FPGA-Scope: A Labkit Implemented Oscilloscope

Kevin Linke, Anartya Mandal

***Abstract*—For our 6.111 final project, we implemented a digital oscilloscope in Verilog on the labkit. Our oscilloscope is able to sample an analog input, display it with vertical and horizontal scaling, and analyze it. The scalability of the FPGA-Scope sets it apart from other oscilloscopes currently available; bandwidth, accuracy and number of inputs can improved without replacing the entire oscilloscope.**

I. Introduction

OUR final project is a digital oscilloscope implemented on the labkit's Field Programmable Gate Array with a computer monitor as the display. In order to accomplish this, we use analog-to-digital converter to sample an analog user input. We store these samples in the FPGA BRAM and extract information about the input signal from them, such as period, peak-to-peak voltage, and offset. The scale of the waveform on the monitor is set via a user interface through the buttons on the labkit. Also displayed will be images of numbers that will provide the user with numeric measurements from the waveform, as well as grid-lines to provide scale. One advantage of the FPGA-Scope is that it would provide a much higher level of scalability than other oscilloscopes. Improving bandwidth and accuracy would simply be a matter of using a better ADC, FPGA or memory. This flexibility would avoid the huge costs of purchasing new oscilloscopes. A further feature that differentiates the FPGA-Scope from other oscilloscopes is the number of signals it can measure. Most oscilloscopes only have four probes, but it would be possible to add many analog inputs to the FPGA-Scope, which would be useful for devices with many outputs, such as an electroencephalogram.

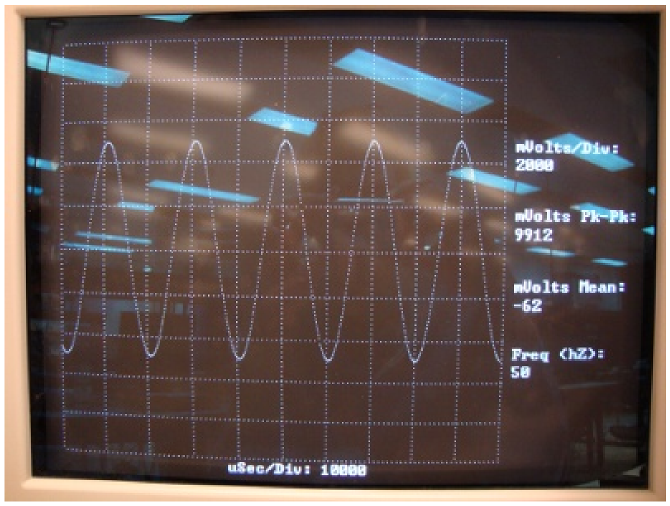

Figure 1. FPGA-Scope

II. Implementation

A. Data Collection (Anartya)

Data collection in the FPGA-Scope is performed by a group of three modules: the ADC, ADC controller, and samples BRAM. Together, they sample the analog input at a user-specified frequency, then store the samples in a BRAM for later scaling and display.

1) ADC:

Description: The FPGA-Scope uses an AD574 12-bit analog-to-digital converter to sample the analog input at a user-defined sampling frequency. An input signal of the appropriate frequency is fed to the ADC by the ADC controller. The status output of the ADC is connected to the samples BRAM, where it determines when the 12-bit data samples are ready to be stored in memory. The speed of the ADC limits the maximum frequency of the input signal to our oscilloscope; we expected the max sampling frequency of the AD574 to be 10kHz, so our expected max input frequency was 1kHz in order to provide ten samples of display resolution per period. However, when testing the completed scope, we discovered that the ADC could comfortably sample at 37.5 kHz, making our max input frequency 3.75 kHz.

Implementation: We used a 12-bit analog-to-digital converter (AD574) from Analog Devices. There were many parameters that we had to first initialize in order to power the chip, and convert data at the proper rate. There are a couple of ways we can initialize the input such as performing 8-bit conversion instead of 12-bit, or allowing a unipolar voltage range (between 0 and 5V, or 0 and 10 Volts) instead of bipolar voltage range (between -2.5 and 2.5V, or -5 and 5 Volts). In our case, we wanted to perform 12-bit analog-to-digital conversion with a bipolar voltage range, so we had to wire the chip to the proper voltage sources and logic in order to achieve our conversion. After initializing the chip, we had to figure out a way to sample the input analog data, with a variable sampling frequency. After some investigation into the data sheet, we found that we could use a specified Read/Convert input that should be a logical one when we are converting an analog sample into a digital sample. The ADC also provides a status signal output that is a logical one when the ADC is performing the conversion. This status signal is important in that when the status signal transitions to a logical zero from a logical one, we are now ready to read the data from the ADC. One difficulty we had was figuring out whether to sample the ADC using the chip's specified Read/Convert input or Conversion Enable (CE) input. According to the chip specifications, Conversion Enable(CE) can be controlled to initiate conversion as well, but through testing we found that keeping Conversion Enable(CE) a logical one and varying the Read/Convert signal allowed us to properly sample. Another difficulty we had was with regard

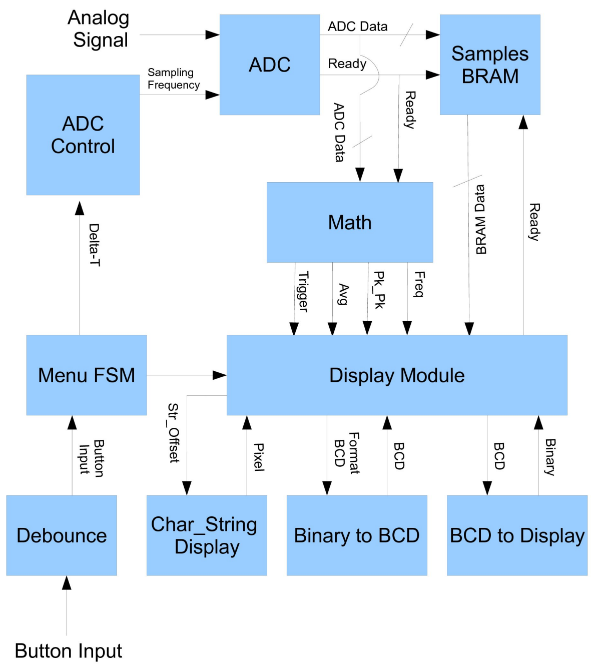

Figure 2. FPGA-Scope block diagram

to timing. According to the data sheet, when the Read/Convert signal is a logical zero, it needs to be held constant for a min of 450 ns, in order for the chip to properly work. When we tried sampling the given inputs with high enough frequencies, we found the chip to not perform conversion properly. We found that the status signal was not appearing when we increase sampling rates to high levels, that way we knew that the chip specifications we definitely the limiting factor.

Testing: In order to test the ADC, we initially just used the FPGA's user input port to feed in our 12-bit digitized samples with a fixed 1kHz sampling frequency implemented using a prototype ADC controller module that created a 1kHz signal with a 50% duty signal and fed these inputs into the hex display to see if the sampling was happening properly. Using the function generator, we were able to create analog waveforms to feed into our ADC such as square, triangle and sine waves. We found that for sine waves that were less that 1Hz, we could see the hex display follow an oscillating pattern slowly moving from a 3-digit hexadecimal 000 (-5V) to a 3-digit hexadecimal FFF (+5V) . Since this confirmed that our ADC chip was working properly, we decided to vary the sampling frequencies and duty cycles of the prototype ADC Controller, and see if we got the proper hex display digits. We also used the logic analyzer to feed our digitized values and used the magnitude formating option in order to display a drawn sine wave. At all times we always used the oscilloscope to look at our input signal from the function generator and status outputs of the ADC to make sure that incorrect measurements were not being taken.

2) *ADC Controller:*

Description: The ADC controller takes as an input the delta-t set by the menu FSM, and produces a control signal of the appropriate frequency that tell the ADC when to sample the incoming signal. For lower frequency inputs, samples need to be collected less frequently, because only 748 samples can be displayed on the monitor, but multiple periods of the signal need to be sampled.

Implementation: Since the job of the ADC Controller is to create control signals of various frequencies and duty cycles in order to sample the ADC, we have two variables, duty cycle, and delta-t as inputs, so that during instantiation, it could be more flexible to produce sampling signals of varying width and speed. We had to use a count register to keep track of how many clock cycles have passed until we get to our desired frequency and make sure that our output Read/Convert signal is valid for the desired duty cycle. One of the difficulties, I had was figuring out how to incorporate duty cycle into the ADC controller. The duty cycle is very important, because we have to meet the timing constraints of the ADC. The timing constrains specifically states that the Read/Convert Signal need to be a valid low for at least 450ns. Therefore, simply creating a variable frequency 50% duty cycle signal can only allow us to sample under limited flexabilities.

Testing: To test the ADC Controller, at first I just took the output through one of the user ports and analyzed it using the oscilloscope and checked the period and duty cycle to make sure that there are correctly timed. After that I directly fed it the Read/Convert signal into the ADC and used the oscilloscope to probe the status signal output of the ADC to see whether the status signal frequency matched the input sampling frequency. I also used the logic analyzer to verify whether the input waveform was being properly sampled in reference to the FPGA's 65Mhz clock.

