Phased array pulsing: A simple digital sonar system

Brian Wong, Bryan Morrissey, Zhen Li
{b_wong, blmorris, zhenli} @ mit.edu

14th December 2007

MIT 6.111 Digital Systems Laboratory

Abstract

For this project we designed and implemented a system that processes the information collected from ultrasonic echo detection to construct a graphical representation of its environment. This required the construction of a support structure for a set of ultrasonic transmitters and receivers, as well as circuitry for signal amplification and analog to digital conversion to interface with the labkit FPGA. All signal processing tasks beyond basic analog amplification and filtering were implemented in Verilog for execution on the 6.111 labkit's Xilinx FPGA. These signal processing tasks include pulse generation, echo detection, time interval measurement, coincidence detection, buffering data to memory, constructing an abstract environmental representation, and interfacing with the video hardware to display this representation on a VGA monitor. Preliminary experiments on the ultrasonic devices and interface circuitry provided promising results, encouraging us to push forward with a challenging, intricate and ultimately successful design which was able in demonstrations to simultaneously track the direction and range to two separate moving objects.

Sonar System Overview and Theory of Operation

A sonar system locates objects by transmitting an acoustic pulse (often in frequencies well above the range of human hearing), detecting the reflected echo signal, and processing the signal to derive information about the object that reflected it. The simplest form of sonar is probably the ranging device, which simply measures the time interval between when the pulse is transmitted and when the echo signal is received. Beyond this in complexity are systems which utilize multiple receivers to determine both the range and direction to the object generating the reflection signal. We designed and built this second type of sonar for our 6.111 final project.

In the context of a sonar (or radar) system, a phased array can mean two different things: manipulating the relative phase of the signal on an array of transmitters to control the direction in which the transmitted pulse is sent, and measuring the relative phase of the echo signal detected by the receivers to determine the direction from which the signal is arriving. Sophisticated radar systems will often utilize both techniques, directing their radiated energy to a specific point of interest and processing the return signal to eliminate any extraneous noise that might have been received from sources in other directions.

In contrast, our sonar project focuses on the second of these strategies, transmitting a sonar pulse to a broad forward region in an effort to "illuminate" multiple objects, and processing the returned signal to extract range and distance information for as many objects as possible. This design path allowed us to eliminate much of the potential complexity from the transmit stage and concentrate instead on producing highly capable signal acquisition and data processing components. The resulting system was able to detect, discriminate and track the return signatures from at least two objects; with the further opportunity for fine-tuning we feel confident that the system has the potential to detect and track several more.

SECTION 1: HARDWARE INTERFACE & DATA ACQUISITION – BRYAN MORRISSEY

Hardware Overview

1.1.1 Introduction

My portion of the project consisted of designing and building the external interface hardware as well as the digital processing modules to acquire the echo signal and prepare the data for analysis by the object detection logic. The physical interface of the sonar system includes several subsystems that are implemented in hardware external to the Xilinx FPGA. These systems include the transmit pulse amplifier, the transmitter and receiver arrays, and the analog circuitry to amplify the received signal and interface with the digital control portion of the Delta-Sigma analog to digital converters. (Figure 1.1)

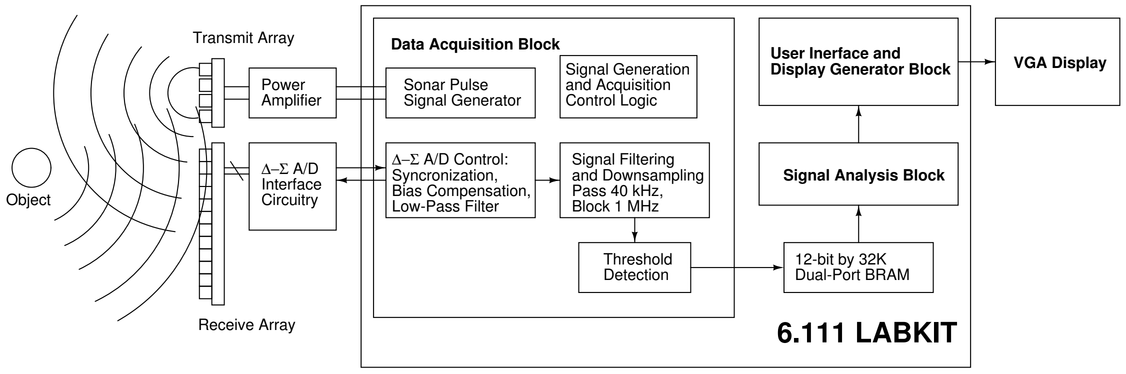

Transmit Array
Object
Receive Array
Power Amplifier
Δ–Σ A/D Interface Circuitry
Data Acquisition Block
Sonar Pulse Signal Generator
Signal Generation and Acquisition Control Logic
Δ–Σ A/D Control: Syncronization, Bias Compensation, Low-Pass Filter
Signal Filtering and Downsampling Pass 40 kHz, Block 1 MHz
Threshold Detection
User Inerface and Display Generator Block
VGA Display
Signal Analysis Block
12-bit by 32K Dual-Port BRAM
6.111 LABKIT

Figure 1.1: Sonar System Overview

1.1.2 Sonar Pulse Generation and Transmission

The ultrasonic transmitter and receiver devices were Kobitone parts 255-400ST16 and 255-400SR16, (datasheet available at [http://www.mouser.com/catalog/specsheets/KT-400244.pdf]) These devices have an operating frequency of 40kHz, well above the range of human hearing. The transmitters are rated to be driven at 20V RMS or higher, at which point they will produce sound pressure levels in excess of 120 dB. The acoustic output is adequate at much lower levels; with a simple bridged op-amp configuration (and a very capable LT1632 op-amp) we built a circuit that could drive 10V RMS into an array of four transmitters in parallel. This was more than adequate for our purposes and had the advantageous feature that it could be constructed on the labkit using only the available +/- 12V power supplies. (Figure 1.2)

