in a similar manner to request and save seven additional measurements. A sum is computed and the most significant 56 bits in the 59-bit result are taken as the average of eight measurements. The result is output onto sensor_stream_averaged and simultaneously the ssa_ready output is activated. This notifies the requestor that the average measurement has been updated. If the reset button is pressed, all registers are reset to their initial states and the FSM is set to the idle state. Given this implementation, the sensor communication module achieves its function as previously described.

Sensor Configuration Module

The sensor configuration module is used by the sensor module to allow the sensor to operate in different soil types. It stores a zero measurement to be subtracted from later measurements in order to calibrate the sensor to the soil type. The input clock is 25 MHz and the module accepts a reset input. When the ssa_ready input is active, the module checks whether the averaged measurement is a zero reading or simply a measurement. If the zero button has been pressed, it is a zero reading, and the module stores the input of sensor_stream_averaged and outputs it on the zero_offset output. If the zero button has not been pressed, then the measurement is assumed to be a sample request. In this case, when ssa_ready is active, the module subtracts the current zero measurement from the input on sensor_stream_averaged. The result is output onto sensor_display. The inputs and outputs of the sensor configuration module are illustrated in Figure 13.

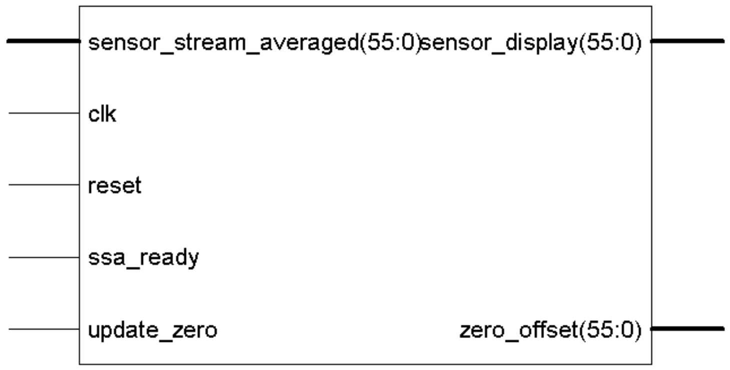

Figure 13 – Inputs and outputs of the Sensor Configuration Module

The sensor configuration module uses the clock signal to synchronously walk through a finite state machine and control the zero_offset and sensor_display outputs. The module monitors the synchronized update_zero button and when a button press is sensed, the module flags a register to indicate that a zero request has been made. In the idle state, the module waits for the ssa_ready input to activate. Once sensed, the module checks whether the zero register has sensed a zero request. If there has been a zero request, the zero_offset output is set to the value on the sensor_stream_averaged input. Otherwise, the sensor_display output is set to the value of the sensor_sensor_averaged input minus the stored zero measurement. If the reset button is pressed, all registers are reset to their initial states and the FSM is set to the idle state. Given this implementation, the sensor configuration module achieves its function as described earlier.

GPS Module

The gps module receives and interprets the gps data signal, updating the date, time, and position upon request. The inputs to the module are the 25 MHz clock, reset, and the data signal. The module samples the continuous stream from the data input at 4800 bits per second. The NMEA data format exports a start bit, followed by an eight-bit ASCII value. The transmission of each byte of data is succeeded by a stop bit and the transmission repeats in a similar fashion until all data is transmitted. The NMEA data format specifies the order in which information will be transmitted and each field of transmitted data is separated by commas. The gps module recognizes a $ symbol as the start byte and once detected counts the number of commas transmitted to determine which information is being sent. As the gps module encounters the date, time, and position periods of the transmission, the x, y, and fix outputs are updated with their current values. When the gps module has updated all of the information of interest, it sets the data_ready output to one for a clock cycle. This notifies the requestor that the gps module has successfully completed its update. The gps module uses the gps clock, gps decoder, and gps communicator modules to perform its functions. Figure 14 depicts the inputs and outputs of the gps module.

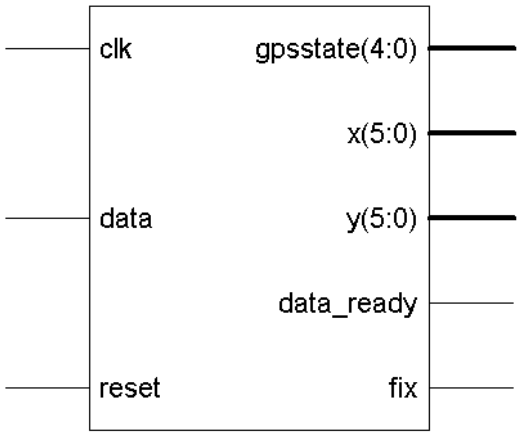

Figure 14 – Inputs and outputs of the GPS Module

GPS Clock Module

The gps clock module is used by the gps module to sequence the sampling of the gps data signal. The input clock is the 25 MHz clock. Nominally, the counting input and the enable output are zero. When the counting input is one, the enable signal outputs a one at a 4.8 KHz rate. When the reset input is one, the enable output is set to zero regardless of the state of the counting input. The inputs and outputs of the gps clock module are illustrated in Figure 15.

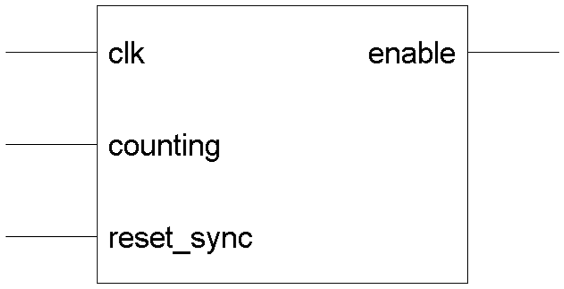

Figure 15 – Inputs and outputs of the GPS Clock Module

The gps clock module uses the clock input to increment a counter register from zero to 5207. In this way, the 25 MHz clock is converted to a 4800 bit per second enable signal. The counter is reset to zero when it reaches 5207. When the counter is exactly in the middle of its count, the enable output is set active for one clock cycle. On the positive edge of the clock, the module first checks for a reset button press, sensing a zero on the reset input. If a reset is sensed, the counter and the enable output are set to zero on the next clock cycle. Given this implementation, the gps clock module performs as previously described.

GPS Decoder Module

The gps decoder module is used by the gps module to properly sequence the sampling of the data stream from the gps. The input clock is 25 MHz and the module accepts a reset input. The gps decoder module waits until it recognizes the initial start bit on the data input. Once recognized, the module sequences through eight more bits of ASCII data and a single stop bit. Once the stop bit is received, the eight-bit ASCII data is sent to the data_out output and the data_ready output is set to one for one clock cycle. This notifies the requestor that each individual byte of ASCII data has been decoded and presented on the data_out output. The inputs and outputs of the gps decoder module are shown in Figure 16.

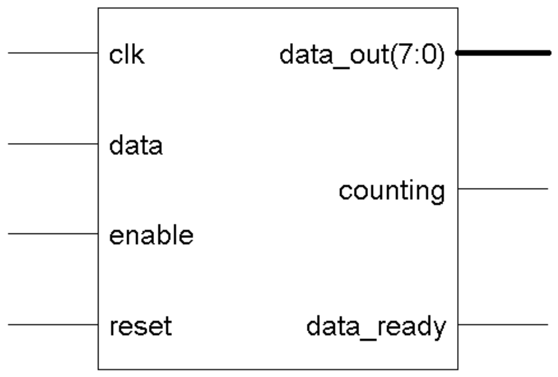

Figure 16 – Inputs and outputs of the GPS Decoder Module

The gps decoder module uses the clock signal to synchronously walk through a finite state machine and sample the GPS data. In the idle state, the module waits for a period of eight clock cycles of idle information before it can guarantee that the GPS is not sending data. The module waits for the first start bit, at which point the counting output is set active, starting the 4800 bit per second enable signal. As each enable signal is received, the FSM builds up an eight-bit ASCII byte and recognizes a stop bit. The module continues in a similar manner to sample the data stream until the GPS finishes sending its update. When each ASCII byte is received and compiled, it is set on the data_out output and the data_ready output is set to one for a single clock cycle. The GPS sends an updated data stream every two seconds. If the reset button is pressed, all registers are reset to their initial states and the FSM is set to the idle state. Given this implementation, the gps decoder module achieves its function as previously described.

Figure 17 depicts the timing diagram for the transmission of the letter p. The module recognizes the idle state of the data stream in the initial part of the diagram. When the start bit is recognized, the counting output resets and the eight-bit ASCII data is read on the subsequent eight enable pulses. The least significant bit is transmitted first. When the stop bit is received on the data line, the counting output is reset and the data_out output is updated. Simultaneously, the data_ready output is set to one for a clock cycle. The data_out value of 80 (00001010) represents the ASCII letter p.