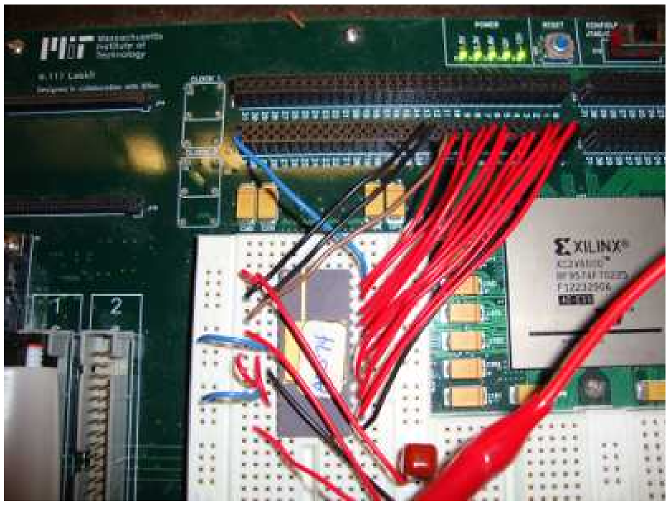

Figure 3. AD574 in labkit setup

3) *Samples BRAM:*

Description: The samples BRAM stores the raw data collected by the ADC. This data is stored here until it is rescaled by the display module for display. Although the screen only displays 748 samples at a time, the BRAM must store three times this many samples, for a total size of 748*3<2^12 bits. The reason for this oversampling is that in order for the waveform to remain in a constant location on the display, we must begin sampling it at a consistent trigger point. Sampling for three times the width of the display window allows us to search through the samples collected for a trigger condition, then display the samples in the window that surrounds this point. We used the method of triggering at peaks.

Implementation: Before implementing the BRAM, we need to know how many samples we need to store in order to determine the size of the BRAM. Since we want to center the waveform at the middle of the screen, we have to use a trigger point in the wave. Since we are using peak triggering, we have to make sure we have at least two periods of the wave to display on the screen. We made two small changes from our original proposal which is storing three periods instead of four and storing twelve samples at a time instead of one, so that the addressing complexity and memory usage is reduced. To implement the samples BRAM, we used IP Core Generator to create a BRAM with the desired 12-bit input and 12-bit addressing ports. In order to write to memory, we have to make sure that the ADC is ready to be read from. When the status signal from the ADC goes from a logical high to a logical low, we want to write to the BRAM exactly when that transition happens. When this condition is satisfied, we assert the write enable signal on the BRAM so that the digitized ADC data gets stored into the BRAM to the current address on the BRAM and then we increment the address to store the next sample at that particular address. But we also have to read from the BRAM, but because we have to read and write, it is necessary to use two BRAM to speed up the process so that while one BRAM is being written to, the other can be read from. To do this we make sure that the write enable signal of one BRAM is always the inverse of the other while the status condition holds from the ADC. When the BRAM is full, we output a full signal to let the display module know to reset the read addresses. The full signal is also important for the ADC to write to the BRAM because we have to make sure the write address is also reset. It is also important that the address does not exceed our max address as calculated before (748*3<2^12 bits). One of the difficulties we had at first was writing to the BRAM. At first, we thought that the status signal and the Read/Convert signal had to be a logical low in order to determine when that the BRAM was ready to be written to. But we found that the BRAM was not storing the correct values, as they all appeared zero. To solve the problem, I coupled the status signal and the Read/Convert signal on the oscilloscope to see whether our logic was correct. We found that both signals were out of phase, and that only status was relevant enough. But more importantly, we found that status is also a logical low before conversion for some time. This meant that we were grabbing data before the analog samples have been converted to digital, hence BRAMs filled with unexpected values.

Testing: To test the BRAM, we used a switch to determine when to read/write to the BRAM, and kept a fixed 1kHz sampling frequency, and a 1 clock cycle width read/enable pulse that allowed me to read data at a test rate of 1Hz. We also created a refill signal that would tell the BRAM to refill whenever, its full, so it would simulate a real time effect. At first we just used the Hex Display to look at the output and input bits into the BRAM in addition to whether the address was incrementing or not. We found that 1Hz read was slow enough so that we can keep tracking of the incrementing effect of sine waves (going from 000 to FFF in hexadecimal in a sinusoidal fashion). Afterwards, we used the logic analyzer to look at both the data being written to and read from the BRAM. With the feature of magnitude display, We were able to plot the waveform drawing of the sine wave on the logic analyzer. This was great because now we are able to tell exactly how many periods have been stored to the BRAM. This will end up to be extremely important in testing the math module where we have to compute parameters of the waveform.

B. Data processing (Anartya)

Data processing in the FPGA-Scope takes raw binary data from data collection and user input modules and converts it to a form can be displayed on the screen. Much of this processing occurs in the math and scaling modules, which extracts the useful information from the analog input and generates a waveform image for display. Other processing occurs in the decimal module, which converts binary numbers to images of decimals that the user can read. The images created by this processing are stored in the numbers and waveform BRAMs.

Description: The math module is responsible for processing the ADC data on its way into the samples BRAM. This processing includes measuring statistics about the input signal for the user. The three main parameters of interest are average voltage, peak-to-peak voltage, and frequency. Average voltage is calculated using a running sum of weighted values in one period. Peak-to-peak voltage is calculated by keeping track of the largest and smallest voltage samples encountered. Frequency is determined by recording the times between two maxima. The math module also determines the trigger address and sends it to the scaling module. The values calculated by the math module are sent to the decimal module for display.

Implementation: The math module is responsible for calculating parameters of the waveform. This is done directly from the ADC outputs to achieve real-time calculations. Our first parameter was peak to peak voltage. To calculate this we use a max register and a min register that would only update its values whenever a data input has a value greater than the highest recorded address. The same goes for the min register, where a data input is stored only if its less than the lowest address recorded. Then we subtract the max register from the min register to get the peak-to-peak voltage. Since our oscilloscope will handle only periodic waveforms, Fourier Transforms will not be necessary. To get the frequency, we used two registers with one being delayed by one clock cycle, in order to detect two max peaks. Since we need at least two periods of a wave in order to determine the two peaks, we know that after a certain number of clock cycles we have found a global max. We initialize both registers to zero, once the non-delayed register finds the first peak equal to the global max, we know that the delayed one has not found one yet and is therefore still set to zero. Once the delayed register finds the first peak, both registers have the same value for the number for clock cycles to reach the max peak. But once the non-delayed register finds the second peak, we know that the delayed one hasn't found it yet. Therefore, we can use the condition that the delayed register cannot have a value of zero or the exact same value as the non-delayed register to be the condition that determines whether or not to find any more peaks. If this condition is not a logical one, then we know that two peaks have been found, and we can subtract the two values of the register to determine the period. We then use a IP core generator divider to divide the period by the sampling frequency to get the frequency of the wave. To find the average voltage, we have to add up all of the samples within one period and divide by the period. We keep a sum register that keeps adding the data input to its stored value after the non-delayed register finds the first peak. It keeps adding these values until the second peak is found. Then we use an IP core generator divider to divide the sum by the period of the waveform to get the average voltage. To find the trigger, we just use the first peak found by the non-delayed register. These values are now encoded in binary, so they are then fed to the decimal display to be further formated to decimal. The greatest difficulty we had was figuring out a good algorithm for peak detections. We spent nearly a week testing several different algorithms that would calculated period for any arbitrary periodic function. But we found that they were really inconsistent. Some worked better for square waves, some for sine waves. Be my code was so general it was inconsistent. So we exploited one major property of our signals, which was that they have to be greater that two periods in order to determine to max peaks. This helped to make our code less generalized and inaccurate. The other major problem was how many bits we should round in order to say that we have reached a global max. This is because noise in the waveforms, doesn't always guarantee that the local maximum is necessarily the recorded global max. Through numerous trial and error methods, we found six bits of rounding enough to get consistent frequency measurements. Since every parameter except peak-to-peak depended on finding peaks, this was provided great calibration.

Testing: In order to test the math module, we used the logic analyzer to look at the contents of the BRAM to determine how fast we needed to generate a sine wave in order to make sure we get exactly three periods. That way we can easily determine what the hex display should give in order to match the appropriate period of the wave for the trigger address. This was just an initial test, we later just used the hex display to just look at the undivided periods, and sums before frequencies and average voltages we done using the IP core dividers. We did the division in hex simply my using the computer's calculator and converted it to decimal to make sure it matched the function generator. We did the same with peak to peak voltage displaying them in hex, and using the computer's calculator to match it with the function generator.

C. *Data Display (Kevin)*

The display module is responsible for converting the raw binary data from the data collection and data processing modules to a form that can be displayed on the monitor and understood by the user. Displaying the input waveform on the screen is handled entirely by the display module, but displaying the numbers determined by the math module requires the assistance of two helper modules, a binary to binary-coded decimal converter, and a module for formatting binary-coded decimals for display on the monitor. Also involved in the display of data is the character string display module, an external module from the 6.111 tools created by Chuang and Terman. This module writes strings of text to the screen, such as delta-t, delta-v and their labels.

1) *Display Module:*

Description: The display module takes the input data stored in the samples BRAM and converts it to the image that is displayed on the screen. Based on the delta-t and delta-v specified by the user, the display module scales the sample data vertically and horizontally, in addition to inverting, shifting and centering it. This feature allows the user to make the waveform fill up the screen appropriately. For instance, if the input signal has a magitude of only 1V, but the user wishes it to fill the entire screen in order to view its fine structure, he or she can set the mVolts/Div to 100. The user may also select

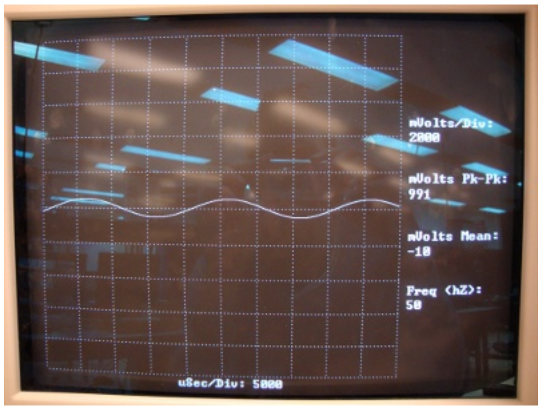

Figure 4. Unscaled 1V sinusoid

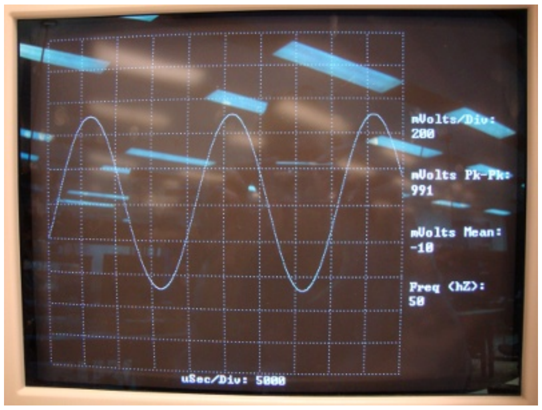

Figure 5. Scaled 1V sinusoid

whether or not to fill the area under the waveform. Leaving the area unfilled works well for low frequency, continuous signals, but for high frequency, or discontinuous signals, such as square waves, the interpolation effect caused by the fill can make the waveform easier to view. Additionally, the display module uses the trigger address from the math module in order to determine which samples from the samples BRAM are actually displayed. Triggering is important because it ensures that a peak of the waveform will always be centered on the screen; without triggering, the waveform randomly move side-to-side on the screen. Lastly, the scaling module adds fixed grid-lines to the image in the waveform BRAM in order to provide scale for the user.