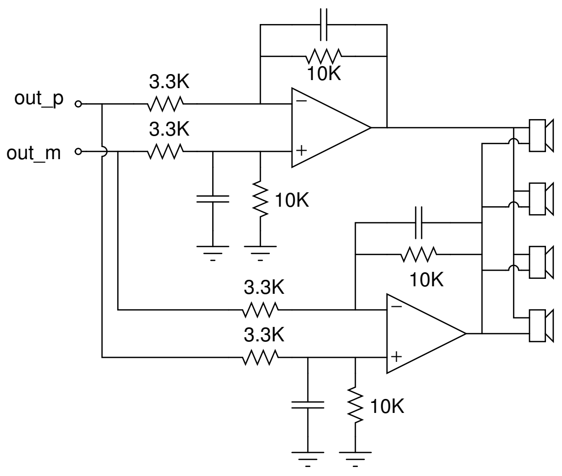


Figure 1.2: Transmitter Amplifier

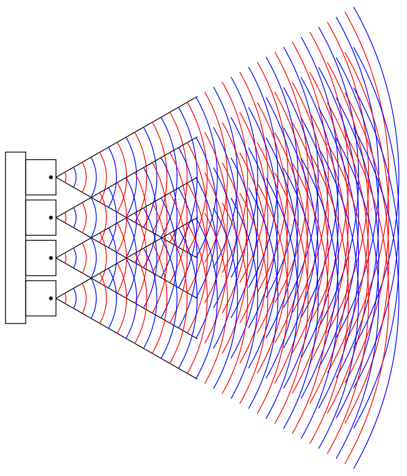


Figure 1.3: Vertical Array Radiation Pattern

Figure 1.3 provides a simplified illustration of the effect that we hoped to achieve by arranging the transmitter array vertically, perpendicular to the horizontal axis of the receiver array. When several transmitters are arranged in a linear array (and driven with the same signal) their sound fields will tend to reinforce each other in coherent wave fronts directly in front of the array, and all along a plane perpendicular to the axis of the array and roughly equidistant from all of the transmitters. In our case, this means that most of the acoustic energy is transmitted into a plane that is parallel to the receiver array, which is precisely the optimal region for our system to determine the direction of a received signal.

1.1.3 Receiver Array and A/D Interface Circuitry

(Note 1: At this point I would like to thank Ron Roscoe for his advice and assistance, primarily for suggesting and providing the op-amps and comparators used in the delta-sigma component of the analog interface. A delta-sigma converter is a challenging application for components assembled on a breadboard, and I might have given up and pursued another approach if there had not been high-performance components readily at hand.)

(Note 2: A full presentation of delta-sigma theory is far beyond the scope of this paper, which is primarily focused on the digital design aspects of a sonar system. The interested reader will find a detailed and thorough introduction to the topic at Wikipedia:
[http://en.wikipedia.org/wiki/Delta-sigma_modulation].

The intersection of the analog and digital regimes always provides interesting design challenges; they are especially apparent in the design of delta-sigma A/D converters. In order for analog and digital circuits to interface directly with each other as they do here, it is necessary for the analog circuitry to generate a digital signal (which it will often do asynchronously) and to design the digital circuitry to produce an output that can be filtered into an effective imitation of an analog input signal.

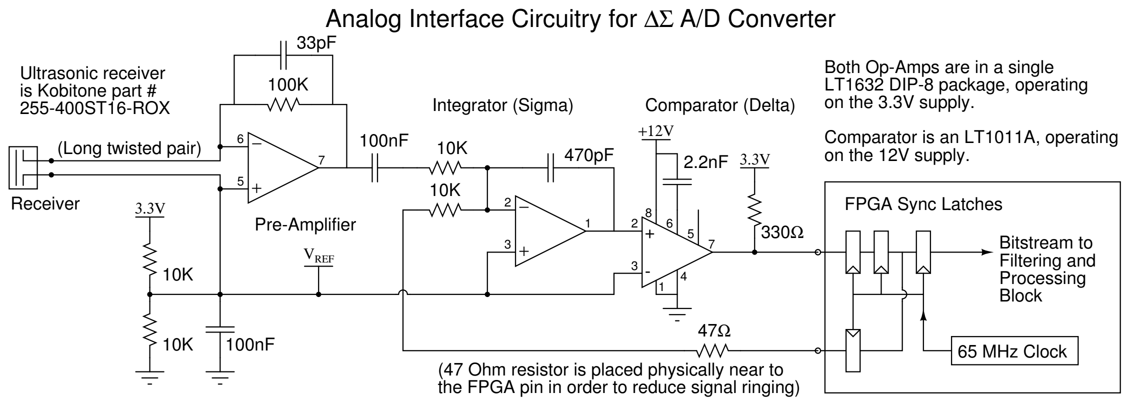

Analog Interface Circuitry for ΔΣ A/D Converter
33pF
100K
Ultrasonic receiver is Kobitone part # 255-400ST16-ROX
(Long twisted pair)
Receiver
Pre-Amplifier
3.3V
10K
10K
100nF
V_REF
100nF
10K
10K
Integrator (Sigma)
470pF
Comparator (Delta)
+12V
2.2nF
3.3V
330Ω
47Ω
(47 Ohm resistor is placed physically near to the FPGA pin in order to reduce signal ringing)
Both Op-Amps are in a single LT1632 DIP-8 package, operating on the 3.3V supply.
Comparator is an LT1011A, operating on the 12V supply.
FPGA Sync Latches
Bitstream to Filtering and Processing Block
65 MHz Clock

Figure 1.4: Delta Sigma Interface Circuitry

The interface circuitry for the analog-to-digital converter consists of twelve identical channels constructed as shown in figure 1.4. (Photographs of the transmit – receive array and the finished circuitry construction are included in the appendix.) The electrical signal from the receiver is amplified through a single op-amp stage (effectively configured as a current-to-voltage converter or trans-impedance amplifier). The amplified signal is then fed to a difference integrator built around the second op-amp, which sums the difference between the amplified analog signal and the digital output from the FPGA. This integrated difference signal is then compared to a reference voltage (V_{REF} = 1.65V or one-half of the op-amp supply voltage, this serves as a virtual ground in a single-supply circuit.) When the integrated difference is greater than the reference voltage, the comparator produces a high output, which is latched through a series of synchronization registers before being simultaneously sent to the digital signal processing stage and back out to the difference integrator. This high logic output immediately causes the output of the integrator to start to drop, eventually falling below V_{REF} and causing the output of the comparator to fall to a logic low value, which leads to the integrated difference voltage rising again and the cycle to repeat. In the final project, the synchronization registers were clocked at 64.8MHz along with the rest of the FPGA. The delta-sigma circuit oscillation had a period of roughly 1.3MHz, but the duty cycle changes to follow the amplified input signal from the receiver circuit in order to keep the output of the difference integrator close to V_{REF}. This variation in duty cycle is what enables the digital signal processing module to generate a digital reconstruction of the received signal.