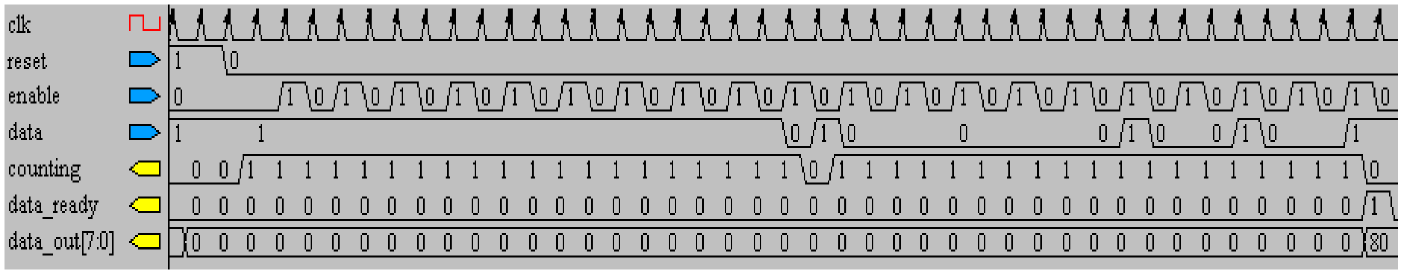

Figure 17 – Timing diagram for the GPS Decoder Module

GPS Communicator Module

The gps communicator module is used by the gps module to update the time, date, and position information. The input clock is 25 MHz and the module accepts a reset input. The gps communicator module reads the eight-bit ASCII data and recognizes commas to separate the preset data fields. The module waits to receive the header of the sentence containing the data of interest. When the sixth comma has been received, the time has been updated. In a similar manner, all of the remaining data is updated as the appropriate comma is encountered. In this way, the gps communication module updates all position, date, and time information every two seconds. The inputs and outputs of the gps communicator module are shown in Figure 18.

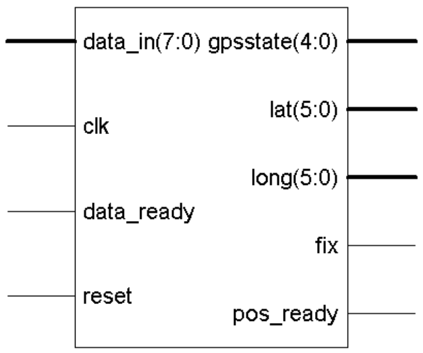

Figure 18 – Inputs and outputs of the GPS Communicator Module

The gps communicator module uses the clock signal to synchronously walk through a finite state machine and sample the gps data stream. In the idle state, the module waits for the first byte of the header, the $ symbol. When this is received, the module continues to look for the appropriate subsequent header characters. Once the header has been identified, the module knows that the data after sixth comma received is the updated time. Similarly the data after the ninth comma are the updated minutes of latitude. The module continues in a similar manner to update the position, date, and time. When the end of the sentence is reached, the FSM returns to the idle state, waiting for the next update two seconds later. If the reset button is pressed, all registers are reset to their initial states and the FSM is set to the idle state. Given this implementation, the gps communicator module achieves its function as earlier described.

Video Module

The video module is responsible for generating the display control signals sent to the LCD. The video module receives data inputs from other main modules, interprets them, and produces the appropriate RGB and HSYNC signals. The inputs to the module are the

25 MHz clock, reset, the position and chronological data from the GPS, and the zero and averaged measurements from the sensor module. The module displays the desired information by incrementally generating the control signals for horizontal lines, pixel by pixel. When the bottom of the screen is reached, the video module updates the data to be displayed and begins to draw the next frame in a similar manner. The module displays the GUI interface on the LCD as described earlier. The GUI is achieved by combining characters and rectangles of different colors.

VGA Module

The VGA module is used by the video module to properly sequence the synchronization signals for the LCD. The input clock is 25 MHz. The module steps the input clock down to 12.5 MHz, which is used to drive the LCD. The module begins at the top left pixel and proceeds to move one pixel to the right every clock cycle. When the edge of the screen is met, the next line is written at the left edge in a similar manner after waiting for the proper delay. After the bottom line has been written, the module waits for the proper delay and then begins back at the top position. At this time it also activates the vreset output and the appropriate vsync output. As the module proceeds through its display to the screen, it places its position on hcount and vcount. When either the right side or bottom of the screen is encountered, hsync or vsync respectively is activated. The inputs and outputs of the VGA module are shown in Figure 19.

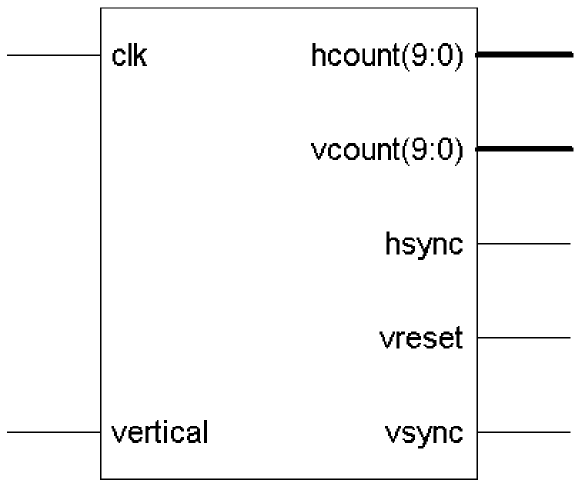

Figure 19 – Inputs and outputs of the VGA Module

The VGA module uses the clock input and pre-defined timing parameters to control the LCD display. Values are defined for the front porch, active, and back porch syncing parameters. A register is used to turn the clock input of 25 MHz into a 12.5 MHz signal. Every time the 12.5 MHz signal is generated, the horizontal position is incremented by one to the right. When the screen width is reached, the module waits for the front porch duration. An active signal is then generated on the hsync output for the duration of the active period. The module then waits for the back porch duration and the vertical count is incremented by one. The horizontal position is reset to zero and the module continues to fill the screen in an interlaced manner, writing all the even lines first and then the odd lines. When the bottom line has been written, the module produces the appropriate signal on vsync using the timing parameters defined for its front porch, active, and back porch. Afterward, the vcount output is reset to zero and the module continues to write another frame. Given this implementation, the VGA module achieves its function as previously described.

Character Display Module

The character display module is used by the video module to display ASCII strings to the screen. The input clock is 25 MHz. The module accepts a location for the top left corner on the cx and cy inputs. In addition, the string to be displayed is placed on cstring and the hcount and vcount are also supplied as inputs. The module uses a BRAM to access a font ROM that converts ASCII inputs to pixel outputs. Text and background colors can be defined by parameters. As the hcount and vcount proceed through their sequencing, the module turns on individual pixels to match the input request. The inputs and outputs of the character display module are shown in Figure 20.

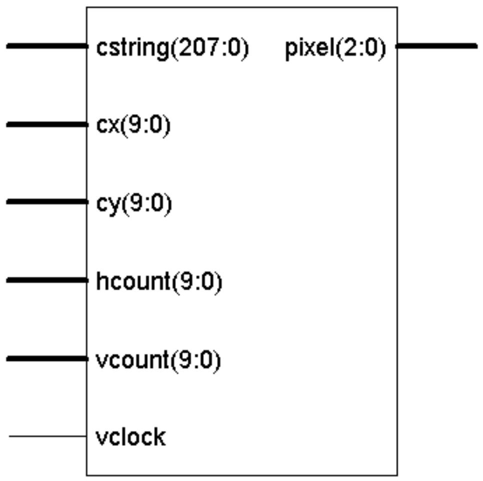

Figure 20 – Inputs and outputs of the Character Display Module

Rectangle Generation Module

The rectangle generation module is used by the video module to display the GUI. The module accepts the hcount and vcount inputs as well as the current measurement on the 56-bit sensor_display input. The module paints the teal and white GUI background to the LCD. The module also breaks up a 56-bit measurement into the seven individual measurements and displays the result for the shallowest reading on the left bar of the graph. Subsequent bars represent the measurement for the next deepest reading. As the hcount and vcount proceed through their sequencing, the module turns on individual pixels to draw the GUI on the screen. The inputs and outputs of the rectangle generation module are shown in Figure 21.

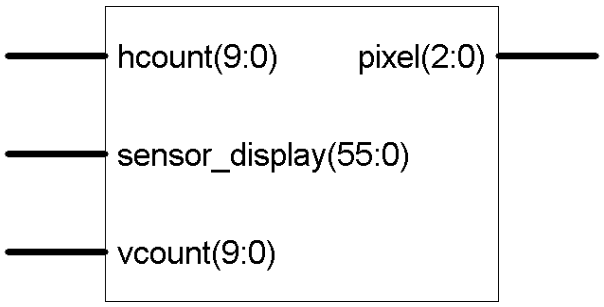

Figure 21 – Inputs and outputs of the Rectangle Generation Module

The rectangle generation module uses the hcount and vcount inputs to decide what color to output for each pixel on the screen. The module checks whether the current pixel should be colored teal, corresponding to the lines breaking up the GUI display. If the hcount and vcount are within the bounds listed for the teal color, the pixel output is set to teal and the active pixel is painted that color. When the pixel should not be painted teal, the module checks whether the current pixel should be colored blue, corresponding to the bar graph of the active measurement. If so, the module breaks the 56-bit sensor_display input into eight-bit measurement and scales seven rectangles across the bottom with heights equal to the magnitude of each measurement. When the pixel should not be painted blue or teal, the module checks whether the current pixel should be colored white. If text or other rectangles have not filled the active pixel, then the pixel is colored white. The pixel output is set to black when syncing to avoid unwanted pixel activations while the LCD is retracing during syncing. Given this implementation, the rectangle generation module performs as described earlier.