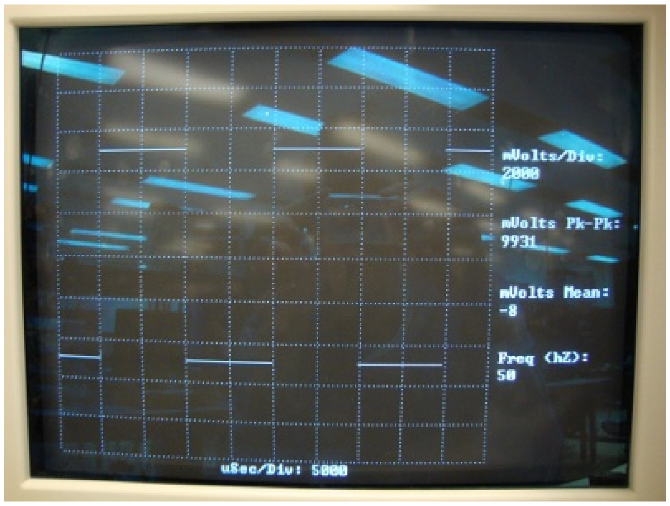

Figure 6. Square wave without fill

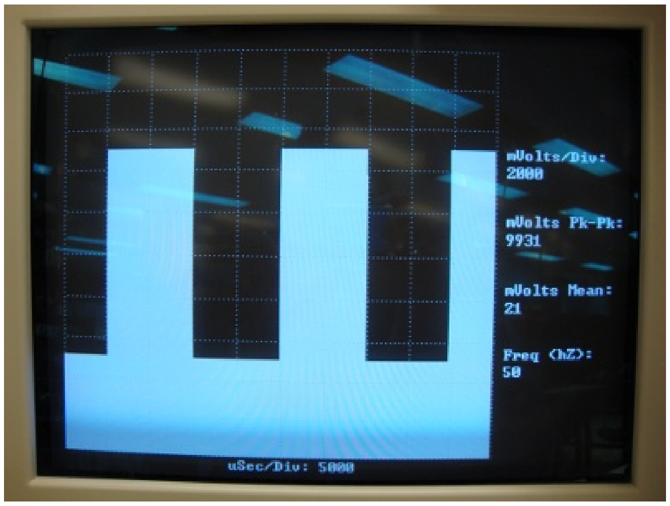

Figure 7. Square wave with fill

Implementation: Initially, we expected that scaling the input waveform data would require that the scaled values be stored in a BRAM for display. This additional memory would necessary to give the oscilloscope sufficient time to invert, center, multiply, and divide each of the values. However, when we implemented the display module, we discovered that all of the scaling operations other than division could be performed in less than one clock cycle. Division required 23 clock cycles, but since we employed a pipelined divider, one sample could be scaled every clock cycle. To compensate, we shifted our address so that each division is requested 23 clock cycles before it is actually needed. According to our original plan, the BRAM where we stored the scaled values was the largest component of our memory usage, so eliminating it represents significant memory savings.

Testing: Because the data display modules and data collection modules were created in parallel, it was necessary to test the display modules with an input BRAM before the samples BRAM module was complete. Consequently, we created a test BRAM called mem_test that we pre-loaded with data corresponding to a sawtooth wave. This BRAM provided a stable, known input that was essential for determining whether the display module was functioning correctly.

2) *Binary to BCD :*

Description: The binary to binary-coded decimal (BCD) converter takes a binary input and converts it to a form that can both be stored in a register and read as a decimal number. This conversion makes the values of average voltage, peak-to-peak voltage, and freq readable for the user.

Implementation: The basic principal behind binary-to-BCD conversion to take a binary number and store it as a series of hex digits. However, even though these hex digits have four bits, they are only allowed to range from zero to nine. As a result, even though they are store in 'binary' the digits can used directly as decimals. The most conceptually simple way to perform such a conversion would be to use a series of counters, one per digit. Each counter would count from zero to nine, and when it overflowed, it would add one to the next higher order digit. Converting a number (n) to BCD would be as simple as incrementing the lowest order counter n times. However, this method would also be slow- converting the number 10,000 would require 10,000 clock cycles. A faster approach would be one that relied more on combinational logic and less on shifting bits. One such method is to create a register representing the decimal places of the BCD output, then shift in the binary number one bit at a time, starting at the least significant bit. Every time a binary bit is shifted in, check each decimal place. If the hex value is greater than or equal to five, add three. By the time the entire binary number is shifted in, the register will contain the converted BCD number. This method would only require 14 clock cylces to convert the number 10,000. However, more speed is possible. Instead of shifting bits, it is possible to interconnect long chains of small modules that perform the operation “if input is greater than or equal to five, add three”. Constructing the BCD in this manner makes it purely combinational, and able to perform conversions in less than one clock cycle.

Testing: Because a binary to BCD converter has very clearly defined outputs with respect to its inputs, testing the converter required only a simple test bench. The test bench fed in the minimum and maximum convertible values, as well as an intermediate value, and confirmed that the output of the converter was correct in each case.

3) *BCD to Display:*

Description: The binary to binary-coded decimal to display converter takes a BCD input and formats it for display on the screen. Thus a decimal number is turned into a well-formatted string that can be used as an input to the character string display module.

Implementation: Even after the binary to BCD converter has converted a binary number to a decimal, the number is still not ready for display on the screen. It must first be converted to a string, which requires mapping each four-bit hex digit to its eight-bit ASCII equivalent. Fortunately, this is as easy as padding a hex three onto the front of the hex digit. After the module has inserted a three before each digit, it also formats the string. Strings displayed by the character string display module are positioned from the end of the string, so an unformatted string would have a variable starting point or leading zeros. To compensate, the BCD to display module determines the order of magnitude of the input number and left shifts it appropriately so that it is left justified and does not have any leading zeros. The ability of the FPGS-Scope to display numbers using the character string display module permitted further memory saving. Initially, we had assumed that a large BRAM would be required to store an image of all the numbers and labels we would display on the screen, but the character string display module has a negligable fixed memory requirement.

D. User Interface (Kevin)

The user interface of the FPGA-Scope allows the user to control the horizontal and vertical scaling of the displayed waveform. It takes user input from the Labkit buttons, and uses it to control the data collection and data processing modules.

Description: The menu FSM takes debounced button inputs from the user and specifies the delta-V and delta-t parameters for the ADC controller, decimal module, delta-t BRAM and scaling module. Button zero specifies that delta-t is to be changed, and button one specifies that delta-v is to be changed. Once either button is pressed, the up and down buttons are used to change the parameter values. After the user is done setting one parameter, he or she presses enter.

Implementation: The menu FSM consists of a state machine with three states: select, delta-t, and delta-v. Buttons zero and one transition from select to delta-t and delta-v respectively, and enter transitions back to select. Initially, we had planned to simply use the up and down arrows to increment and decrement the delta-t and delta-v values. However, since delta-t and delta-v range from 1 to 999,999 and 1 to 9,999 respectively, it would be inconvenient for the user to change their values over their full range one unit at a time. To solve this problem, we implemented an additional interface whereby the user can select a digit within the parameter he or she wishes to change, then increment or decrement that specific digit. Now changing delta-t from 1 to 100,000 requires two changes (increment the 100,000s place and decrement the 1s place) instead of 99,999 increments.

Testing: In order to test the functionality of the menu FSM, we created a test bench that simulated many different possible sequences of button inputs. Using Modelsim, we verified that these sequences produced the correct outputs. For later debugging, and so that we could view the user settings for delta-t and delta-v before they were displayed on the monitor, we used the hex display to show delta-t, delta-v and the state of menu FSM.

III. Conclusions

Overall, the FPGA-Scope was a highly successful project. We achieved our goal of implementing a functional, accurate and scalable digital oscilloscope in Verilog. All of the features were had planned to implement were completed, and our peak detection algorithm proved surprisingly effective. Our maximum input frequency was 3.75 kHz, 275% higher than we had anticipated. Furthermore, memory usage turned out to be much less than expected; 54 Kb instead of 1380 Kb, a factor of 25 reduction. Since the FPGA has roughly 3 Mb of BRAM memory, many more channels could be added to the oscilloscope. The FPGA-Scope is also scalable in several other ways: a lower performance FPGA could be used to save cost, or a higher performance one could be used to improve accuracy and bandwidth.

Appendix A: Works Cited

Discussion of binary to BCD conversion. (2011, December 8). [Online]. Available: www.ee.duke.edu/~dwyer/courses/.../**Binary_to_BCD**_Converter.pdf

AD574 datasheet.