1.2.1 Overview of digital signal processing modules

The Data Acquisition system contains twelve long series of Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) in parallel to condition the returning prior to the threshold detection system, which ultimately determines which data qualifies as a strong enough return signal to be stored to the BRAM for analysis. While these long parallel filtering structures take up relatively large quantities of FPGA resources, they have minimal control requirements. This is because they all designed to run continuously, shifting data through at the constant 65MHz clock rate, or else to sample data at a constant 1MHz clock rate.

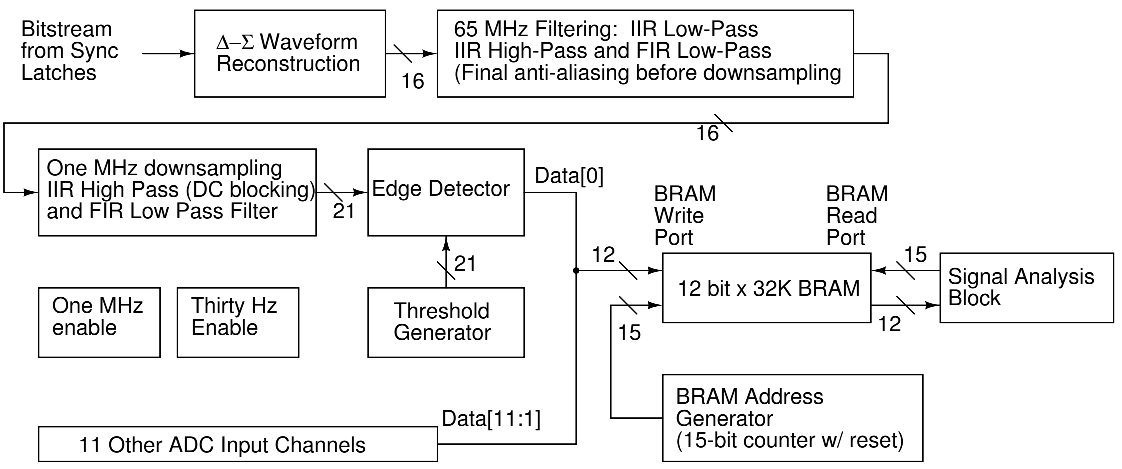

Bitstream from Sync Latches
Δ–Σ Waveform Reconstruction
16
65 MHz Filtering: IIR Low-Pass IIR High-Pass and FIR Low-Pass (Final anti-aliasing before downsampling
16
One MHz downsampling IIR High Pass (DC blocking) and FIR Low Pass Filter
21
Edge Detector
Data[0]
21
Threshold Generator
One MHz enable
Thirty Hz Enable
BRAM Write Port
BRAM Read Port
12
15
12 bit x 32K BRAM
15
Signal Analysis Block
12
BRAM Address Generator (15-bit counter w/ reset)
Data[11:1]
11 Other ADC Input Channels

Figure 1.5: Signal flow through data acquisition system digital processing

In part because many of the Data Acquisition subsystems are designed to function in a relatively straightforward manner, control of the entire system involves very few control signals and conditions. All of the digital modules within the Data Acquisition system derive all of their timing and control signals from a combination of three timer signals. First there is the 65MHz (more accurately 64.8MHz) system clock which synchronizes all activity in the FPGA. Then there is the `one MHz enable` (created by dividing the 64.8 MHz system clock by 64) which sets the down-sampling and buffering rate, as well as providing the signal to increment the BRAM write address pointer. Finally, there is the `thirty Hertz enable` which resets the BRAM address generator and the Threshold generator to prepare the system to process the signal returning from the pulse that is sent to the transmitter array.

The system completes a full data acquisition cycle thirty times every second, with all major sub-processes either running continuously or triggered to start or to reset upon receiving the thirty Hertz enable signal. Most importantly, this is the trigger signal that initiates the generation of the 40kHz output pulse (see figures 1.1-3). At the same time, the Threshold generator resets to its maximum level, to prevent the system from getting overloaded by intense reflections from nearby objects in the first few milliseconds after the pulse. Finally, this pulse resets the BRAM pointer so that the bottom address of the space corresponds to data that gets written at the moment the pulse is sent, with the BRAM address then implicitly coding the number of microseconds after the most recent pulse was sent that the data in that address was recorded.

1.3 Module Technical Details

1.3.1 Pulse Generation

The output pulse is send to the pulse amplifier on a pair of digital output pins. In order to transmit a sixteen cycle pulse at 40kHz, the two pins are switched on and off twice every 1620 clock cycles (64,800,000 / 40,000 = 1,620). Instead of switching both pins at the same time (producing a differential square wave on the output) the outputs are shifted 120 degrees out of phase with each other. The result is that the difference between the two signals is a slightly less rough stepped approximation of a sine wave, of the form (0, +1, +1, 0, -1, -1, 0, +1, +1, 0, -1, 1, 0, ...). (see figure 1.2)

1.3.2 Delta-Sigma Waveform reconstruction

Recall from the description of the delta-sigma hardware interface (1.1.3) that the relevant information about the incoming waveform is encoded in the changing duty cycle of the bitstream coming from the synchronization latches. This bitstream is sampled at 64.8MHz, but it is recording information about a signal with a nominal frequency of 40kHz. This one-bit, high-speed signal can be used to generate a running total of the number of ones and zeros that have come in, adding a small about to the sum if we get a one, and subtracting that same small amount is a zero arrives.