Testing and Debugging

Each main module was tested and debugged separately before being compiled in the top level module. As each individual module was completed, each was individually simulated and tested using a test bench to ensure that all the likely inputs to the module produced the desired outputs. Errors that produced warnings were corrected line-by-line. A test bench was created to simulate the total functionality of the controller. This simulation helped to identify modules that needed coding corrections and allowed for the controller to be tested before it was programmed to the FPGA. After programming, the controller was tested for function and corrections were made to the modules as necessary.

The GPS data signal was first interpreted using a connection to the serial port of a PC. The hyper terminal program was used to display the transmitted ASCII data to verify the output of the GPS. Once the sentence structure of the GPS data was known, development of the gps module proceeded. The module was tested to successfully update the position and chronological information every two seconds. This was done by visually confirming the correspondence of the gps communicator state and LEDs on the Spartan 3 board. In addition, the updated information was verified to correctly update in the appropriate locations on the LCD display. The tests confirmed that the gps module was functioning as proposed.

The controlling circuitry for the sensor underwent several iterations before the final design was selected. Originally op amps were used, however, the use of switches and an ADC were found to give more control of routing voltages to the desired pads on the sensor. In addition the use of the ADC allows for high-speed and high-resolution measurements that permit a more powerful and precise calibration of the sensor. Once the circuitry was selected, the sensor module was developed. A logic analyzer was used to confirm that the ADC was being sequenced properly and that measurements were actually corresponding to the actual analog voltage inputs. In addition, the module was confirmed to properly sequence through the switches. The zero function of the module was tested by confirming that the zero request successfully resulted in an updated zero measurement. The sensor was tested to properly detect the location and relative amount of water across each different pad pair by placing a moist cloth across the sensor. Areas that were not exposed to the cloth correctly displayed a zero the on the bar graph, while areas that were exposed to dampness registered a measurement judged to be near the saturation level. This testing confirmed the successful function of the sensor module as proposed.

The datasheet for the LCD was not available to assist in the development of the video module. The control signals for the LCD were discovered using a logic analyzer. It was also discovered that the LCD uses interlacing to display its picture. Because the exact clock frequency of the LCD was unattainable on the Spartan 3 board, the timing observed on the logic analyzer could not be used for the video module. Instead, a brute force method was used to obtain proper timing signals for a frequency that the Spartan 3 board could easily produce. Once obtained, the timing was fine tuned on a PC using the software PowerStrip. The video module was developed once the exact LCD timings were found. The video module was tested to successfully control the LCD and to respond appropriately to updates in the data to be displayed.

Conclusions

The main objective of the final project was to design and implement the soil-water characterization unit. There has been an extensive growth in knowledge of the Spartan 3 board and its resources. Also, a there is a better understanding of how the Xilinx architecture makes changing the FPGA basis of the design relatively simple. The soil-water characterization unit was successfully designed, implemented, tested, and documented, verifying that the objectives of this project as proposed have been met and exceeded.

Each main module of the controller design touches on a concept that is used throughout digital design. One largely powerful method is that of breaking up complex controllers into simpler modules and instantiating those modules in a top-level module at a later time. This method proved not only useful in designing the controller, but also in troubleshooting and testing specific parts of the design that required attention. Another powerful method was that of serial communication. Using a standard protocol on a serial standard was shown to be an effective means of sending small amounts of ASCII data. In addition, the standard allowed for a relatively simple connection of the GPS data stream to a PC. The sensor module showed how an ADC could be used to make a large number of measurements in a amount of short time. This allows for rapid updates of readings, no matter what voltage source the ADC is converting. Finally, the font ROM used in the character display module showed how storing some preset values for later use can produce powerful results. Any data could be stored to a ROM, including pictures, instructions, or any other desired information. These methods of digital design will prove useful in future applications.