6.111 Lab #3 [Online]. Available: web.mit.edu/6.111/www/f2011/

6.111 Tools [Online]. Available: web.mit.edu/6.111/www/f2011/

Appendix B: Verilog Code

```
///////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version)
//
///////////////////////////////////////////////////////////////////////////////

module debounce (input reset, clock, noisy,
                 output reg clean);

  reg [19:0] count;
  reg new;

  always @(posedge clock)
    if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
    else if (noisy != new) begin new <= noisy; count <= 0; end
    else if (count == 650000) clean <= new;
    else count <= count+1;

endmodule
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 28-Nov-06 CJT: fixed race condition between CE and RS (thanks Javier!)
//
// This verilog module drives the labkit hex dot matrix displays, and puts
// up 16 hexadecimal digits (8 bytes).  These are passed to the module
// through a 64 bit wire ("data"), asynchronously.
//
///////////////////////////////////////////////////////////////////////////////

module display_16hex (reset, clock_27mhz, data,
                      disp_blank, disp_clock, disp_rs, disp_ce_b,
                      disp_reset_b, disp_data_out);

  input reset, clock_27mhz;    // clock and reset (active high reset)
  input [63:0] data;           // 16 hex nibbles to display

  output disp_blank, disp_clock, disp_data_out, disp_rs, disp_ce_b,
         disp_reset_b;
```

```
reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

////////////////////////////////////////////////////////////////////////////
//
// Display Clock
//
// Generate a 500kHz clock for driving the displays.
//
////////////////////////////////////////////////////////////////////////////

reg [4:0] count;
reg [7:0] reset_count;
reg clock;
wire dreset;

always @(posedge clock_27mhz)
  begin
        if (reset)
          begin
            count = 0;
            clock = 0;
          end
        else if (count == 26)
          begin
            clock = ~clock;
            count = 5'h00;
          end
        else
          count = count+1;
  end

always @(posedge clock_27mhz)
  if (reset)
    reset_count <= 100;
  else
    reset_count <= (reset_count==0) ? 0 : reset_count-1;

assign dreset = (reset_count != 0);

assign disp_clock = ~clock;

////////////////////////////////////////////////////////////////////////////
//
// Display State Machine
//
////////////////////////////////////////////////////////////////////////////

reg [7:0] state;                // FSM state
reg [9:0] dot_index;            // index to current dot being clocked out
reg [31:0] control;          // control register
reg [3:0] char_index;        // index of current character
reg [39:0] dots;             // dots for a single digit
reg [3:0] nibble;            // hex nibble of current character

assign disp_blank = 1'b0; // low <= not blanked

always @(posedge clock)
  if (dreset)
    begin
        state <= 0;
        dot_index <= 0;
        control <= 32'h7F7F7F7F;
    end
  else
    casex (state)
      8'h00:
        begin
          // Reset displays
          disp_data_out <= 1'b0;
          disp_rs <= 1'b0; // dot register
          disp_ce_b <= 1'b1;
          disp_reset_b <= 1'b0;
          dot_index <= 0;
          state <= state+1;
        end

      8'h01:
        begin
          // End reset
          disp_reset_b <= 1'b1;
          state <= state+1;
        end

      8'h02:
        begin
          // Initialize dot register (set all dots to zero)
          disp_ce_b <= 1'b0;
          disp_data_out <= 1'b0; // dot_index[0];
          if (dot_index == 639)
              state <= state+1;
          else
              dot_index <= dot_index+1;
        end

      8'h03:
        begin
          // Latch dot data
          disp_ce_b <= 1'b1;
```

```
            dot_index <= 31;              // re-purpose to init ctrl reg
            disp_rs <= 1'b1; // Select the control register
            state <= state+1;
          end

        8'h04:
          begin
            // Setup the control register
            disp_ce_b <= 1'b0;
            disp_data_out <= control[31];
            control <= {control[30:0], 1'b0}; // shift left
            if (dot_index == 0)
                 state <= state+1;
            else
                 dot_index <= dot_index-1;
          end

        8'h05:
          begin
            // Latch the control register data / dot data
            disp_ce_b <= 1'b1;
            dot_index <= 39;              // init for single char
            char_index <= 15;             // start with MS char
            state <= state+1;
            disp_rs <= 1'b0;              // Select the dot register
          end

        8'h06:
          begin
            // Load the user's dot data into the dot reg, char by char
            disp_ce_b <= 1'b0;
            disp_data_out <= dots[dot_index]; // dot data from msb
            if (dot_index == 0)
              if (char_index == 0)
                state <= 5;                          // all done, latch data
                 else
                 begin
                   char_index <= char_index - 1;        // goto next char
                   dot_index <= 39;
                 end
            else
                 dot_index <= dot_index-1;    // else loop thru all dots
          end

    endcase

always @ (data or char_index)
  case (char_index)
    4'h0:                  nibble <= data[3:0];
```

```
      4'h1:              nibble <= data[7:4];
      4'h2:              nibble <= data[11:8];
      4'h3:              nibble <= data[15:12];
      4'h4:              nibble <= data[19:16];
      4'h5:              nibble <= data[23:20];
      4'h6:              nibble <= data[27:24];
      4'h7:              nibble <= data[31:28];
      4'h8:              nibble <= data[35:32];
      4'h9:              nibble <= data[39:36];
      4'hA:              nibble <= data[43:40];
      4'hB:              nibble <= data[47:44];
      4'hC:              nibble <= data[51:48];
      4'hD:              nibble <= data[55:52];
      4'hE:              nibble <= data[59:56];
      4'hF:              nibble <= data[63:60];
    endcase

   always @(nibble)
    case (nibble)
      4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
      4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
      4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
      4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
      4'h4: dots <= 40'b00011000_00010100_00010010_01111111_00010000;
      4'h5: dots <= 40'b00100111_01000101_01000101_01000101_00111001;
      4'h6: dots <= 40'b00111100_01001010_01001001_01001001_00110000;
      4'h7: dots <= 40'b00000001_01110001_00001001_00000101_00000011;
      4'h8: dots <= 40'b00110110_01001001_01001001_01001001_00110110;
      4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
      4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
      4'hB: dots <= 40'b01111111_01001001_01001001_01001001_00110110;
      4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
      4'hD: dots <= 40'b01111111_01000001_01000001_01000001_00111110;
      4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
      4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
    endcase

endmodule

///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
```

```
///////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
```

```
///////////////////////////////////////////////////////////////////////////

module FPGA_scope (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
            ac97_bit_clock,

            vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
            vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
            vga_out_vsync,

            tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
            tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
            tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

            tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
            tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
            tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
            tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

            ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
            ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

            ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
            ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

            clock_feedback_out, clock_feedback_in,

            flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
            flash_reset_b, flash_sts, flash_byte_b,

            rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

            mouse_clock, mouse_data, keyboard_clock, keyboard_data,

            clock_27mhz, clock1, clock2,

            disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
            disp_reset_b, disp_data_in,

            button0, button1, button2, button3, button_enter, button_right,
            button_left, button_down, button_up,

            switch,

            led,

            user1, user2, user3, user4,

            daughtercard,
               systemace_data, systemace_address, systemace_ce_b,
               systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

               analyzer1_data, analyzer1_clock,
               analyzer2_data, analyzer2_clock,
               analyzer3_data, analyzer3_clock,
               analyzer4_data, analyzer4_clock);

output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
input  ac97_bit_clock, ac97_sdata_in;

output [7:0] vga_out_red, vga_out_green, vga_out_blue;
output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
       vga_out_hsync, vga_out_vsync;

output [9:0] tv_out_ycrcb;
output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
       tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
       tv_out_subcar_reset;

input  [19:0] tv_in_ycrcb;
input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
       tv_in_hff, tv_in_aff;
output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
       tv_in_reset_b, tv_in_clock;