If we could count on the number of ones and zeros being equal on average over the long term, then the running total derived by adding and subtracting the same small amount would produce a reasonable approximation of the original analog signal (albeit with a substantial high-frequency saw tooth signal superimposed). However, experiments revealed that each channel had a substantial long-term bias towards either ones or zeroes. As a result, the delta-sigma module incorporates two strategies to limit the effect of this bias. First, by keeping track of the average bias, the amount that is added and subtracted each time can be made different from each other. Second, a simple weighted average decay function is incorporated, ensuring that over the long term the average value of the running average will decay to zero.

1.3.3 Signal Filtering and Down-sampling

While signal filtering and down-sampling may seem like distinct functions to be performed by separate modules, it was actually easier to add a latch enable signal to the filter modules that had already been designed for high frequency processing than to separate the functions. Because IIR and FIR filters are inherently implemented as latched pipelines, it was simple to design a simple set of filter modules to perform all of the high-frequency and low-frequency filtering.

Three fairly simple filters made it into the final design. Two of these were single pole IIR filters, essentially weighted averages, configured to act as either high-pass or low-pass filters. The third filter was an eighth-order FIR low-pass filter. As can be seen in the Verilog code listing, each filter had its specific purpose that it was used for. The IIR LPF filter was specifically used to reduce the sawtooth component superimposed on the signal coming out of the delta-sigma reconstruction module. The HPF IIR was used to reduce any residual DC bias that was left from the delta-sigma module, both before and after down-sampling. Finally the FIR filter (describe as a “triangle FIR” in the code for the shape of its impulse response and coefficients – 1-3-5-7-7-5-3-1) was used to apply a final smoothing and anti-aliasing function, both before down-sampling and before the threshold detection module.

1.3.4 Threshold Generator and Edge Detector

The purpose of the threshold generator is to calculate an informed guess for how strong a signal should be in order to count as a bona-fide reflection. The problem arises from the fact that radiated power intensities for all energy radiators (including acoustic transmitters) falls off proportionally to the inverse of the distance to the source, squared (1/r^2 relationship). For a sonar or radar system, the problem is even worse: in these cases, the energy lands on an object to be reflected back falls off as 1/r^2, while the small amount of energy that is reflected back itself falls off as 1/r^2. The net effect is that the amount of acoustic power that we can expect to receive as reflected energy decays with distance proportionally to 1/r^4.

The problem that we face is mitigated somewhat be the fact that our systems are designed to directly detect voltage (or sound pressure, directly at the receiver). Sound pressure is proportional to the square of power level, as a result, the measured voltage signal for an echo received from an object can be expected to follow a 1/r^2 law.

The final assumption that should be explicitly stated is that the distance to an object from the transmit-receive array will be proportional to the amount of time that has passed since the most recent pulse was sent. Because of this, we can use a simple time counter to calculate the distance that a signal arriving at any given instant must have traveled before arriving at the receiver.

The threshold generator utilizes the Xilinx Corgen divider module to compute a form to the following relationship: (within specified bounds and for `t_count` grater than some defined minimum value):

$$Threshold = \frac{Threshold_{MAX}}{t_{count} \times t_{count}}$$

The received signal on each channel is then compared to the calculated threshold. Whenever the signal rises above the threshold in the positive direction, the edge detector puts out a logical one, which is held high for 12 BRAM data sample cycles (12 microseconds, or approximately half of the period of a 40kHz signal.

1.3.5 BRAM Data Buffer

The BRAM buffer is the interface between the data acquisition system and the signal analysis system. In the same way that time is counted by the Threshold generator for the purposes of generating a threshold signal, the BRAM address generator keeps a count of microseconds passed since the pulse was transmitted and stores one row of data, twelve-bits wide, from the edge detection module into the BRAM. (The one MHz enable signal also serves as the BRAM write enable). The BRAM is then able to record 32,768 microseconds worth of information on its twelve channels – in theory, enough time to for sound to make a round trip of well over 30 feet, or enough to provide a theoretical outer limit of approximately 16 feet or 5 meters for detection.

In reality, it turned out that clear detection of echo signals beyond 2-3 meters was challenging and required special circumstances, but detection of object less than 2 meters away became routine. The signal analysis section and the appendix contain numerous images showing what good echo signals detections looked like when displayed on the Tektronix Logic Analyzer.