Appendix A

TOP LEVEL MODULE

```
module starter_board(
     // clocks
     clock_50MHz, clock_socket,

     // 256k x 32 SRAM
     sram_a,sram_ce_b,sram_we_b,
     sram_io1,sram_ce1,sram_ub1,sram_lb1,
     sram_io2,sram_ce2,sram_ub2,sram_lb2,

     // I/O
     segment,an,sw,btn,led,
     vga_r,vga_g,vga_b,vga_hs,vga_vs,
     ps2_d,ps2_c,
     rs232_rxd,rs232_txd,rs232_rxd_a,rs232_txd_a,
     flash_d0,flash_oe,flash_rclk,

     // expansion connectors
     A1_4,A1_21,A1_22,
     A2_io,A2_db,A2_astb,A2_dstb,A2_write,A2_wait,A2_reset,

B1_db,B1_adr,B1_we,B1_oe,B1_cs,B1_astb,B1_dstb,B1_write,B1_wait,B1_rese
t,B1_int
     );

   input clock_50MHz;
   input clock_socket;

   output [17:0] sram_a;
   output       sram_ce_b;
   output       sram_we_b;
   inout [15:0]  sram_io1;
   output       sram_ce1;
   output       sram_ub1;
   output       sram_lb1;
   inout [15:0]  sram_io2;
   output       sram_ce2;
   output       sram_ub2;
   output       sram_lb2;

   output [7:0]  segment;   // active low
   output [3:0]  an;        // active low
   input [7:0]  sw;
   input [3:0]  btn;
   output [7:0]  led;

   output       vga_r;
   output       vga_g;
   output       vga_b;
   output       vga_hs;
   output       vga_vs;
```

```
inout        ps2_d;
input        ps2_c;

input        rs232_rxd;
output       rs232_txd;
input        rs232_rxd_a;
output       rs232_txd_a;

input        flash_d0;
output       flash_oe;
output       flash_rclk;

inout        A1_4,A1_21,A1_22;

inout [18:1]  A2_io;
inout [7:0]  A2_db;
inout        A2_astb;
inout        A2_dstb;
inout        A2_write;
inout        A2_wait;
inout        A2_reset;

inout [7:0]  B1_db;
inout [5:0]  B1_adr;
inout        B1_we;
inout        B1_oe;
inout        B1_cs;
inout        B1_astb;
inout        B1_dstb;
inout        B1_write;
inout        B1_wait;
inout        B1_reset;
inout        B1_int;

// SRAM pins
assign sram_a = 18'h0;
assign sram_ce_b = 1'b1;
assign sram_we_b = 1'b1;
assign sram_io1 = 18'h0;
assign sram_ce1 = 1'b0;
assign sram_ub1 = 1'b0;
assign sram_lb1 = 1'b0;
assign sram_io2 = 1'b0;
assign sram_ce2 = 1'b0;
assign sram_ub2 = 1'b0;
assign sram_lb2 = 1'b0;

// misc. I/O
assign segment = 8'hFF;   // active low
assign an = 4'd0;         // active low
assign ps2_d = 1'bz;
assign rs232_txd = 1'b0;
assign rs232_txd_a = 1'b0;
assign flash_oe = 1'b1;
assign flash_rclk = 1'b0;

// expansion connectors
```

```
assign A1_4 = 1'b0;
assign A1_21 = 1'b0;
assign A1_22 = 1'b0;
assign A2_io[18:17] = 2'd0;
assign A2_astb = 1'b0;
assign A2_dstb = 1'b0;
assign A2_write = 1'b0;
assign A2_wait = 1'b0;
assign A2_reset = 1'b0;
assign B1_db[0] = 1'b0;
assign B1_db[7:5] = 3'd0;
assign B1_adr[5] = 1'b0;
assign B1_we = 1'b0;
assign B1_oe = 1'b0;
assign B1_cs = 1'b0;
assign B1_astb = 1'b0;
assign B1_dstb = 1'b0;
assign B1_write = 1'b0;
assign B1_wait = 1'b0;
assign B1_reset = 1'b0;
assign B1_int = 1'b0;

// Use FPGA's digital clock manager to produce a
// 25MHz clock
wire vidclock_unbuf,vidclock;
DCM vclk1(.CLKIN(clock_50MHz),.CLKFX(vidclock_unbuf));
// synthesis attribute DUTY_CYCLE_CORRECTION of vclk1 is "TRUE"
// synthesis attribute CLKFX_DIVIDE of vclk1 is 4
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 2
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 20
BUFG vclk2(.O(vidclock),.I(vidclock_unbuf));

// Reset Generation
wire power_on_reset;
SRL16 reset_sr (.D(1'b0), .CLK(vidclock), .Q(power_on_reset),
                .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

wire reset_button;
synchronize b3(vidclock, btn[3], reset_button);

wire reset = power_on_reset | reset_button;
assign led[7] = reset;

// 640x480 VGA display
wire [9:0] hcount, vcount;
wire vga_hsync, vga_vsync;
wire vreset;

// 8 Switches
wire [1:0] switch;
assign switch[0] = sw[0];

VGA_module vga1(vidclock,vga_hsync,vga_vsync,hcount,vcount,switch[0],vreset);

// Rectangle Generation - The rectangle module defines the
```

```
   rectangles to be drawn on screen
wire [2:0] pixel;
wire [55:0] sensor_display;
Rectangle_generation rect_gen(sensor_display,hcount,vcount,pixel);

// Button 1 is start_sensor
wire start_sensor;
synchronize b1(vidclock, btn[1], start_sensor);

// Button 0 is zero_request
wire zero_request;
synchronize b0(vidclock, btn[0], zero_request);

// Instantiate GPS Module and I/O connections
     wire [5:0] x,y;
     wire fix;
     wire data;
     wire [4:0] gpsstate;

     assign data = (reset) ? 1 : ~B1_adr[4];
     assign led[0] = ~data;

     GPSmodule gpsmod(vidclock, reset, data, x, y, data_ready,


     assign led[1] = (gpsstate == 1) ? 1 : 0;
     assign led[2] = (gpsstate == 2) ? 1 : 0;
     assign led[3] = (gpsstate == 8) ? 1 : 0;
     assign led[4] = (gpsstate == 16) ? 1 : 0;

     // Instantiate Sensor Module
     wire cs;
     wire wr;
     wire rd;
     wire intr;
     wire initiate;
     wire [7:0] sensor_in;
     wire [7:0] sensor_data;
     wire [15:0] sensor_sel;

     assign intr = (reset) ? 1 : B1_db[4];
     assign sensor_in = A2_db[7:0];

     assign led[5] = (zero_request) ? 1 : 0;
     assign led[6] = (start_sensor) ? 1 : 0;

     Sensormodule sensormod(vidclock, reset, zero_request,
          start_sensor, sensor_in, intr, cs, wr, rd, sensor_sel,
          sensor_data, initiate, sensor_display, ssa_ready);

     assign A2_io[16:1] = (reset) ? 16'd0 : sensor_sel[15:0];
     assign B1_db[3] = (reset) ? 1 : wr;
     assign B1_db[2] = (reset) ? 1 : rd;
     assign B1_db[1] = (reset) ? 1 : cs;

// character display module: Title
wire [199:0] cstring = "MussiLabs Soil-Water Unit";
wire [2:0]  cdpixel;
char_string_display cd(vidclock,hcount,vcount,
                      cdpixel,cstring,10'd112,10'd2);
defparam cd.NCHAR = 25;
defparam cd.NCHAR_BITS = 5;
defparam cd.color = 0;
defparam cd.bg_color = 6;

// character display module: Location Info
wire [63:0] cstring0 = "Location";
wire [2:0]  cdpixel0;
char_string_display cd0(vidclock,hcount,vcount,
                      cdpixel0,cstring0,10'd0,10'd28);
defparam cd0.NCHAR = 8;
defparam cd0.NCHAR_BITS = 3;
defparam cd0.color = 0;
defparam cd0.bg_color = 6;

wire [95:0] cstring00 = "            ";
wire [2:0]  cdpixel00;
char_string_display cd00(vidclock,hcount,vcount,
                      cdpixel00,cstring00,10'd128,10'd28);
defparam cd00.NCHAR = 12;
defparam cd00.NCHAR_BITS = 4;
defparam cd00.color = 7;
defparam cd00.bg_color = 7;

// Fixed Info
wire [255:0] cstring01 = "Date:               ";
wire [2:0]  cdpixel01;
char_string_display cd01(vidclock,hcount,vcount,
                      cdpixel01,cstring01,10'd0,10'd64);
defparam cd01.NCHAR = 20;
defparam cd01.NCHAR_BITS = 5;
defparam cd01.color = 0;
defparam cd01.bg_color = 7;

wire [255:0] cstring02 = "Time:               ";
wire [2:0]  cdpixel02;
char_string_display cd02(vidclock,hcount,vcount,
                      cdpixel02,cstring02,10'd0,10'd94);
defparam cd02.NCHAR = 20;
defparam cd02.NCHAR_BITS = 5;
defparam cd02.color = 0;
defparam cd02.bg_color = 7;

wire [255:0] cstring03 = "Latitude:           ";
wire [2:0]  cdpixel03;
char_string_display cd03(vidclock,hcount,vcount,
                      cdpixel03,cstring03,10'd0,10'd124);
defparam cd03.NCHAR = 20;
defparam cd03.NCHAR_BITS = 5;
defparam cd03.color = 0;
defparam cd03.bg_color = 7;

wire [255:0] cstring04 = "Longitude:          ";
wire [2:0]  cdpixel04;
```

```
char_string_display cd04(vidclock,hcount,vcount,
                      cdpixel04,cstring04,10'd0,10'd152);
defparam cd04.NCHAR = 20;
defparam cd04.NCHAR_BITS = 5;
defparam cd04.color = 0;
defparam cd04.bg_color = 7;

// character display module: Settings Info
wire [63:0] cstring1 = "Settings";
wire [2:0]  cdpixel1;
char_string_display cd1(vidclock,hcount,vcount,
                      cdpixel1,cstring1,10'd325,10'd28);
defparam cd1.NCHAR = 8;
defparam cd1.NCHAR_BITS = 3;
defparam cd1.color = 0;
defparam cd1.bg_color = 6;

wire [95:0] cstring10 = "            ";
wire [2:0]  cdpixel10;
char_string_display cd10(vidclock,hcount,vcount,
                      cdpixel10,cstring10,10'd453,10'd28);
defparam cd10.NCHAR = 12;
defparam cd10.NCHAR_BITS = 4;
defparam cd10.color = 7;
defparam cd10.bg_color = 7;

wire [79:0] cstring111 = (gpsstate <= 1) ? " >>       " : (gpsstate
     <= 2) ? " >>>>     " : (gpsstate <= 18) ? " >>>>>>   " :
     (gpsstate <= 16) ? " >>>>>>>>>" : "          "       ;
wire [2:0]  cdpixel111;
char_string_display cd111(vidclock,hcount,vcount,
                      cdpixel111,cstring111,10'd452,10'd64);
defparam cd111.NCHAR = 10;
defparam cd111.NCHAR_BITS = 4;
defparam cd111.color = 1;
defparam cd111.bg_color = 7;

// Fixed Info
wire [63:0] cstring11 = "GPS Com:";
wire [2:0]  cdpixel11;
char_string_display cd11(vidclock,hcount,vcount,
                      cdpixel11,cstring11,10'd325,10'd64);
defparam cd11.NCHAR = 8;
defparam cd11.NCHAR_BITS = 3;
defparam cd11.color = 0;
defparam cd11.bg_color = 7;

wire [143:0] cstring12 = "Zero:    Insert 0 ";
wire [2:0]  cdpixel12;
char_string_display cd12(vidclock,hcount,vcount,
                      cdpixel12,cstring12,10'd325,10'd94);
defparam cd12.NCHAR = 18;
defparam cd12.NCHAR_BITS = 5;
defparam cd12.color = 0;
defparam cd12.bg_color = 7;

// Button 2 is start_sensor
```

```
   wire next_sample;
   synchronize b2(vidclock, btn[2], next_sample);

        reg [2:0] sample_num = 0;
        reg old_next_sample = 0;
        reg start_next_sample = 0;
        wire [7:0] sample_num_ascii = sample_num + 49;
        always @ (posedge vidclock)
                begin
                        old_next_sample <= (reset) ? 1 : (next_sample) ?
                                                                0 : 1;
                        start_next_sample <= (reset) ? 0 : (next_sample
                                  == 1 && old_next_sample == 1) ? 1 : 0;
                        sample_num <= (reset) ? 0 : (start_next_sample) ?
                                            sample_num + 1 : sample_num;
                end

        wire [143:0] cstring13 = {"Sample:  ", sample_num_ascii,
                                                 "        "};
   wire [2:0]  cdpixel13;
   char_string_display cd13(vidclock,hcount,vcount,
                         cdpixel13,cstring13,10'd325,10'd138);
   defparam cd13.NCHAR = 18;
   defparam cd13.NCHAR_BITS = 5;
   defparam cd13.color = 0;
   defparam cd13.bg_color = 7;

   // character display module: Results Info
   wire [55:0] cstring2 = "Results";
   wire [2:0]  cdpixel2;
   char_string_display cd2(vidclock,hcount,vcount,
                         cdpixel2,cstring2,10'd0,10'd196);
   defparam cd2.NCHAR = 7;
   defparam cd2.NCHAR_BITS = 3;
   defparam cd2.color = 0;
   defparam cd2.bg_color = 6;

//On the 25 MHz clock, generate the required picture
   reg h_sync_old, v_sync_old;
   reg [2:0] rgb_old;

   always @(posedge vidclock)
                begin
                                h_sync_old <= vga_hsync;
                                v_sync_old <= vga_vsync;
                                rgb_old <= pixel | cdpixel | cdpixel0 |
                               cdpixel00 | cdpixel01 | cdpixel02 |
                                cdpixel03 | cdpixel04 |
                                cdpixel1 | cdpixel10 | cdpixel111 |
                                cdpixel11 | cdpixel12 | cdpixel13
                               cdpixel2;
                end

   assign switch[1] = sw[1];

   assign     B1_adr[3] = (h_sync_old);
   assign     B1_adr[0] = {8{rgb_old[0]}};
   assign      B1_adr[1] =  {8{rgb_old[1]}};
   assign      B1_adr[2] = {8{rgb_old[2]}};

endmodule
```