inout  tv_in_i2c_data;

inout  [35:0] ram0_data;
output [18:0] ram0_address;
output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
output [3:0] ram0_bwe_b;

inout  [35:0] ram1_data;
output [18:0] ram1_address;
output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
output [3:0] ram1_bwe_b;

input  clock_feedback_in;
output clock_feedback_out;

inout  [15:0] flash_data;
output [23:0] flash_address;
output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
input  flash_sts;

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;
input  clock_27mhz, clock1, clock2;

output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
input  disp_data_in;
output  disp_data_out;

input  button0, button1, button2, button3, button_enter, button_right,
        button_left, button_down, button_up;
input  [7:0] switch;
output [7:0] led;

inout [31:0] user1, user2, user3, user4;

inout [43:0] daughtercard;

inout  [15:0] systemace_data;
output [6:0]  systemace_address;
output systemace_ce_b, systemace_we_b, systemace_oe_b;
input  systemace_irq, systemace_mpbrdy;

output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
                analyzer4_data;
output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

////////////////////////////////////////////////////////////////////////////
//
// I/O Assignments
//
////////////////////////////////////////////////////////////////////////////

// Audio Input and Output
assign beep= 1'b0;
assign audio_reset_b = 1'b0;
assign ac97_synch = 1'b0;
assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// Video Output
assign tv_out_ycrcb = 10'h0;
assign tv_out_reset_b = 1'b0;
assign tv_out_clock = 1'b0;
assign tv_out_i2c_clock = 1'b0;
assign tv_out_i2c_data = 1'b0;
assign tv_out_pal_ntsc = 1'b0;
assign tv_out_hsync_b = 1'b1;
assign tv_out_vsync_b = 1'b1;
assign tv_out_blank_b = 1'b1;
assign tv_out_subcar_reset = 1'b0;
```

```
// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input

// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 24'h0;
assign flash_ce_b = 1'b1;
assign flash_oe_b = 1'b1;
assign flash_we_b = 1'b1;
assign flash_reset_b = 1'b0;
assign flash_byte_b = 1'b1;
// flash_sts is an input

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;
// rs232_rxd and rs232_cts are inputs
// PS/2 Ports
// mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

// LED Displays
/*assign disp_blank = 1'b1;
assign disp_clock = 1'b0;
assign disp_rs = 1'b0;
assign disp_ce_b = 1'b1;
assign disp_reset_b = 1'b0;
assign disp_data_out = 1'b0;*/
// disp_data_in is an input

// Buttons, Switches, and Individual LEDs
//lab3 assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

// User I/Os
assign user1 = 32'hZ;
assign user2 = 32'hZ;
assign user3[30:13] = 18'hZ;
assign user4 = 32'hZ;

// Daughtercard Connectors
assign daughtercard = 44'hZ;

// SystemACE Microprocessor Port
assign systemace_data = 16'hZ;
assign systemace_address = 7'h0;
assign systemace_ce_b = 1'b1;
assign systemace_we_b = 1'b1;
assign systemace_oe_b = 1'b1;
// systemace_irq and systemace_mpbrdy are inputs

// Logic Analyzer
assign analyzer1_data = 16'h0;
assign analyzer1_clock = 1'b1;
assign analyzer2_data = 16'h0;
assign analyzer2_clock = 1'b1;
assign analyzer3_data = 16'h0;
assign analyzer3_clock = 1'b1;
assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;

////////////////////////////////////////////////////////////////////////////
//
// FPGA-Scope, modified from lab3
//
////////////////////////////////////////////////////////////////////////////
```

```
// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

// power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
                    .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
wire reset,user_reset;
debounce db1(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_enter),.clean(user_reset));
assign reset = user_reset | power_on_reset;

// buttons for menu FSM
wire up, down, left, right, button_0, button_1;
debounce db2(.reset(reset),.clock(clock_65mhz),.noisy(~button_up),.clean(up));
debounce db3(.reset(reset),.clock(clock_65mhz),.noisy(~button_down),.clean(down));
debounce db4(.reset(reset),.clock(clock_65mhz),.noisy(~button_left),.clean(left));
debounce db5(.reset(reset),.clock(clock_65mhz),.noisy(~button_right),.clean(right));
debounce db6(.reset(reset),.clock(clock_65mhz),.noisy(~button0),.clean(button_0));
debounce db7(.reset(reset),.clock(clock_65mhz),.noisy(~button1),.clean(button_1));

// generate basic XVGA video signals
wire [10:0] hcount;
wire [9:0]  vcount;
wire hsync,vsync,blank;
        wire [2:0] pixel;
xvga xvga1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
          .hsync(hsync),.vsync(vsync),.blank(blank));

// feed XVGA signals to user's FPGA-scope
wire phsync,pvsync,pblank;
        wire [19:0] delta_t_bin;
        wire [13:0] delta_v_bin;
        wire [23:0] delta_t_dec;
        wire [15:0] delta_v_dec;
        wire [1:0] state;
        wire [2:0] button;
        wire [11:0] dout;
        wire [9:0] addr_req;
        wire [11:0] addra;
        wire [11:0] pk_pk;
        wire [11:0] avg;
        wire [13:0] freq;
        wire [11:0] trigger;
        display_module dm(.vclock(clock_65mhz),.reset(reset),.hcount(hcount),
                    .vcount(vcount),.hsync(hsync),.vsync(vsync),.blank(blank),
                                                  .delta_v_bin(delta_v_bin),.dout(dout),.fill(switch[7]
),.phsync(phsync),
                                                  .pvsync(pvsync),.pblank(pblank),.pixel(pixel),.addr
a(addr_req));

        assign addra = addr_req+trigger-374;

  menu_fsm mfsm(.clk(clock_65mhz),.button_0(button_0),.button_1(button_1),
                  .button_up(up),.button_down(down),.button_left(left),

.button_right(right),.button_enter(user_reset),.state_out(state),.button_out(button),
                                        .delta_t_bin(delta_t_bin),.delta_v_bin(delta_v_bin),
                                        .delta_t_dec(delta_t_dec),.delta_v_dec(delta_v_dec));
  //{delta_t_dec,8'b0,delta_v_dec,16'b0}
        wire [19:0] quotient;
        div divtest(.clk(clock_65mhz),.dividend(delta_t_bin),.divisor(delta_v_bin),.quotient(quotient));
  //display_16hex
d1(.reset(reset),.clock_27mhz(clock_27mhz),.data({delta_t_dec,8'b0,delta_v_dec,11'b0,button,state}),
        display_16hex
d1(.reset(reset),.clock_27mhz(clock_27mhz),.data({delta_t_dec,8'b0,delta_v_dec,4'b0,dout[11:0]}),
              .disp_blank(disp_blank),.disp_clock(disp_clock),.disp_rs(disp_rs),.disp_ce_b(disp_ce_b),
                     .disp_reset_b(disp_reset_b),.disp_data_out(disp_data_out));

        parameter OFFSET_X = 11'd799;//x offset of display data (was 787)
        parameter OFFSET_Y = 10'd204;//y offset of display data
        parameter LS = 10'd30;//line spacing between label and number
        parameter DS = 10'd80;//spacing between display data
  // display delta-t
  wire [16*8-1:0] cs_delta_t;
        assign cs_delta_t[16*8-1:6*8] = "uSec/Div: ";
        bcd2display b2d_delta_t(.bcd(delta_t_dec),.display(cs_delta_t[47:0]));
  wire [2:0]  cdp_delta_t;
  char_string_display cd_delta_t(.vclock(clock_65mhz),.hcount(hcount),
                          .vcount(vcount),.pixel(cdp_delta_t),

.cstring(cs_delta_t),.cx(11'd313),.cy(10'd739));//was 408
  defparam    cd_delta_t.NCHAR = 16;
  defparam    cd_delta_t.NCHAR_BITS = 4;

  // display delta-v label
  wire [11*8-1:0] cs_delta_v_label = "mVolts/Div:";
  wire [2:0]  cdp_delta_v_label;
```

```
   char_string_display cd_delta_v_label(.vclock(clock_65mhz),.hcount(hcount),
                                          .vcount(vcount),.pixel(cdp_delta_v_label),

.cstring(cs_delta_v_label),.cx(OFFSET_X),

.cy(OFFSET_Y));
   defparam    cd_delta_v_label.NCHAR = 11;
   defparam    cd_delta_v_label.NCHAR_BITS = 4;

         // display delta-v digits
   wire [6*8-1:0] cs_delta_v;
         bcd2display b2d_delta_v(.bcd({8'b0,delta_v_dec}),.display(cs_delta_v));
   wire [2:0]  cdp_delta_v;
   char_string_display cd_delta_v(.vclock(clock_65mhz),.hcount(hcount),
                                    .vcount(vcount),.pixel(cdp_delta_v),

.cstring(cs_delta_v),.cx(OFFSET_X),

.cy(OFFSET_Y+LS));
   defparam    cd_delta_v.NCHAR = 6;
   defparam    cd_delta_v.NCHAR_BITS = 3;

         // display vpp label
   wire [13*8-1:0] cs_vpp_label = "mVolts Pk-Pk:";
   wire [2:0]  cdp_vpp_label;
   char_string_display cd_vpp_label(.vclock(clock_65mhz),.hcount(hcount),
                                     .vcount(vcount),.pixel(cdp_vpp_label),
                                                                              .cstring(cs_vp
p_label),.cx(OFFSET_X),
                                                                              .cy(OFFSET_
Y+LS+DS));
   defparam    cd_vpp_label.NCHAR = 13;
   defparam    cd_vpp_label.NCHAR_BITS = 4;

         // display vpp digits
   wire [6*8-1:0] cs_vpp;
         wire [16:0] b2b_vpp_o;
         wire [21:0] vpp;
         assign vpp = pk_pk*625;
         bin2bcd b2b_vpp(.bin(vpp[21:8]),.bcd(b2b_vpp_o));
         bcd2display b2d_vpp(.bcd({7'b0,b2b_vpp_o}),.display(cs_vpp));
   wire [2:0]  cdp_vpp;
   char_string_display cd_vpp(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
                                .pixel(cdp_vpp),.cstring(cs_vpp),.cx(OFFSET_X),
                                                                   .cy(OFFSET_Y+2*LS+DS));
   defparam    cd_vpp.NCHAR = 6;
   defparam    cd_vpp.NCHAR_BITS = 3;

         // display vm label
   wire [12*8-1:0] cs_vm_label = "mVolts Mean:";
   wire [2:0]  cdp_vm_label;
   char_string_display cd_vm_label(.vclock(clock_65mhz),.hcount(hcount),
                                    .vcount(vcount),.pixel(cdp_vm_label),

.cstring(cs_vm_label),.cx(OFFSET_X),

.cy(OFFSET_Y+2*LS+2*DS));
   defparam    cd_vm_label.NCHAR = 12;
   defparam    cd_vm_label.NCHAR_BITS = 4;

         // display vm digits
   wire [6*8-1:0] cs_vm;
   wire [16:0] b2b_vm_o;
         wire [21:0] vm_scaled;
         assign vm_scaled = avg*625;
         wire [13:0] vm;
         assign vm = (vm_scaled[21:8]>14'd5000)?(vm_scaled[21:8]-14'd5000):(14'd5000-
vm_scaled[21:8]);
   bin2bcd b2b_vm(.bin(vm),.bcd(b2b_vm_o));