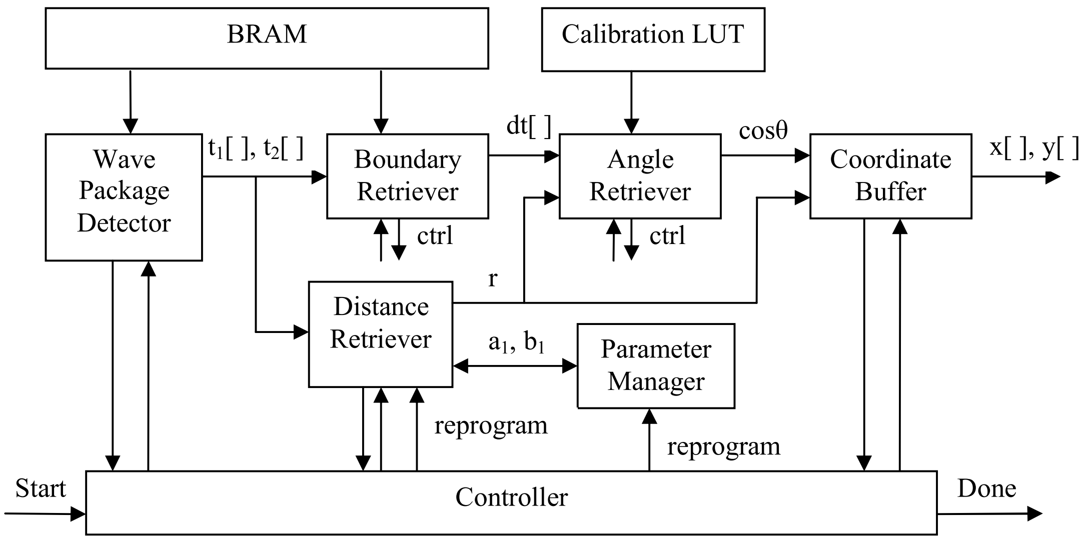
SECTION 2: SIGNAL PROCESSING UNIT – ZHEN LI

Signal Processing Unit Overview

2.1.1 Introduction

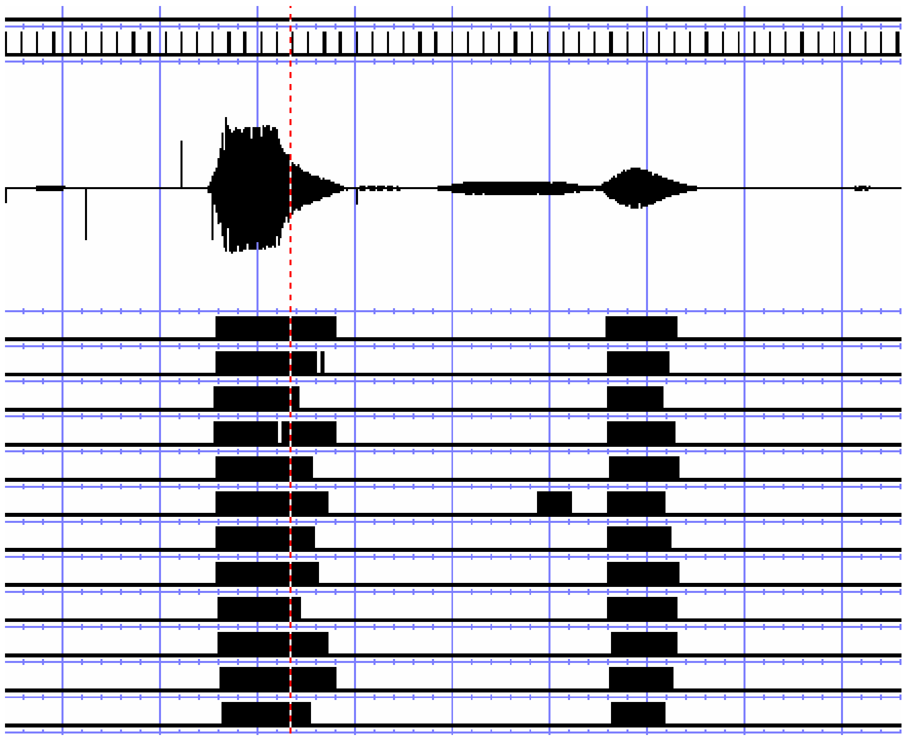
The Signal Processing Unit takes binary streams from the Data Acquisition Unit as inputs, and provides the (x, y) coordinates of objects as outputs. The binary streams from the Data Acquisition Unit represent the reflected signals received from microphones at a sample rate of 1 Mbit/s (with the signal period 25us). The distances (r) between the transducer array and the objects are calculated from the transmit-to-receive time delays. The direction angles (ϑ) are retrieved from the phase delays among receivers. Finally, a set of coordinates are buffered and passed on to the User Interface Unit.

This unit contains seven different modules: Wave Package Detector, Distance Retriever, Boundary Retriever, Angle Retriever, Coordinate Buffer, Parameter Manager, and the Controller, as shown in Graph 2.1.



Graph 2.1 Block Diagram of Signal Processing Unit

This unit is an asynchronous machine on the system level. The Controller signals each module to `start` and each module signals back to the Controller with `done`. Wave Package Detector scans through the whole BRAM and divides the raw data in to packages. At the same time, it is responsible for noise rejection. As shown below (Graph 2.2), there are two wave packages, and an obvious noise pick-up on channel 6. Distance Retriever takes each wave package and calculates the distances of corresponding objects. In parallel, Boundary Retriever, which should really be called Phase Retriever, takes each wave package and tracks the (unwrapped) phase difference between the first and last signal. Angle Retriever calculates the directional angle from the distance and the phase difference. Then Coordinate Buffer turns the distance and angle into x-y coordinates and updates the output register array.


Graph 2.2 Received Signals (12 channels)

2.2 Wave Package Detector

The top-level Wave Package Detector (WPD12) has twelve single-channel wave package detector (WPD) instances. WPD12 reads the BRAM, coordinates the twelve WPDs for twelve channels, and talks to the Controller. It outputs a set of t1 and t2, indicating the starting and ending time of each wave package.

2.2.1 Signal Channel Detection

2.2.1.1 Detection Rule

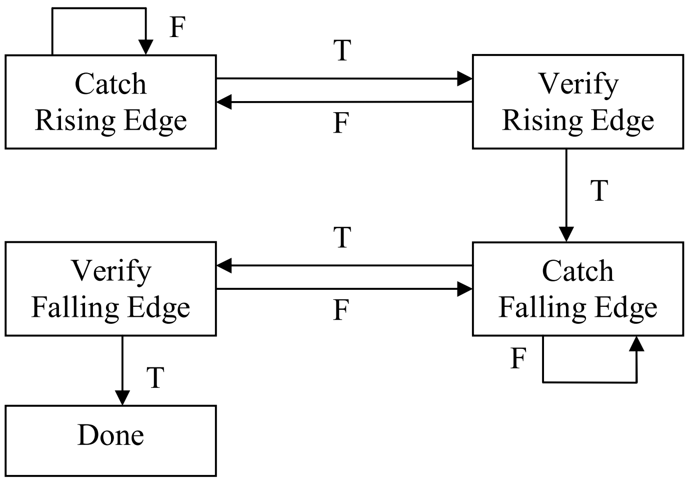
Conceptually, if the following events happen in sequence, a valid wave package (on a given channel) is detected:

1) A clear rising edge (not a glitch from ADC)
2) The signal is periodic with a period of 25us approximately
3) The signal has at least N periods
4) A clear falling edge (not a small blanking)

By observing the experimental data, the periodicity of reflected waves is almost guaranteed (thanks to the band-pass filter at the data acquisition side). Therefore, the detection rule #2 can be omitted.

2.2.1.2 Noise Immunization

Further experiments show that single-period noise is very rare, but very narrow glitches are quite common. This is perhaps because of the quiet background at 40 kHz. (Narrow glitches have much broader bandwidth.) So few-period receptions are better to be considered as detections. Therefore, the detection rules can be simplified. The state transition diagram is shown below in Graph 2.3.