SYNCHRONIZE MODULE

```
module synchronize(clk,in,out);
  input clk, in;

  output out;

  reg r, out;

  always @ (posedge clk)
        begin
        r <= in;
        out <= r;
        end
endmodule
```

SENSOR MODULE

```
module Sensormodule(clk, reset, zero_request, start_sensor, sensor_in,
                   intr, cs, wr, rd, sensor_sel, sensor_data, initiate,
                   sensor_display, ssa_ready);

input clk, reset, start_sensor, zero_request;
input [7:0] sensor_in;
input intr;

output cs, wr, rd, initiate, ssa_ready;
output [7:0] sensor_data;
output [15:0] sensor_sel;
output [55:0] sensor_display;

wire enable;
wire sensor_ready;
wire [7:0] sensor_data;
wire active;
wire initiate;
wire intr;
wire cs;
wire wr;
wire rd;
wire stream_ready;
wire start_sample;
wire ssa_ready;
wire [15:0] sensor_sel;
wire [55:0] sensor_stream_raw, sensor_stream_averaged, zero_offset,
                                                    sensor_display;

reg old, old2, start_sensor_blip, update_zero;

Sensor_clock sensor_clk(clk, reset, active, enable);

Sensor_control sensor_ctl(clk, reset, enable, initiate, intr, cs, wr,
                   Rd, sensor_in, sensor_ready, sensor_data, active);

Sensor_decoder sensor_dec(clk, reset, start_sample, sensor_ready,
                  sensor_data, initiate, stream_ready, sensor_sel,
                                               sensor_stream_raw);

Sensor_communicator sensor_com(clk, reset, start_sensor_blip,
                stream_ready, sensor_stream_raw, start_sample,
                           sensor_stream_averaged, ssa_ready);


Sensor_config sensor_conf(clk, reset, update_zero,
                 sensor_stream_averaged, ssa_ready, zero_offset,
                                                sensor_display);

always @ (posedge clk)
        begin
                old <= (reset) ? 1 : (start_sensor) ? 0 : 1;
                start_sensor_blip <= (reset) ? 0 : (start_sensor == 1 &&
                                 old == 1) ? 1 : (update_zero) ? 1 : 0;

                old2 <= (reset) ? 1 : (zero_request) ? 0 : 1;
                update_zero <= (reset) ? 0 : (reset) ? 0 : (zero_request
                             == 1 && old2 == 1) ? 1 : 0;
          end

endmodule
```

SENSOR CLOCK MODULE

```
module Sensor_clock(clk, reset, active, enable);

input clk, reset, active;

output enable;

// Convert 25MHz clock to a 50KHz clock
parameter clocks_per_sample = 500;

reg enable;
reg [12:0] counter;

always @ (posedge clk)
        begin
        if (reset)
                begin
                enable <= 1'b0;
                counter <=13'd0;
                end

        if (active)
                begin

                if (counter == ((clocks_per_sample / 2) - 1))
                        begin
                        enable <= 1'b1;
                        counter <= counter + 1;
                        end

                else if (counter == (clocks_per_sample - 1))
                        counter <= 13'd0;

                else
                        begin
                        enable <= 1'b0;
                        counter <= counter + 1;
                        end
                end

        else
                begin
                enable <= 1'b0;
                counter <=13'd0;
                end

        end

endmodule
```

SENSOR CONTROL MODULE

```
module Sensor_control(clk, reset, enable, initiate, intr, cs, wr, rd,
                      sensor_in, sensor_ready, sensor_data, active);

input clk, reset, enable, intr, initiate;
input [7:0] sensor_in;

output active, sensor_ready, cs, wr, rd;
output [7:0] sensor_data;

reg counter, active, sensor_ready, cs, wr, rd, readolddata;
reg [3:0] state;
reg [7:0] sensor_data, keycode;

always @ (posedge clk)
        begin
        if (reset)
                begin
                state <= 0;
                active <= 0;
                counter <= 0;
                sensor_ready <= 0;
                sensor_data <= 0;
                cs <= 1;
                wr <= 1;
                rd <= 1;
                end

        else
                begin

                case (state)
                // Idle State
                0:      begin
                        active <= 0;
                        counter <= 0;
                        cs <= 1;
                        wr <= 1;
                        rd <= 1;
                        sensor_data <= 0;
                        sensor_ready <= 0;
                        if (initiate)
                                begin
                                        state <= 1;
                                        readolddata <= (intr) ? 1 : 0;
                                        active <= 1;
                                end
                        end
                // Wait For Address Setup
                        1:      begin
                                        if (enable == 1)
                                                begin
                                                      state <= state + 1;
                                                        cs <= 0;
                                                        counter <= 1;
                                                end
                                end
                        // Write Request
                        2:      begin
                                        counter <= (enable == 1) ?
                                            counter - 1 : counter;
                                        if (counter == 0)
                                                begin
                                                        state <= state + 1;
                                                        counter <= 1;
                                                        wr <= 0;
                                                end
                                end
                        // Wait for intr
                        3:      begin
                                        counter <= (enable == 1) ?
                                            counter - 1 : counter;
                                        if (counter == 0)
                                                begin
                                                        state <= state + 1;
                                                        counter <= 1;
                                                        cs <= 1;
                                                        wr <= 1;
                                                        active <= 0;
                                                end
                                end
                        // Conversion
                        4:   begin
                                        if (intr == 0)
                                                begin
                                                        active <= 1;
                                                        counter <= (enable
                                                         == 1) ? counter –
                                                         1 : counter;
                                                        if (counter == 0)
                                                                begin
                                                      state <= state + 1;
                                                      counter <= 1;
                                                      cs <= 0;
                                                                end
                                                end
                                end
                        // Read Request
                        5:     begin
                                        counter <= (enable == 1) ?
                                            counter - 1 : counter;
                                        if (intr == 0 && counter == 0)
                                                begin
                                                        state <= state + 1;
                                                        counter <= 1;
                                                        rd <= 0;
                                                end
                                end
                        // Read Request
                        6:     begin
                                        counter <= (enable == 1) ?
                                        counter - 1 : counter;
```

```
                                if (intr == 1 && counter == 0)
                                        begin
                                             state <= state + 1;
                                             keycode <= sensor_in;
                                             counter <= 1;
                                        end
                        end
                // Read Request
                7:      begin
                                counter <= (enable == 1) ?
                                    counter - 1 : counter;
                                if (intr == 1 && counter == 0)
                                        begin
                                              state <= state +  1;
                                              rd <= 1;
                                        end
                        end
                // Back To Normal
                8:      begin
                                counter <= (enable == 1) ?
                                    counter - 1 : counter;
                                if (intr == 1 && counter == 0)
                                        begin
                                                state <= 0;
                                           sensor_data <= keycode;
                                                sensor_ready <= 1;
                                                cs <= 1;
                                        end
                        end
                // Default
                default:
                        begin
                                state <= 0;
                                active <= 0;
                                counter <= 0;
                                sensor_ready <= 0;
                                sensor_data <= 0;
                                cs <= 1;
                                wr <= 1;
                                rd <= 1;
                        end
                endcase
        end

    end

endmodule
```