   bcd2display b2d_vm(.bcd({7'b0,b2b_vm_o}),.display(cs_vm));
   wire [2:0] cdp_vm;
         wire [2:0] cdp_neg;
         wire [10:0] neg_space;
         assign neg_space = (vm_scaled[21:8]>14'd5000)?11'b0:11'd16;
   char_string_display cd_vm(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
                              .pixel(cdp_vm),.cstring(cs_vm),.cx(OFFSET_X+neg_space),
                                                                  .cy(OFFSET_Y+3*LS+2*DS));
   defparam    cd_vm.NCHAR = 6;
   defparam    cd_vm.NCHAR_BITS = 3;
         wire [7:0] cs_neg;
         assign cs_neg = (vm_scaled[21:8]>14'd5000)?8'b0:"-";
         char_string_display cd_neg(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
                              .pixel(cdp_neg),.cstring(cs_neg),.cx(OFFSET_X),
                                                                  .cy(OFFSET_Y+3*LS+2*DS));
         defparam cd_neg.NCHAR = 1;
   defparam cd_neg.NCHAR_BITS = 1;

         // display freq label
   wire [10*8-1:0] cs_freq_label = "Freq (hZ):";
   wire [2:0]  cdp_freq_label;
   char_string_display cd_freq_label(.vclock(clock_65mhz),.hcount(hcount),
                                     .vcount(vcount),.pixel(cdp_freq_label),

.cstring(cs_freq_label),.cx(OFFSET_X),

.cy(OFFSET_Y+3*LS+3*DS));
   defparam    cd_freq_label.NCHAR = 10;
   defparam    cd_freq_label.NCHAR_BITS = 4;
```

```
        // display freq digits
  wire [6*8-1:0] cs_freq;
        wire [16:0] b2b_freq_o;
  bin2bcd b2b_freq(.bin(freq),.bcd(b2b_freq_o));
  bcd2display b2d_freq(.bcd({7'b0,b2b_freq_o}),.display(cs_freq));
  wire [2:0]  cdp_freq;
  char_string_display cd_freq(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
                        .pixel(cdp_freq),.cstring(cs_freq),.cx(OFFSET_X),

.cy(OFFSET_Y+4*LS+3*DS));
  defparam    cd_freq.NCHAR = 6;
  defparam    cd_freq.NCHAR_BITS = 3;

  // switch[0] selects which video generator to use:
  //  00: text
  //  01: 1 pixel outline of active video area (adjust screen controls)
  //  10: color bars
  reg [2:0] rgb;
  reg b,hs,vs;
  wire [2:0] or1,or2,or3,or4,or5,or6,or7,or8,or9,final_pixel;
        assign or1 = cdp_delta_t | cdp_delta_v_label;
        assign or2 = cdp_delta_v | cdp_vpp_label;
        assign or3 = cdp_vpp | cdp_vm_label;
        assign or4 = cdp_vm | cdp_freq_label;
        assign or5 = cdp_freq | pixel;
        assign or6 = or5 | cdp_neg;
        assign or7 = or1 | or2;
        assign or8 = or3 | or4;
        assign or9 = or6 | or7;
        assign final_pixel = or8 | or9;
  always @(posedge clock_65mhz) begin
    hs <= hsync;
    vs <= vsync;
    b <= blank;
    if (switch[1:0] == 2'b01) begin
        // 1 pixel outline of visible area (white)
        rgb <= (hcount==0 | hcount==1023 | vcount==0 | vcount==767) ? 7 : 0;
    end else if (switch[1:0] == 2'b10) begin
        // color bars
        rgb <= hcount[8:6];
    end else begin
      // default: text
        rgb <= final_pixel;
    end
  end

  // VGA Output.  In order to meet the setup and hold times of the
  // AD7125, we send it ~clock_65mhz.
```

```
   assign vga_out_red = {8{rgb[2]}};
   assign vga_out_green = {8{rgb[1]}};
   assign vga_out_blue = {8{rgb[0]}};
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_blank_b = ~b;
   assign vga_out_pixel_clock = ~clock_65mhz;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;

	//Integration with Anartya's modules


	reg [11:0] data_in;
	wire read_convert;
	wire full;
	assign user3[12] = read_convert;

	assign refill = reset; /// CHANGE THIS

	ADC_Controller adc_ctl(.clk(clock_65mhz),.delta_t(delta_t_bin),
			.read_convert(read_convert));
   Samples_BRAM samples_bram(.status(user3[31]),.clk(clock_65mhz),
				.addr_r(addra),.data_in(user3[11:0]),
									.mem_out(dout),.full(full));

	Math
math(.clk(clock_65mhz),.full(full),.data_in(user3[11:0]),.delta_t(delta_t_bin),.status(user3[31]),
		.pk_pk(pk_pk),.avg(avg),.freq(freq),.trigger(trigger));

	assign led = ~{3'b00,button,state};
endmodule

////////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
//
////////////////////////////////////////////////////////////////////////////////

module xvga(input vclock,
	output reg [10:0] hcount,    // pixel number on current line
	output reg [9:0] vcount,	 // line number
	output reg vsync,hsync,blank);

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount == 1023);
   assign hsyncon = (hcount == 1047);
  assign hsyncoff = (hcount == 1183);
  assign hreset = (hcount == 1343);

  // vertical: 806 lines total
  // display 768 lines
  wire vsyncon,vsyncoff,vreset,vblankon;
  assign vblankon = hreset & (vcount == 767);
  assign vsyncon = hreset & (vcount == 776);
  assign vsyncoff = hreset & (vcount == 782);
  assign vreset = hreset & (vcount == 805);

  // sync and blanking
  wire next_hblank,next_vblank;
  assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
  assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
  always @(posedge vclock) begin
    hcount <= hreset ? 0 : hcount + 1;
    hblank <= next_hblank;
    hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

    vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
    vblank <= next_vblank;
    vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

    blank <= next_vblank | (next_hblank & ~hreset);
  end
endmodule
```

```
`timescale 1ns / 1ps
module menu_fsm(//menu for setting delta-t and delta-v
  input clk,//65mHz
  input button_0,//button inputs
  input button_1,
  input button_up,
  input button_down,
  input button_left,
  input button_right,
  input button_enter,
  output [1:0] state_out,//state out for display
  output [2:0] button_out,//button out for display
  output [20:0] delta_t_bin,//user selected delta_t
  output [13:0] delta_v_bin,//user selected delta_v
  output reg [23:0] delta_t_dec = 24'h10000,
  output reg [15:0] delta_v_dec = 16'h2000
  );
  reg [1:0] next_state = 0;
  reg [1:0] state = 0;
  reg [23:0] next_delta_t_dec = 0;
  reg [15:0] next_delta_v_dec = 0;
  parameter SELECT = 2'b00;
  parameter DELTA_T = 2'b01;
  parameter DELTA_V = 2'b10;
  reg [2:0] next_button = 3'b0;
  reg [2:0] button = 3'b0;
  parameter NONE = 3'b000;
  parameter UP = 3'b001;
  parameter DOWN = 3'b010;
  parameter LEFT = 3'b011;
  parameter RIGHT = 3'b100;
  parameter ENTER = 3'b101;
  reg [2:0] next_index = 0;
  reg [2:0] index = 0;
  always @ * begin // use blocking assignment for always @ *
    case (state)
      SELECT: begin//press "0" or "1" to change delta-t or delta-v
                    next_state = button_0?DELTA_T:(button_1?DELTA_V:SELECT);
                    next_index = 0;
                    next_delta_t_dec = delta_t_dec;
                    next_delta_v_dec = delta_v_dec;
                    end
      DELTA_T: begin//change delta-t
                    if (next_button==NONE) begin
                      next_index = (button==LEFT)?((index<5)?(index+1):index):
                                   ((button==RIGHT)?((index>0)?(index-1):index):index);
        next_delta_t_dec = (button==UP)?(delta_t_dec+
((delta_t_dec[index*4+:4]<4'b1001)<<(index*4))):
                                   ((button==DOWN)?(delta_t_dec-
((delta_t_dec[index*4+:4]>0)<<(index*4))):delta_t_dec);
        next_state = (button==ENTER)?SELECT:DELTA_T;
                            next_delta_v_dec = delta_v_dec;
                    end else begin
                      next_index = index;
                            next_delta_t_dec = delta_t_dec;
                      next_state = DELTA_T;
                            next_delta_v_dec = delta_v_dec;
                    end
                    end
    DELTA_V: begin//change delta-v
                    if (next_button==NONE) begin
                      next_index = (button==LEFT)?((index<3)?(index+1):index):
                                   ((button==RIGHT)?((index>0)?(index-1):index):index);
        next_delta_v_dec = (button==UP)?(delta_v_dec+
((delta_v_dec[index*4+:4]<4'b1001)<<(index*4))):
                                   ((button==DOWN)?(delta_v_dec-
((delta_v_dec[index*4+:4]>0)<<(index*4))):delta_v_dec);
        next_state = (button==ENTER)?SELECT:DELTA_V;
                            next_delta_t_dec = delta_t_dec;
                    end else begin
                      next_index = index;
                            next_delta_v_dec = delta_v_dec;
                      next_state = DELTA_V;
                            next_delta_t_dec = delta_t_dec;
                    end
                    end
    default: begin
                    next_index = index;
                    next_state = SELECT;
                    next_delta_t_dec = delta_t_dec;
                    next_delta_v_dec = delta_v_dec;
                    end
  endcase
        //mini-state machine to determine the last button pressed
  if (button_enter) next_button = ENTER;
        else if (button_up) next_button = UP;
        else if (button_down) next_button = DOWN;
        else if (button_left) next_button = LEFT;
        else if (button_right) next_button = RIGHT;
        else next_button = NONE;
 end
 always @ (posedge clk) begin//update state machines
  state <= next_state;
  delta_t_dec <= next_delta_t_dec;
  delta_v_dec <= next_delta_v_dec;
  button <= next_button;
  index <= next_index;
 end
```

```
  assign state_out = state;
  assign button_out = button;
  assign delta_t_bin = 100000*delta_t_dec[23:20]+10000*delta_t_dec[19:16]+
                       1000*delta_t_dec[15:12]+100*delta_t_dec[11:8]+
                                                              10*delta_t_dec[7:4]+delta_t_dec[3:0];
  assign delta_v_bin = 1000*delta_v_dec[15:12]+100*delta_v_dec[11:8]+
                                                              10*delta_v_dec[7:4]+delta_v_dec[3:0];
endmodule
```

```
`timescale 1ns / 1ps

module bin2dcb_test;

	// Inputs
	reg [13:0] bin;

	// Outputs
	wire [16:0] dcb;

	// Instantiate the Unit Under Test (UUT)
	bin2dcb uut (
		.bin(bin),
		.dcb(dcb)
	);

	initial begin
		// Initialize Inputs
		bin = 0;

		// Wait 100 ns for global reset to finish
		#100;
     $display("0 = %h", $dcb);
		bin = 14'h3FFF;
		#100;
		$display("3FFF = %h", $dcb);



	end

endmodule
```