Graph 2.3 State Transition Diagram of WPD

2.2.2 Multi-Channel Consideration

2.2.2.1 Operation

A real reflection signal is detected when most (or all) channels receive a signal approximately at the same time. *Done* signal is asserted when one signal is detected. Wave package detection finishes and *Finish* signal is asserted when the entire BRAM is scanned.

2.2.2.2 Missing & False Alarm

There are two major sources of noise in multi-channel assembly: single-channel missing and single-channel false alarm. One solution to the single-channel missing/false alarm problem: if N (N is big, e.g. N=10) channels get their detections (*done* signals) AND the rest (12-N) channels still have not caught their rising edge, the (12-N) channels will be marked as MISSED in a 12-bit indicator. (Distance Retriever will not take MISSED channels in the average, and Boundary Retriever will omit MISSED channels in linear prediction.)

Conversely, if M (M is small, e.g. M=2) channels get their detections (*done* signals) AND the rest (12-M) channels still have not caught their rising edge, the M channels will be regarded as false alarm, and these detections will be ignored.

Experimental results indicate that single-channel missing/false alarm is very rare. Furthermore, even if this happened, only one object will possibly be screwed for 1/30 second. Therefore, this part is implemented but not used/tested. (Because this part of work is done in parallel with the transducer-receiver part)

2.3 Distance Retriever

2.3.1 Basic Functions

The Distance Retriever module is pretty straight forward. It takes the average of transmit-to-receive time delays, and calculates the distance directly:

$$r = vt_d / 2$$

Because the length of reflected wave depends on the shape and surface condition of the object, t_d is measured from transmission starting point to leading edge of received signals.

2.3.2 Calibration (Parameter Manager)

Due to the transducer delay and other unknown reasons, a better formula for r is as follows:

$$r = at_d + b$$

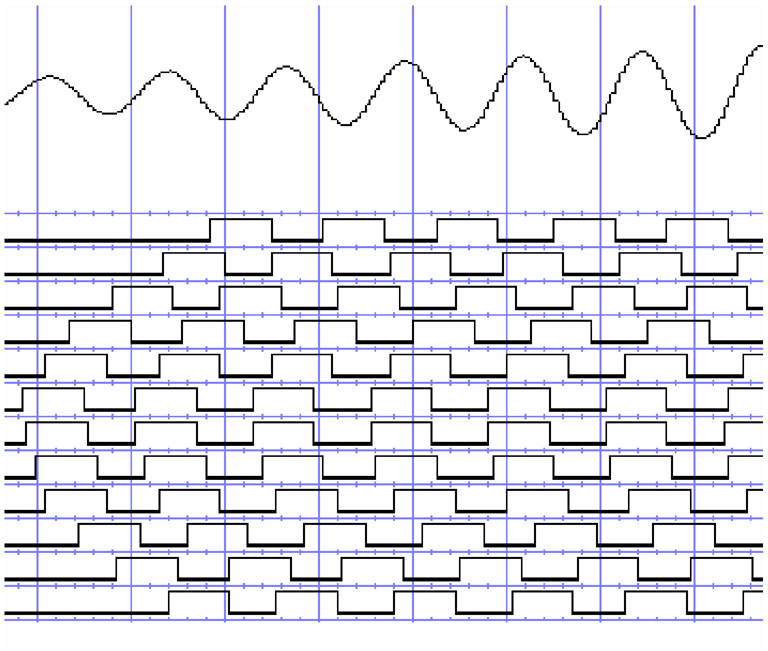
With a calibration function, users may choose to calibration the coefficients when needed. Experiments show that in the formula above *a* is a relatively stable constant, but *b* may change when sensitivity (reception threshold) changes. Thus, *a* is pre-calibrated in hardware, and *b* is programmable at run-time. Incorporated with the Calibration function in the User Interface Unit, users will be instructed to put one single object 3 meters straight ahead of the transducer. The system back calculates *b* and stores it in the Parameter Manager module.

2.4 Boundary Retriever (Phase Retriever)

2.4.1 Motivation & Challenges

The directional angle of an object can be calculated from the arrival time differences of reflected signals. One way to get the time difference is from the (unwrapped) phase difference. (More alternatives are discussed in Chapter 2.4.3) This is the most difficult, interesting and challenging part of the whole design. The two major difficulties are:

1) The phases of received signals are curved
2) Phase differences between two neighboring receivers can be larger than π, even $2*\pi$.



Graph 2.4 Phase Information in Received Signals

2.4.2 Linear Prediction Method

If it is known a priori that the phase difference between two given channels (say channel 1 and 2) is smaller than π, then the rest can be tracked in the following way:

1) Take channel 1 as a reference (Phase(1)=0), then get Phase(2) within [-π, π]
2) Using Phase(1) and Phase(2), do a linear prediction of Phase_pred(3)
3) Look for Phase(3) within [Phase_pred(3)-π, Phase_pred(3)+π]
4) Repeat (2) & (3) for the nest channel.

(Staring at the procedure, it is just the way how naked eyes tracks phase)

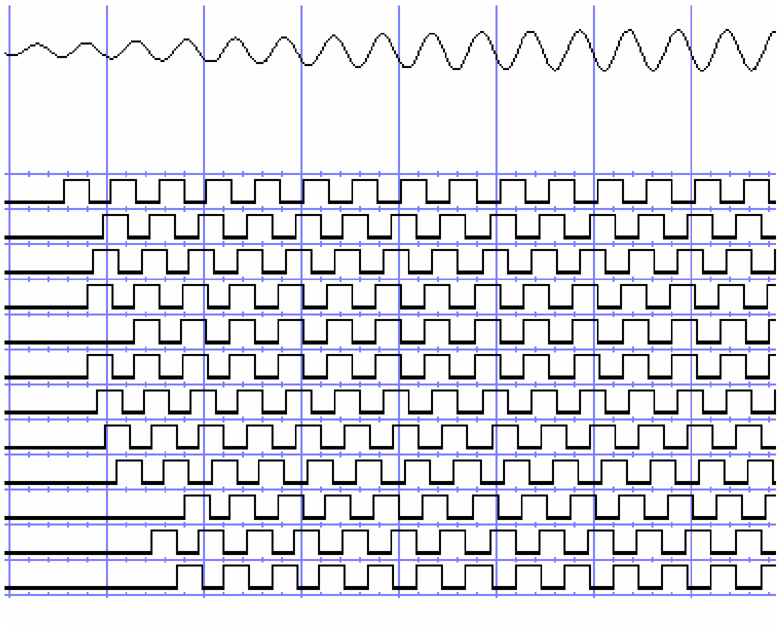
2.4.3 Reference Point Decision

There is still one more question left, namely the "a priori small" phase difference. Experimental results suggest that one good indicator is the leading edge (wave front). This means, if the wave is coming from left (right), choose the most left (right) two receivers as references. If the wave is coming from straight ahead, the leading edges can be arbitrary, but this does not matter because in this case none of the neighboring phase differences is larger than π.