SENSOR DECODER MODULE

```
module Sensor_decoder(clk, reset, start_sample, sensor_ready,
                       sensor_data, initiate, stream_ready, sensor_sel,
                                                    sensor_stream_raw);

input clk, reset, start_sample, sensor_ready;
input [7:0] sensor_data;

output initiate, stream_ready;
output [15:0] sensor_sel;
output [55:0] sensor_stream_raw;

reg initiate, stream_ready;
reg [4:0] state;
reg [15:0] sensor_sel;
reg [55:0] sensor_stream_raw, keycode;

always @ (posedge clk)
        begin
        if (reset)
                begin
                state <= 0;
                keycode <= 0;
                sensor_sel <= 16'b0000000000000000;
                initiate <= 0;
                stream_ready <= 0;
                sensor_stream_raw <= 0;
                end
        else
                case (state)
                // Idle State
                0:      begin
                                sensor_stream_raw <= 0;
                                stream_ready <= 0;
                                sensor_sel <= 16'b0000000000000000;
                                initiate <= 0;
                                if (start_sample == 1)
                                        begin
                                                state <= state + 1;
                                                initiate <= 1;
                                                sensor_sel <=
                                                     16'b0000001000000001;
                                        end
                        end
                1:      begin
                                initiate <= 0;
                                if (sensor_ready == 1)
                                        begin
                                                keycode[7:0] <=
                                                             sensor_data;
                                                state <= state + 1;
                                                initiate <= 1;
                                                sensor_sel <=
                                                    16'b0000010000000010;
                                        end
                        end
                2:   begin
                                initiate <= 0;
                                if (sensor_ready == 1)
                                        begin
                                                keycode[15:8] <=
                                                             sensor_data;
                                                state <= state + 1;
                                                initiate <= 1;
                                                sensor_sel <=
                                                    16'b0000100000000100;
                                        end
                        end
                3:      begin
                                initiate <= 0;
                                if (sensor_ready == 1)
                                        begin
                                                keycode[23:16] <=
                                                             sensor_data;
                                                state <= state + 1;
                                                initiate <= 1;
                                                sensor_sel <=
                                                    16'b0001000000001000;
                                        end
                        end
                4:      begin
                                initiate <= 0;
                                if (sensor_ready == 1)
                                        begin
                                                keycode[31:24] <=
                                                             sensor_data;
                                                state <= state + 1;
                                                initiate <= 1;
                                                sensor_sel <=
                                                    16'b0010000000010000;
                                        end
                        end
                5:      begin
                                initiate <= 0;
                                if (sensor_ready == 1)
                                        begin
                                                keycode[39:32] <=
                                                             sensor_data;
                                                state <= state + 1;
                                                initiate <= 1;
                                                sensor_sel <=
                                                    16'b0100000000100000;
                                        end
                        end
                6:      begin
                                initiate <= 0;
                                if (sensor_ready == 1)
                                        begin
                                                keycode[47:40] <=
                                                             sensor_data;
                                                state <= state + 1;
                                                initiate <= 1;
```

```
                                                sensor_sel <=
                                                    16'b0100000000100000;
                                        end
                        end
                7:      begin
                                initiate <= 0;
                                if (sensor_ready == 1)
                                        begin
                                                keycode[55:48] <=
                                                              sensor_data;
                                                state <= state + 1;
                                        end
                        end
                8:      begin
                        state <= 0;
                        sensor_stream_raw <= keycode;
                        stream_ready <= 1;
                        end
                default:        begin
                                state <= 0;
                                sensor_sel <= 16'b0000000000000000;
                                stream_ready <= 0;
                                sensor_stream_raw <= 0;
                                end
                endcase

        end

endmodule
```

SENSOR CONFIGURATION MODULE

```
module Sensor_config(clk, reset, update_zero, sensor_stream_averaged,
                              ssa_ready, zero_offset, sensor_display);

input clk, reset, ssa_ready, update_zero;
input [55:0] sensor_stream_averaged;

output [55:0] zero_offset, sensor_display;

reg zeroing;
reg [1:0] state;
reg [55:0] zero_offset;

always  @ (posedge clk)
        begin

                if (reset)
                begin
                        state <= 0;
                        zero_offset <= 0;
                        zeroing <= 0;
                end

        else

                case (state)
                // Idle State
                0:      begin
                                zeroing <= 0;
                                if (update_zero == 1)
                                        begin
                                                state <= state + 1;
                                                zeroing <= 1;
                                        end
                        end
                1:      begin
                                        if (ssa_ready == 1)
                                        begin
                                                state <= 0;
                                                zero_offset <=
                                                   sensor_stream_averaged;
                                        end
                        end
                endcase

        end

        assign sensor_display = (reset) ? 0 : (ssa_ready) ? ((zeroing) ?
                        0 : ((sensor_stream_averaged >= zero_offset) ?
                        sensor_stream_averaged-zero_offset: 0)) :
                        sensor_display;

endmodule
```

SENSOR COMMUNICATOR MODULE

```
module Sensor_communicator(clk, reset, start_sensor_blip, stream_ready,
                              sensor_stream_raw, start_sample,
                              sensor_stream_averaged, ssa_ready);


input clk, reset, start_sensor_blip, stream_ready;
input [55:0] sensor_stream_raw;

output start_sample, ssa_ready;
output [55:0] sensor_stream_averaged;

reg start_sample, ssa_ready;
reg [3:0] state;
reg [55:0] sensor_stream_averaged, s1, s2, s3, s4, s5, s6, s7, s8;
reg [58:0] sum;

always @ (posedge clk)
        begin
        if (reset)
                begin
                        state <= 0;
                        s1 <= 0;
                        s2 <= 0;
                        s3 <= 0;
                        s4 <= 0;
                        s5 <= 0;
                        s6 <= 0;
                        s7 <= 0;
                        s8 <= 0;
                        start_sample <= 0;
                        sensor_stream_averaged <= 0;
                        ssa_ready <= 0;
                end
        else
                case (state)
                // Idle State
                0:      begin
                        s1 <= 0;
                        s2 <= 0;
                        s3 <= 0;
                        s4 <= 0;
                        s5 <= 0;
                        s6 <= 0;
                        s7 <= 0;
                        s8 <= 0;
                        start_sample <= 0;
                        sensor_stream_averaged <= 0;
                        ssa_ready <= 0;
                                if (start_sensor_blip == 1)
                                        begin
                                                state <= state + 1;
                                                start_sample <= 1;
                                        end
                        end
```

```
1:      begin
              start_sample <= 0;
              if (stream_ready == 1)
                     begin
                            s1 <= sensor_stream_raw;
                            state <= state + 1;
                            start_sample <= 1;
                     end
        end
2:    begin
              start_sample <= 0;
              if (stream_ready == 1)
                     begin
                            s2 <= sensor_stream_raw;
                            state <= state + 1;
                            start_sample <= 1;
                     end
        end
3:      begin
              start_sample <= 0;
              if (stream_ready == 1)
                     begin
                            s3 <= sensor_stream_raw;
                            state <= state + 1;
                            start_sample <= 1;
                     end
        end
4:      begin
              start_sample <= 0;
              if (stream_ready == 1)
                     begin
                            s4 <= sensor_stream_raw;
                            state <= state + 1;
                            start_sample <= 1;
                     end
        end
5:      begin
              start_sample <= 0;
              if (stream_ready == 1)
                     begin
                            s5 <= sensor_stream_raw;
                            state <= state + 1;
                            start_sample <= 1;
                     end
        end
6:      begin
              start_sample <= 0;
              if (stream_ready == 1)
                     begin
                            s6 <= sensor_stream_raw;
                            state <= state + 1;
                            start_sample <= 1;
                     end
        end
7:      begin
              start_sample <= 0;
              if (stream_ready == 1)
```

```
                                        begin
                                                s7 <= sensor_stream_raw;
                                                state <= state + 1;
                                                start_sample <= 1;
                                        end
                        end
                8:      begin
                                start_sample <= 0;
                                if (stream_ready == 1)
                                        begin
                                                s8 <= sensor_stream_raw;
                                                state <= state + 1;
                                                start_sample <= 1;
                                        end
                        end
                9:      begin
                                sum <= s1 + s2 + s3 + s4 + s5 +
                                                        s6 + s7 + s8;
                                sensor_stream_averaged <= sum[58:3];
                                ssa_ready <= 1;
                                state <= 0;
                        end
                default:        begin
                                        state <= 0;
                                        s1 <= 0;
                                        s2 <= 0;
                                        s3 <= 0;
                                        s4 <= 0;
                                        s5 <= 0;
                                        s6 <= 0;
                                        s7 <= 0;
                                        s8 <= 0;
                                        start_sample <= 0;
                                        sensor_stream_averaged <= 0;
                                        ssa_ready <= 0;
                                end
                endcase

        end

endmodule
```

GPS MODULE

```
module GPSmodule(clk, reset, data, x, y,
               data_ready, fix, gpsstate);

input clk, reset, data;

output fix, data_ready;
output [4:0] gpsstate;
output [5:0] x, y;

reg [5:0] x,y;

wire gps_ready;
wire [5:0] x_gps, y_gps;
wire fix;
wire reset;
wire [7:0] ascii_byte;

GPS_clock gpsclk(clk, reset, counting, enable);

GPS_decoder gpsdec(clk, reset, enable, data, counting, data_ready,
                                                     ascii_byte);

GPScommunicator gps(clk, reset, data_ready, ascii_byte, x_gps, y_gps,
                                              fix, gpsstate, gps_ready);

always @ (negedge gps_ready)
        begin
        x <= x_gps;
        y <= y_gps;
        end
endmodule
```

GPS CLOCK MODULE

```
module GPS_clock(clk, reset, counting, enable);

input clk, reset, counting;

output enable;

//  Clock Speed / GPS Baud Rate = Clock Cycles Per Bit
//  (50Mhz)*(2)/(4*4800) = 5207
parameter clocks_per_bit = 5207;

reg enable;
reg [12:0] counter;

always @ (posedge clk)
        begin

        if (reset == 1)
                begin
                enable <= 1'b0;
                counter <=13'd0;
                end

        if (counting)
                begin

                if (counter == ((clocks_per_bit / 2) - 1))
                        begin
                        enable <= 1'b1;
                        counter <= counter + 1;
                        end

                else if (counter == (clocks_per_bit - 1))
                        counter <= 13'd0;

                else
                        begin
                        enable <= 1'b0;
                        counter <= counter + 1;
                        end
                end

        else
                begin
                enable <= 1'b0;
                counter <=13'd0;
                end

        end

endmodule
```