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   22:05:44 11/20/2011
// Design Name:   menu_fsm
// Module Name:   /afs/athena.mit.edu/user/k/l/klinke/Desktop/FPGA-Scope/Menu_FSM_Test.v
// Project Name:  FPGA-Scope
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: menu_fsm
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module Menu_FSM_Test;

	// Inputs
	reg clk;
	reg button_0;
	reg button_1;
	reg button_up;
	reg button_down;
	reg button_left;
	reg button_right;
	reg button_enter;

	// Outputs
	wire [1:0] state_out;
	wire [2:0] button_out;
	wire [15:0] delta_t_bin;
	wire [9:0] delta_v_bin;
	wire [23:0] delta_t_dec;
	wire [15:0] delta_v_dec;

	// Instantiate the Unit Under Test (UUT)
	menu_fsm uut (
		.clk(clk),
		.button_0(button_0),
		.button_1(button_1),
		.button_up(button_up),
		.button_down(button_down),
		.button_left(button_left),
		.button_right(button_right),
		.button_enter(button_enter),
		.state_out(state_out),
		.button_out(button_out),
		.delta_t_bin(delta_t_bin),
		.delta_v_bin(delta_v_bin),
		.delta_t_dec(delta_t_dec),
		.delta_v_dec(delta_v_dec)
	);

always #5 clk=!clk;

	initial begin
		// Initialize Inputs
		clk = 0;
		button_0 = 0;
		button_1 = 0;
		button_up = 0;
		button_down = 0;
		button_left = 0;
		button_right = 0;
		button_enter = 0;

		// Wait 100 ns for global reset to finish
		#1000;

		// Add stimulus here
  button_0 = 1;
		#2000;
		button_0 = 0;
		#2000;

		button_up = 1;
		#2000;
		button_up = 0;
		#2000;
		button_up = 1;
		#2000;
		button_up = 0;
		#2000;
		button_up = 1;
		#2000;
		button_up = 0;
		#2000;

		button_left = 1;
```

```
#2000;
button_left = 0;
#2000;

button_up = 1;
#2000;
button_up = 0;
#2000;
button_up = 1;
#2000;
button_up = 0;
#2000;
button_up = 1;
#2000;
button_up = 0;
#2000;

button_enter = 1;
#2000;
button_enter = 0;
#2000;

button_1 = 1;
#2000;
button_1 = 0;
#2000;

button_up = 1;
#2000;
button_up = 0;
#2000;
button_up = 1;
#2000;
button_up = 0;
#2000;
button_up = 1;
#2000;
button_up = 0;
#2000;

button_left = 1;
#2000;
button_left = 0;
#2000;

button_up = 1;
#2000;
button_up = 0;
#2000;
```

```
				button_up = 1;
				#2000;
				button_up = 0;
				#2000;
				button_up = 1;
				#2000;
				button_up = 0;
				#2000;
		end

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// Display Module- combines the contents of all the BRAM for the final display image
//
//////////////////////////////////////////////////////////////////////////////////

module display_module (
   input vclock,           // 65MHz clock
   input reset,            // 1 to initialize module
   input [10:0] hcount,// horizontal index of current pixel (0..1023)
   input [9:0]    vcount, // vertical index of current pixel (0..767)
   input hsync,            // XVGA horizontal sync signal (active low)
   input vsync,            // XVGA vertical sync signal (active low)
   input blank,            // XVGA blanking (1 means output black pixel)
         input [13:0] delta_v_bin, //vertical scale
         input [11:0] dout, // ADC data from BRAM
         input fill,      //whether to fill underneath the signal

   output phsync,          // FPGA-scope's horizontal sync
   output pvsync,          // FPGA-scope's vertical sync
   output pblank,          // FPGA-scope's blanking
   output [2:0] pixel,     // FPGA-scope's pixel
         output [9:0] addra      // read address
   );

   // REPLACE ME! The code below just generates a color checkerboard
   // using 64 pixel by 64 pixel squares.
   assign phsync = hsync;
   assign pvsync = vsync;
   assign pblank = blank;

         wire pixel_input;
         parameter INDENT = 34;//left border on screen
         parameter DELAY = 23;//number of clock cycles required for division
   parameter V_PPD = 70;//vertical pixels per division
   parameter H_PPD = 75;//horizontal pixels per division
         //map hcount to an address
         assign addra = (hcount[9:0]>INDENT-DELAY)?(hcount[9:0]-(INDENT-DELAY)):10'b0;
         //old test code
         //mem_test test(.addra(addra),.clka(vclock),.wea(1'b0),.dina(12'b0),.douta(dout));

         wire gridline_input;//create background gridlines
   assign gridline_input = ((((hcount==INDENT)|(hcount==INDENT+H_PPD)|
                             (hcount==INDENT+2*H_PPD)|(hcount==INDENT+3*H_PPD)|
                                                                     (hcount==INDENT+4*H_PPD)|
(hcount==INDENT+5*H_PPD)|
                                                                     (hcount==INDENT+6*H_PPD)|
(hcount==INDENT+7*H_PPD)|
                                                                        (hcount==INDENT+8*H_PPD)|
(hcount==INDENT+9*H_PPD)|

(hcount==INDENT+10*H_PPD))&(vcount[2:0]==3'b0))|
                                                                        (((vcount==INDENT)|
(vcount==INDENT+V_PPD)|
                        (vcount==INDENT+2*V_PPD)|(vcount==INDENT+3*V_PPD)|
                                                                        (vcount==INDENT+4*V_PPD)|
(vcount==INDENT+5*V_PPD)|
                                                                        (vcount==INDENT+6*V_PPD)|
(vcount==INDENT+7*V_PPD)|
                                                                        (vcount==INDENT+8*V_PPD)|
(vcount==INDENT+9*V_PPD)|

(vcount==INDENT+10*V_PPD))&(hcount[2:0]==3'b0)))&

(hcount>=INDENT)&(hcount<=750+INDENT)&

(vcount>=INDENT)&(vcount<=700+INDENT);
        wire [19:0] mult_dout;
        wire [19:0] scaled_dout;
        wire [11:0] flipped_dout;
        assign flipped_dout = ~dout;
        assign mult_dout = flipped_dout*171;
        wire [19:0] center;
        assign center = V_PPD*5000;
        wire [19:0] v_offset;
        div div_offset(.clk(vclock),.dividend(center),.divisor(delta_v_bin),
                       .quotient(v_offset));
        div div_scale(.clk(vclock),.dividend(mult_dout),.divisor(delta_v_bin),
                       .quotient(scaled_dout));
        wire [9:0] shifted_data;
        assign shifted_data = scaled_dout[9:0]+350+INDENT-v_offset[9:0];
        wire compare;
        //select whether to fill underneath the waveform
        assign compare = fill?(shifted_data<=vcount):(shifted_data==vcount);
  assign pixel_input = (compare&(hcount>=INDENT)&(hcount<INDENT+748)&(vcount>=INDENT)
                                                        &(vcount<INDENT+700))^gridline_input;
  assign pixel = {pixel_input,pixel_input,pixel_input};

endmodule
```

```
`timescale 1ns / 1ps
module menu_fsm(//menu for setting delta-t and delta-v
  input clk,//65mHz
  input button_0,//button inputs
  input button_1,
  input button_up,
  input button_down,
  input button_left,
  input button_right,
  input button_enter,
  output [1:0] state_out,//state out for display
  output [2:0] button_out,//button out for display
  output [20:0] delta_t_bin,//user selected delta_t
  output [13:0] delta_v_bin,//user selected delta_v
  output reg [23:0] delta_t_dec = 24'h10000,
  output reg [15:0] delta_v_dec = 16'h2000
  );
  reg [1:0] next_state = 0;
  reg [1:0] state = 0;
  reg [23:0] next_delta_t_dec = 0;
  reg [15:0] next_delta_v_dec = 0;
  parameter SELECT = 2'b00;
  parameter DELTA_T = 2'b01;
  parameter DELTA_V = 2'b10;
  reg [2:0] next_button = 3'b0;
  reg [2:0] button = 3'b0;
  parameter NONE = 3'b000;
  parameter UP = 3'b001;
  parameter DOWN = 3'b010;
  parameter LEFT = 3'b011;
  parameter RIGHT = 3'b100;
  parameter ENTER = 3'b101;
  reg [2:0] next_index = 0;
  reg [2:0] index = 0;
  always @ * begin // use blocking assignment for always @ *
    case (state)
      SELECT: begin//press "0" or "1" to change delta-t or delta-v
                    next_state = button_0?DELTA_T:(button_1?DELTA_V:SELECT);
                    next_index = 0;
                    next_delta_t_dec = delta_t_dec;
                    next_delta_v_dec = delta_v_dec;
                    end
      DELTA_T: begin//change delta-t
                    if (next_button==NONE) begin
                      next_index = (button==LEFT)?((index<5)?(index+1):index):
                                   ((button==RIGHT)?((index>0)?(index-1):index):index);
        next_delta_t_dec = (button==UP)?(delta_t_dec+
((delta_t_dec[index*4+:4]<4'b1001)<<(index*4))):
                                   ((button==DOWN)?(delta_t_dec-
((delta_t_dec[index*4+:4]>0)<<(index*4))):delta_t_dec);
        next_state = (button==ENTER)?SELECT:DELTA_T;
                            next_delta_v_dec = delta_v_dec;
                    end else begin
                      next_index = index;
                            next_delta_t_dec = delta_t_dec;
                      next_state = DELTA_T;
                            next_delta_v_dec = delta_v_dec;
                    end
                    end
      DELTA_V: begin//change delta-v
                    if (next_button==NONE) begin
                      next_index = (button==LEFT)?((index<3)?(index+1):index):
                                   ((button==RIGHT)?((index>0)?(index-1):index):index);
        next_delta_v_dec = (button==UP)?(delta_v_dec+
((delta_v_dec[index*4+:4]<4'b1001)<<(index*4))):
                                   ((button==DOWN)?(delta_v_dec-
((delta_v_dec[index*4+:4]>0)<<(index*4))):delta_v_dec);
        next_state = (button==ENTER)?SELECT:DELTA_V;
                            next_delta_t_dec = delta_t_dec;
                    end else begin
                      next_index = index;
                            next_delta_v_dec = delta_v_dec;
                      next_state = DELTA_V;
                            next_delta_t_dec = delta_t_dec;
                    end
                    end
      default: begin
                    next_index = index;
                    next_state = SELECT;
                    next_delta_t_dec = delta_t_dec;
                    next_delta_v_dec = delta_v_dec;
                    end
    endcase
           //mini-state machine to determine the last button pressed
    if (button_enter) next_button = ENTER;
           else if (button_up) next_button = UP;
           else if (button_down) next_button = DOWN;
           else if (button_left) next_button = LEFT;
           else if (button_right) next_button = RIGHT;
           else next_button = NONE;
  end
  always @ (posedge clk) begin//update state machines
    state <= next_state;
    delta_t_dec <= next_delta_t_dec;
    delta_v_dec <= next_delta_v_dec;
    button <= next_button;
    index <= next_index;
  end
  assign state_out = state;
  assign button_out = button;
  assign delta_t_bin = 100000*delta_t_dec[23:20]+10000*delta_t_dec[19:16]+
                        1000*delta_t_dec[15:12]+100*delta_t_dec[11:8]+
                                                                          10*delta_t_dec[7:4]+delta_t_dec[3:0];
  assign delta_v_bin = 1000*delta_v_dec[15:12]+100*delta_v_dec[11:8]+
                                                                          10*delta_v_dec[7:4]+delta_v_dec[3:0];
endmodule
```