2.4.4 Possible Alternative Solutions

2.4.4.1 Leading Edge Approximation

One way to estimate the reflected wave arrival time difference is to simply take the time difference of the leading edges. However, this approach is very inaccurate because in most cases the leading edges are very noisy, as shown below in Graph 2.5. This is mainly because the threshold on each channel is the same, but the received signal strength on each channel can vary. This was the original thought and that is why this module is called Boundary Retriever.



Graph 2.5 Leading Edge

2.4.4.2 Half-Period (Max) Phase Shift Assumption

This method assumes every pair of neighboring receivers gets signals with phase difference less than π. This assumption is only good for $\vartheta<15^{o}$.

2.5 Angle Retriever

2.5.1 Basic Function

Given the phase difference (or time difference *dt*) of two received signals at two receivers separated by a distance *d*, the first order estimation is as follows:

$$\cos\theta = vdt / d$$

This formula is a far field approximation. For near objects, this approximation may lead to an error as large as 10% at $r = 20$cm. (see Table 2.1)

2.5.2 Calibration Look-up Table

Exact analytical solution can increase the accuracy, but a better way is to hammer it away. Given the complexity of this calibration, it does not seem to be a good idea to let users do the calibration. Moreover, the calibration coefficients are stable over different situations, a built in look-up table is a proper choice. The method is just brute force. Make grids in the region of interest, get an experimental measurement for each grid, and then store the calibration data in a ROM.

2.6 Coordinate Generator & Buffer

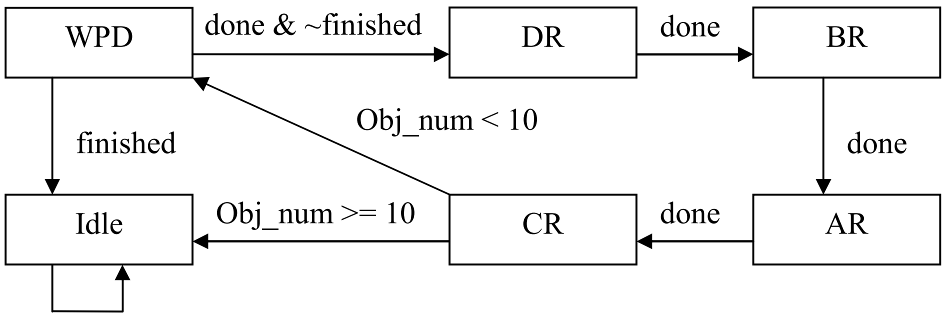
This module takes the polar coordinates, and converts them into Cartesian coordinates (x, y) for XVGA display. To convert cosine into sine and etc, some math functions in Core-gen, like square-root, are invoked.

2.7 Controller

2.7.1 Main Finite State Machine (FSM)

Due to the undetermined length of each wave package, an asynchronous system seems to be a proper choice. The controller signals each module to `start`, and each module talks to the Controller when it is `done`. The Controller also manages the memory read port, and distributes it to Wave Package Detector or Boundary Retriever when proper.

The Controller talks to the outside world. It starts this unit when the Data Acquisition is done, and signals the User Interface Unit to update the image when the signal processing is done. The Controller is a finite state machine, as shown below.



Graph 2.6 State Transition Diagram of the Controller

2.7.2 Distributed Start/Done Signal Management

In order to talk with the Controller, each module manages its own `start/done` signal. Each module will be idle unless its `start` signal is asserted. Each module has to have a `done` signal generator.

SECTION 3: USER INTERFACE DESIGN – BRIAN WONG

User Interface Design Overview

3.1.1 Introduction

The sonar user interface aims to provide a realistic simulation of a professional sonar system found in submarines. The sonar project user interface should simulate both the visual characteristics of a sonar system, but also the audio output of the system when objects are within detectable range.

In the final version of the sonar user interface demonstrated to the 6.111 course staff, the system delivers key features of professional sonar including a sonar grid background, a sweeper line with an alpha-blended trailing edge, warning and notification audio signals, multiple object representation, distance and angular estimations, and an intuitive graphical user interface menu.

The following sections, 3.1.2, 3.1.3, and 3.1.4, provides a brief overview of the key features in the sonar project user interface. Section 3.2 provides a more in-depth description of the technical implementation of each sub-module. The following picture shows the final version of the sonar user interface.

3.1.2 Technical Features

The sonar project user interface provides a realistic simulation of a professional sonar system. Given the current (hcount, vcount) of the XVGA control signal, a module generates a 24-bit RGB pixel signal of the sonar grid background. The sonar grid module creates the background by reading sprite data from a block memory located onboard the 6.111 labkit. The following picture shows the sonar grid generated on a black background. The color chosen for the sonar grid is dark green to provide a non-obtrusive view of the foreground objects.

In addition to the sonar grid module, the sweeper module generates the sweeper line found in most commercial sonar systems. The sweeper module draws a pie-shaped object on the screen by checking if (hcount, vcount) are within an angular bound and within a pre-determined distance from the center of the screen. The angular bound is represented by an upper angular bound and a lower angular bound. In order for the module to create a rotating sweeper line, the angular bound increments over time until it resets itself to zero if it is larger than 360 degrees. The sweeper module also provides information to the alpha-blending module for the incremental alpha-blending effect.