GPS DECODER MODULE

```
module GPS_decoder(clk, reset, enable, data, counting, data_ready,
                                                            data_out);

input clk, reset, enable, data;

output counting, data_ready;
output [7:0] data_out;

reg counting, data_ready, decoding, decoding2;
reg [3:0] state, start_count;
reg [7:0] keycode, data_out;

always @ (posedge clk)
        begin
        if (reset)
                begin
                state <= 0;
                keycode <= 0;
                counting <= 0;
                data_ready <= 0;
                data_out <= 0;
                start_count <= 0;
                decoding <= 0;
                decoding2 <= 0;
                end

        else
                begin

                if (decoding == 0)
                        begin
                        state <= 0;
                        data_ready <= 0;
                        decoding2 <= 0;

                        if (data)
                                begin
                                counting <= 1;

                                if (enable)
                                        begin

                                        if (start_count[3])
                                                start_count <=
                                                        start_count;

                                        else
                                        start_count <= start_count + 1;
                                        end
                                end
                else
                        begin

                        if (start_count[3])
                                begin
                                decoding <= 1;
                                counting <= 0;
                                state <= 0;
                                start_count <= 0;
                                end
                        end
                end

                else
                        begin
                        data_ready <= 0;
                        if (decoding2)
                                if (state == 0)
                                        if (!data)
                                                counting <= 1;
                        if (decoding2 == 0)
                                counting <= 1;

                        case (state)
                        4'd0: // Start Bit
                                if (enable)
                                        state <= 1;
                        4'd1: // Bit 0
                                if (enable)
                                        begin
                                        keycode[0] <= data;
                                        state <= state + 1;
                                        end

                        4'd2: // Bit 1
                                if (enable)
                                        begin
                                        keycode[1] <= data;
                                        state <= state + 1;
                                        end

                        4'd3: // Bit 2
                                if (enable)
                                        begin
                                        keycode[2] <= data;
                                        state <= state + 1;
                                        end

                        4'd4: // Bit 3
                                if (enable)
                                        begin
                                        keycode[3] <= data;
                                        state <= state + 1;
                                        end

                        4'd5: // Bit 4
                                if (enable)
                                        begin
                                        keycode[4] <= data;
                                        state <= state + 1;
                                        end
```

```
                              4'd6: // Bit 5
                                    if (enable)
                                          begin
                                          keycode[5] <= data;
                                          state <= state + 1;
                                          end

                              4'd7: // Bit 6
                                    if (enable)
                                          begin
                                          keycode[6] <= data;
                                          state <= state + 1;
                                          end

                              4'd8: // Bit 7
                                    if (enable)
                                          begin
                                          keycode[7] <= data;
                                          state <= state + 1;
                                          end

                              4'd9: // Stop Bit
                                    if (enable)
                                          begin
                                          counting <= 0;
                                          state <= 0;
                                          data_out <= keycode;
                                          data_ready <= 1;
                                          decoding2 <= 1;
                                          end
                              endcase

                              end
                              end
                        end
endmodule
```

GPS COMMUNICATOR MODULE

```
module GPScommunicator(clk, reset, data_ready, data_in, lat,
                                      long, fix, gpsstate, pos_ready);

input clk, data_ready, reset;
input [7:0] data_in;

output fix, pos_ready;
output [4:0] gpsstate;
output [5:0] lat, long; //Minutes

reg fix, pos_ready;
reg [4:0] gpsstate, counter;
reg [5:0] lat, long;
reg [16:0] latitude, longitude;

parameter idle = 0;
parameter header1 = 1;
parameter header2 = 2;
parameter header3 = 3;
parameter header4 = 4;
parameter header5 = 5;
parameter Time = 6;
parameter va = 7;
parameter latdeg = 8;
parameter latmin = 9;
parameter ns = 10;
parameter longdeg = 11;
parameter longmin = 12;
parameter ew = 13;
parameter speed = 14;
parameter cgood = 15;
parameter fixdate = 16;
parameter magvar = 17;
parameter ew2 = 18;

always @ (posedge clk)
        begin

        if (reset)
                begin
                gpsstate <= idle;
                latitude <= 17'd0;
                longitude <= 17'd0;
                lat <= 0;
                long <= 0;
                fix <= 0;
                pos_ready <= 0;
                end

        else if (data_ready)
                case (gpsstate)
                idle: if (data_in == 36)//$
                                gpsstate <= header1;
                        else gpsstate <= idle;
```

```
header1: if (data_in == 71)//G
                gpsstate <= header2;
        else gpsstate <= idle;
header2: if (data_in == 80)//P
                gpsstate <= header3;
        else gpsstate <= idle;
header3: if (data_in == 82)
                gpsstate <= header4;
        else gpsstate <= idle;
header4: if (data_in == 77)
                gpsstate <= header5;
        else gpsstate <= idle;
header5: if (data_in == 67)
                gpsstate <= va;
        else gpsstate <= idle;
va: if (data_in == 44)
                gpsstate <= Time;
        else
                begin
                fix <= data_in[0];
                gpsstate <= idle;
                end
Time: if (data_in == 44)
                begin
                counter <= 1;
                gpsstate <= latdeg;
                end
        else gpsstate <= Time;
latdeg: if (counter == 0)
                begin
                counter <= 4;
                gpsstate <= latmin;
                end
        else
                begin
                counter <= counter - 1;
                gpsstate <= latdeg;
                end
latmin: if (data_in == 44)
                begin
                gpsstate <= ns;
                lat <= 55586 – latitude /16;
                end
        else
                begin
                if (data_in != 46)
                latitude <= latitude * 10 +
                                (data_in - 48);
                counter <= counter - 1;
                gpsstate <= latmin;
                end
ns: if (data_in == 44)
                begin
                counter <= 1;
                gpsstate <= longdeg;
                end
        else
```

```
                gpsstate <= ns;
longdeg: if (counter == 0)
                begin
                counter <= 4;
                gpsstate <= longmin;
                end
        else
                begin
                counter <= counter - 1;
                gpsstate <= longdeg;
                end
longmin: if (data_in == 44)
                begin
                gpsstate <= ew;
                long <= (217001 - longitude) / 16;
                long <= 54;
                end
        else
                begin
                if(data_in != 46)
                longitude <= longitude * 10 +
                        (data_in - 48);
                counter <= counter - 1;
                gpsstate <= longmin;
                end
ew: if (data_in == 44)
                gpsstate <= speed;
        else
                gpsstate <= ew;
speed: if (data_in == 44)
                gpsstate <= cgood;
        else
                begin
                gpsstate <= speed;
                end
cgood: if (data_in == 44)
                gpsstate <= fixdate;
        else
                gpsstate <= cgood;
fixdate: if (data_in == 44)
                gpsstate <= magvar;
        else
                gpsstate <= fixdate;
magvar: if (data_in == 44)
                gpsstate <= ew2;
        else
                gpsstate <= magvar;
ew2: if (data_in == 44)
                begin
                gpsstate <= idle;
                pos_ready <= 1;
                end
        else
                gpsstate <= ew2;
default:        begin
                gpsstate <= idle;
                latitude <= 0;
```

```
                                        longitude <= 0;
                                        end
                        endcase

                        else
                                pos_ready <= 0;

                end

        endmodule
```

VGA MODULE

```
module VGA_module(clk,hsync,vsync,hcount,vcount,vertical,vreset);

   input clk;     // 25Mhz
   input vertical;
   output hsync;
   output vsync;
   output [9:0] hcount, vcount;
   output vreset;

   reg         pcount;    // used to generate pixel clock at 12.5 MHZ
   wire        en = (pcount == 0);
   always @ (posedge clk) pcount <= ~pcount;

   // Sync Signals


   reg         hsync,vsync;
   reg [9:0]   hcount;   // pixel number on current line
   reg [9:0]   vcount;  // line number

        parameter hsbeg = 666;
        parameter hsend = 733;
        parameter hsres = 802;
        parameter vsbeg = 486;
        parameter vsend = 494;
        parameter vsres = 524;

   // horizontal: 794 pixels = 31.76us
   // display 640 pixels per line
   wire        hsyncon,hsyncoff,hreset;
   assign      hsyncon = en & (hcount == hsbeg);
   assign      hsyncoff = en & (hcount == hsend);
   assign      hreset = en & (hcount == hsres);

   // vertical: 528 lines = 16.77us
   // display 480 lines
   wire        vsyncon,vsyncoff,vreset;
   assign      vsyncon = hreset & (vcount == vsbeg);
   assign      vsyncoff = hreset & (vcount == vsend);
   assign      vreset = hreset & (vcount == vsres + vertical);

   // sync and blanking
   always @(posedge clk)
        begin
               hcount <= en ? (hreset ? 0 :  hcount + 1) : hcount;
               hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;//active low
               vcount <= hreset ? (vreset ? 1 : (vcount == vsres-1) ? 0
                                              :  vcount + 2) : vcount;
               vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;//active low
        end

endmodule
```