```
`timescale 1ns / 1ps

module bcd2display(//converts decimals to well-formatted strings
   input [23:0] bcd,//bcd input
   output [47:0] display//formatted string output
   );
        wire [2:0] digit_offset_t;
        assign digit_offset_t = (bcd>20'h99999)?0:(bcd>20'h9999)?1:(bcd>20'h999)?2:
                                (bcd>20'h99)?3:(bcd>20'h9)?4:5;
        assign display = {4'h3,bcd[23:20],4'h3,bcd[19:16],
                            4'h3,bcd[15:12],4'h3,bcd[11:8],
                                                            4'h3,bcd[7:4],4'h3,bcd[3:0]}<<digit_offset_t*8;
endmodule
```

```
`timescale 1ns / 1ps
module bin2bcd(//converts binary numbers to BCD
   input [13:0] bin,//binary number to convert
   output [16:0] bcd//converted output
   );
   wire [3:0] r1,r2,r3,
              r4a,r4b,r5a,r5b,r6a,r6b,
              r7a,r7b,r7c,r8a,r8b,r8c,r9a,r9b,r9c,
              r10a,r10b,r10c,r10d;
         assign bcd[0] = bin [0];
         //large array of small adding modules to simulate shifting bits
   m m1({0,bin[13:11]},r1);
   m m2({r1[2:0],bin[10]},r2);
   m m3({r2[2:0],bin[9]},r3);
   m m4b({r3[2:0],bin[8]},r4b);
   m m5b({r4b[2:0],bin[7]},r5b);
   m m6b({r5b[2:0],bin[6]},r6b);
   m m7c({r6b[2:0],bin[5]},r7c);
   m m8c({r7c[2:0],bin[4]},r8c);
   m m9c({r8c[2:0],bin[3]},r9c);
   m m10d({r9c[2:0],bin[2]},r10d);
   m m11d({r10d[2:0],bin[1]},bcd[4:1]);

         m m4a({0,r1[3],r2[3],r3[3]},r4a);
         m m5a({r4a[2:0],r4b[3]},r5a);
         m m6a({r5a[2:0],r5b[3]},r6a);
         m m7b({r6a[2:0],r6b[3]},r7b);
         m m8b({r7b[2:0],r7c[3]},r8b);
         m m9b({r8b[2:0],r8c[3]},r9b);
         m m10c({r9b[2:0],r9c[3]},r10c);
         m m11c({r10c[2:0],r10d[3]},bcd[8:5]);

         m m7a({0,r4a[3],r5a[3],r6a[3]},r7a);
         m m8a({r7a[2:0],r7b[3]},r8a);
         m m9a({r8a[2:0],r8b[3]},r9a);
         m m10b({r9a[2:0],r9b[3]},r10b);
         m m11b({r10b[2:0],r10c[3]},bcd[12:9]);

         m m10a({0,r7a[3],r8a[3],r9a[3]},r10a);
         m m11a({r10a[2:0],r10b[3]},bcd[16:13]);
endmodule

module m(//this module adds 3 to the input if it is greater than or equal to 5
   input [3:0] i,
   output [3:0] o
   );
         assign o = (i>=4'b101)?(i+4'b11):i;
endmodule
```

```
`timescale 1ns / 1ps
module ADC_Controller   (input clk,
                                                                      input [19:0] delta_t,
                                                                      output reg read_convert);

  parameter PULSE_WIDTH = 33;
  reg [32:0] count=1;
  wire [23:0] scaled_delta_t;
  assign scaled_delta_t = delta_t*14;
  wire [19:0] clk_cyc;
  assign clk_cyc = (scaled_delta_t[23:4]>PULSE_WIDTH)?scaled_delta_t[23:4]:20'd34;
  always @ (posedge clk) begin
                if (count >= clk_cyc) count <= 1;
                else count <= count + 1;
                read_convert <= (count > PULSE_WIDTH);
  end

endmodule
```

```
`timescale 1ns / 1ps
module Samples_BRAM #(parameter SAMPLES_WIDTH=12)
                                                  (input status, clk,
                                                   input [11:0] data_in, addr_r,
                                                   output [11:0] mem_out,
                                                   output full
                                                   );

        reg sel = 0;
        wire [11:0] addr1,addr2,mem_out1,mem_out2;
        reg [11:0] addr_w = 0;
        reg status_sm, next_status_sm;
        wire transition;

        // instantiate BRAMs
        assign addr1 = sel?addr_w:addr_r;
        assign addr2 = sel?addr_r:addr_w;
  mybram2 memory1(.addra(addr1),.clka(clk),.wea(sel),.dina(data_in),.douta(mem_out1));
  mybram2 memory2(.addra(addr2),.clka(clk),.wea(!sel),.dina(data_in),.douta(mem_out2));

        parameter ADDR_MAX = 748*3-1;

  always @ (posedge clk) begin
                next_status_sm <= status;
                status_sm <= next_status_sm;
                if (full) addr_w <= 0;
                else if (transition) begin
                        // if we haven't reached max address increment address and write to BRAM
                                if (addr_w != ADDR_MAX) addr_w <= addr_w + 1;// increment address
                                else addr_w <= addr_w;
           end
                if (full) sel <= !sel;
        end

        assign transition = (status_sm == 1) && (next_status_sm == 0);
        assign full = (addr_w == ADDR_MAX);
        assign mem_out = sel?mem_out2:mem_out1;

endmodule
```

```
`timescale 1ns / 1ps
module Math (input status, clk, full,
                                input [11:0] data_in,
                                input [19:0] delta_t,
                                output reg [11:0] pk_pk, avg, trigger,
                                output reg [13:0] freq
                                );

        reg [11:0] max = 0, min = 12'hFFF, gmax = 0;
        reg status_sm, next_status_sm;
        wire transition;
        reg [23:0] sum = 0;
        reg [11:0] count = 0;
        reg [11:0] data_in_delay_1, data_in_delay_2, addr_p1, addr_p2;
        wire [26:0] freq_out;
        wire [11:0] avg_out;

        always @ (posedge clk) begin
                next_status_sm <= status;
                status_sm <= next_status_sm;

                if (full ==1) begin
                        max <= 0;
                        min <= 12'hFFF;
                        gmax <= 0;
                        count <= 0;
                        addr_p1 <= 0;
                        addr_p2 <= 0;
                        sum <= 0;
                        freq <= freq_out[13:0];
            pk_pk <= max - min;
            trigger <= addr_p1;
                        avg <= avg_out;
                end

                if (transition) begin

                        data_in_delay_1 <= data_in;
                        data_in_delay_2 <= data_in_delay_1;

                        if (!full) count <= count + 1;
                        else count <= count;
                        if (max[11:0] <= data_in[11:0]) max <= data_in;
                        else max <= max;

                        if (min > data_in) min <= data_in;
                        else min <= min;

                        if (count == 12'h2EC) gmax <= max;
```

```
                    if (count > 12'h2EC) begin
                                    if (!((addr_p1 != 0) && (addr_p1 != addr_p2))) begin
                                      if (addr_p1 != 0) sum <= sum + data_in;
                                            if ((data_in[11:6] == gmax[11:6]) && (data_in_delay_1[11:6] <
gmax[11:6]))
                                                    addr_p2 <= count;
                                            if ((data_in_delay_1[11:6] == gmax[11:6]) &&
(data_in_delay_2[11:6] < gmax[11:6]))
                                                    addr_p1 <= count-1;
                              end
                    end
                  end
        end

        wire [11:0] period;
        assign period = addr_p2 - addr_p1;
        parameter FREQ_DIVIDEND = 27'd75000000;
        wire [26:0] freq_divisor;
        assign freq_divisor = delta_t*period;
        div_avg div_avg1(.clk(clk),.dividend(sum),.divisor(period),.quotient(avg_out));
        div_freq
div_freq1(.clk(clk),.dividend(FREQ_DIVIDEND),.divisor(freq_divisor),.quotient(freq_out));
        assign transition = (status_sm == 1) && (next_status_sm == 0);

endmodule
```