CHARACTER DISPLAY MODULE

```
///////
//
// video character string display
//
///////

module char_string_display (vclock,hcount,vcount,pixel,cstring,cx,cy);

   parameter NCHAR = 26;       // number of 8-bit characters in cstring
   parameter NCHAR_BITS = 5; // number of bits in NCHAR
   parameter color = 0;
   parameter bg_color = 0;

   input vclock;       // 65MHz clock
   input [9:0] hcount; // horizontal index of current pixel (0..1023)
   input [9:0] vcount; // vertical index of current pixel (0..767)
   output [2:0] pixel; // char display's pixel
   input [NCHAR*8-1:0] cstring;       // character string to display
   input [9:0] cx;
   input [9:0] cy;

   // 1 line x 8 character display (8 x 12 pixel-sized characters)

   wire [9:0]  hoff = hcount-1-cx;
   wire [9:0]  voff = vcount-cy;
   //< NCHAR
   wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+4:4];
   wire [2:0]  h = hoff[3:1];            // 0 .. 7
   wire [3:0]  v = voff[4:1];           // 0 .. 11

   // look up character to display (from character string)
   reg [7:0]  char;
   integer  n;
   always @(*)
        for (n=0 ; n<8 ; n = n+1 )// 8 bits per character (ASCII)
         char[n] <= cstring[column*8+n];

   // look up raster row from font rom
   wire reverse = char[7];
   wire [10:0] font_addr = char[6:0]*12 + v;// 12 bytes per character
   wire [7:0]  font_byte;
   font_rom f(font_addr,vclock,font_byte);

   // generate character pixel if we're in the right h,v area
   wire [2:0] cpixel = (font_byte[7 - h] ^ reverse) ? color : bg_color;
   wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <=
                            cx+NCHAR*16) & (vcount < cy + 24));
   wire [2:0] pixel = dispflag ? cpixel : 0;

endmodule
```

FONT ROM MODULE

```
/*******************************************************************
*     This file is owned and controlled by Xilinx and must be used
*     solely for design, simulation, implementation and creation of
*     design files limited to Xilinx devices or technologies. Use
*     with non-Xilinx devices or technologies is expressly prohibited
*     and immediately terminates your license.
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
*     FOR A PARTICULAR PURPOSE.
*     Xilinx products are not intended for use in life support
*     appliances, devices, or systems. Use in such applications are
*     expressly prohibited.
*     (c) Copyright 1995-2004 Xilinx, Inc.
*     All rights reserved.
*******************************************************************/
module font_rom (
        addr,
        clk,
        dout);    // synthesis black_box

input [10 : 0] addr;
input clk;
output [7 : 0] dout;

// synopsys translate_off

        BLKMEMSP_V6_1 #(
                11,     // c_addr_width
                "0",    // c_default_data
                1536,   // c_depth
                0,      // c_enable_rlocs
                0,      // c_has_default_data
                0,      // c_has_din
                0,      // c_has_en
                1,      // c_has_limit_data_pitch
                0,      // c_has_nd
                0,      // c_has_rdy
                0,      // c_has_rfd
                0,      // c_has_sinit
                0,      // c_has_we
                18,     // c_limit_data_pitch
                "font_rom.mif", // c_mem_init_file
                0,      // c_pipe_stages
                0,      // c_reg_inputs
                "0",    // c_sinit_value
                8,      // c_width
                0,      // c_write_mode
                "0",    // c_ybottom_addr
                1,      // c_yclk_is_rising
                1,      // c_yen_is_high
                "hierarchy1",  // c_yhierarchy
                0,      // c_ymake_bmm
                "16kx1",       // c_yprimitive_type
                1,      // c_ysinit_is_high
                "1024", // c_ytop_addr
                0,      // c_yuse_single_primitive
                1,      // c_ywe_is_high
                1)      // c_yydisable_warnings
        inst (
                .ADDR(addr),
                .CLK(clk),
                .DOUT(dout),
                .DIN(),
                .EN(),
                .ND(),
                .RFD(),
                .RDY(),
                .SINIT(),
                .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of font_rom is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of font_rom is "black_box"

endmodule
```

RECTANGLE GENERATION MODULE

```
module Rectangle_generation(sensor_display,hcount,vcount,pixel);

//Define the size and color of the necessary parameters
   parameter barwidth = 90;
   parameter barcolor = 3'b100;
   parameter linecolor = 3'b110;
   parameter white = 3'b111;
   parameter black = 3'b000;
   parameter screen_height = 479;
   parameter screen_width = 639;

   input [55:0] sensor_display;
   input [9:0] hcount;
   input [9:0] vcount;

   output [2:0] pixel;

   reg [2:0] pixel;      // Output pixel of current location

   always @ (sensor_display or hcount or vcount)
        begin

        // Set pixel to the color of the corresponding object if the
        // current hcount and vcount location is within the boundaries
        // of the object's definition
        pixel <=
               ((vcount >= 2 && vcount <= 25)&&((hcount >= 0 && hcount
               <= 112)||(hcount >= 512 && hcount <= screen_width))) ?
               linecolor : ((vcount >= 0 && vcount <= 1)&&(hcount >= 0
               && hcount <= screen_width)) ?  linecolor : ((vcount >= 25
               && vcount <= 27)&&(hcount >= 0 && hcount <=
               screen_width)) ? linecolor : ((vcount >= 192 && vcount
               <= 196)&&(hcount >= 0 && hcount <= screen_width)) ?
               linecolor : ((vcount >= 28 && vcount <= 191)&&(hcount >=
               321 && hcount <= 324)) ? linecolor : ((hcount >= 0 &&
               hcount <= (barwidth)) && (vcount >= screen_height - 5 –
               sensor_display[7:0] && vcount <= (screen_height))) ?
               barcolor : ((hcount >= (barwidth + 1) && hcount <= (2 *
               barwidth)) && (vcount >= screen_height - 5 –
               sensor_display[15:8] && vcount <= (screen_height))) ?
               barcolor : ((hcount >= ((2 * barwidth) + 1) && hcount <=
               (3 * barwidth)) && (vcount >= screen_height - 5 –
               sensor_display[23:16] && vcount <= (screen_height))) ?
               barcolor : ((hcount >= ((3 * barwidth) + 1) && hcount <=
               (4 * barwidth)) && (vcount >= screen_height - 5 –
               sensor_display[31:24] && vcount <= (screen_height))) ?
               barcolor : ((hcount >= ((4 * barwidth) + 1) && hcount <=
               (5 * barwidth)) && (vcount >= screen_height - 5 –
               sensor_display[39:32] && vcount <= (screen_height))) ?
               barcolor : ((hcount >= ((5 * barwidth) + 1) && hcount <=
               (6 * barwidth)) && (vcount >= screen_height - 5 –
               sensor_display[47:40] && vcount <= (screen_height))) ?
               barcolor : ((hcount >= ((6 * barwidth) + 1) && hcount <=
               (7 * barwidth+2)) && (vcount >= screen_height - 5 –
```

```
              sensor_display[55:48] && vcount <= (screen_height))) ?
              barcolor :((vcount >= 52 && vcount <= 64)&&(hcount >= 0
              && hcount <= 320)) ?
              white :
                     ((vcount >= 88 && vcount <= 93)&&(hcount >= 0 &&
                     hcount  <= 320)) ?
              white :
                     ((vcount >= 118 && vcount <= 123)&&(hcount >= 0
                     && hcount <= 320)) ?
              white :
                     ((vcount >= 146 && vcount <= 151)&&(hcount >= 0
                     && hcount <= 320)) ?
              white :
                     ((vcount >= 176 && vcount <= 191)&&(hcount >= 0
                     && hcount <= 320)) ?
              white :
                     ((vcount >= 52 && vcount <= 63)&&(hcount >= 325
                     && hcount <= screen_width)) ?
              white :
                     ((vcount >= 88 && vcount <= 93)&&(hcount >= 325
                     && hcount <= screen_width)) ?
              white :
                     ((vcount >= 116 && vcount <= 137)&&(hcount >= 325
                     && hcount <= screen_width)) ?
              white :
                     ((vcount >= 162 && vcount <= 191)&&(hcount >= 325
                     && hcount <= screen_width)) ?
              white :
                     ((vcount >= 64 && vcount <= 89)&&(hcount >= 613
                     && hcount <= screen_width)) ?
              white :
                     ((vcount >= 94 && vcount <= 133)&&(hcount >= 613
                     && hcount <= screen_width)) ?
              white :
                     ((vcount >= 138 && vcount <= 162)&&(hcount >= 613
                     && hcount <= screen_width)) ?
              white :
                     ((vcount >=220 && vcount <=
                     screen_height)&&(hcount >= 0 && hcount <=
                     screen_width)) ?
              white:
                     ((vcount >=197 && vcount <= 219)&&(hcount >= 113
                     && hcount <= screen_width)) ?
              white :
              black;
       end

endmodule
```