Pixels generated by the sweeper module are fed into the alpha-blending module for alpha-blending effect. If the (hcount, vcount) are within the range of the angular bound that requires alpha-blending, the sweeper pixel is alpha-blended with its background grid pixel as the final output. The alpha-blending weight for each RGB value is assigned based on the location of the (hcount, vcount) within the pie-shaped object generated by the sweeper module.

The menu module and the text generation module provides basic user interface to the sonar system. The menu module generates a three-item menu that enables interaction with the labkit PS/2 mouse, while the text generation module generates the text pixels for displaying angle, distance, and speed estimation data. The siren generation module and the alarm generation module generate the appropriate audio output to simulate a professional sonar setup. The project system generates a sonar "ping" when the sweeper intersects with an object within detectable range. An alternating warning siren is generated when the object is within a 110-pixel radius of the center.

3.1.3 Module Overview

The sonar user interface module is composed of twelve different sub-modules. These sub-modules provide core functionality to generate the 24-bit RGB display pixel to the XVGA monitor and the various audio signals to the speaker. The following section provides a brief overview of the feature of each module. The technical implementation details of these modules are provided in Section 3.2.

1. Sonar Grid Module
   This module is responsible to generating the 24-bit RGB background pixel for the XVGA display module on the 6.111 labkit.

2. Sweeper Module
   This module is responsible for generating the sweeper pixel.

3. Object Representation Module
   This module is responsible for generating the object representation pixel.

4. Alpha-blending Module
   This module is responsible for generating the alpha-blended 24-bit RGB pixel the blends the sweeper pixel and the background pixel.

5. Siren Generation Module
   This module is responsible for generating the audio signal for the sonar "ping" produced when objects are detected in range.

6. Alarm Generation Module
   This module is responsible for generating the audio warning signal when objects are within a critical distance of the sensor.

7. Binary to ASCII Converter Module
   This module is responsible for converting binary-represented numbers to ASCII-represented numbers for the text generating module.

8. Menu Interface Module
   This module is responsible for generating the menu user interface pixel.

9. Text Generation Module
   This module is responsible for generating the text pixel from ASCII signal.

3.1.4 Module Block Diagram

The following block diagram provides an overview of the modular design. Some common inputs and outputs of the modules are not included in the diagram for simplicity. These signals include the 65MHz clock, vsync_p, hsync_p, and blank_p. The 65MHz clock is chosen throughout our system because the XVGA signal requires a 65MHz update rate to draw each pixel 60 times per second.

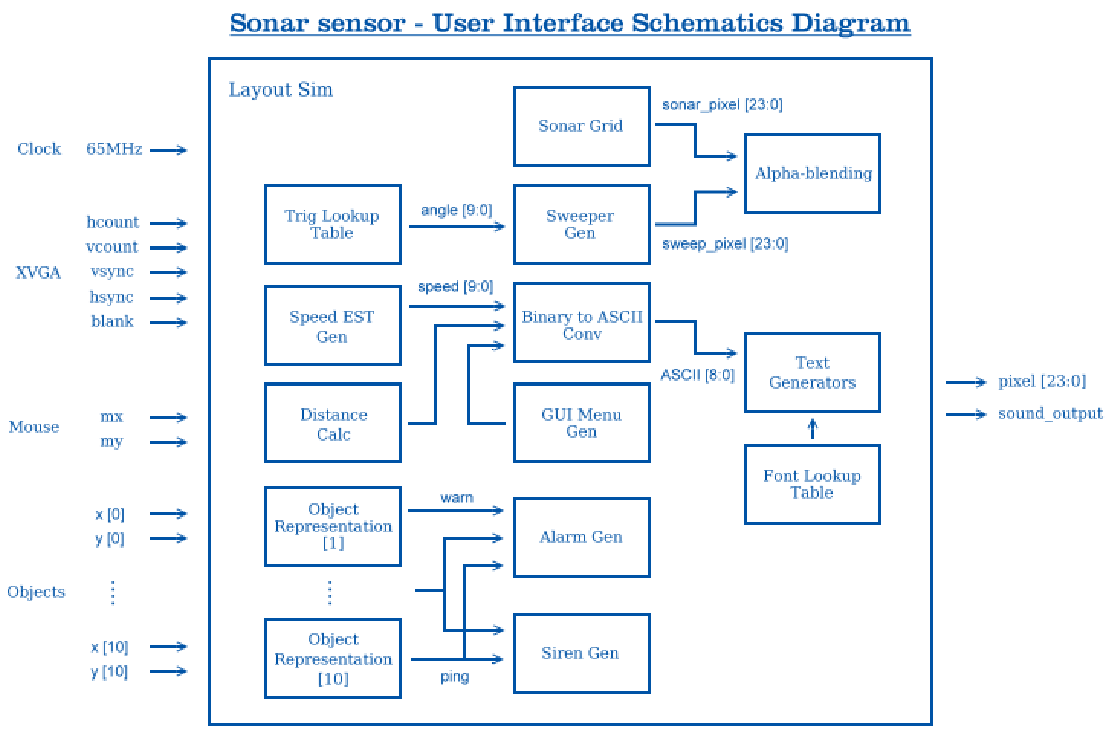


Figure 1. User interface system block diagram

3.2 Technical Details

3.2.1 Sonar Grid Module

The sonar grid module generates the pixel required to display the sonar background. The module is composed of a `(512*384*1)`–bit block RAM that stores the sonar pixel for one quarter of the screen, and core logic that replicates the BRAM data to draw the entire sonar module on a `1024*768` XVGA monitor. The block RAM data is generated by a Java program written by Brian Wong. The data is encapsulated in a COE file format that is accepted by the Xilinx single-port BRAM generator.

The BRAM stores sonar grid bitmap for the top left quadrant of the screen. Transformation to the coordinates is applied based on the location of hcount and vcount. This transformation enables the system to save 75% BRAM memory space since the sonar grid is symmetric about the x-axis and the y-axis. As an example transformation, if `(hcount, vcount)` pair is located in the first quadrant (top right), then `hcount' = 1024 - hcount`, and `vcount' = vcount`.

The `1024` parameter arises from the fact that we are using a XVGA resolution display. No transformation to vcount is necessary in this case. The transformation translates a coordinate pair in the first quadrant to an equivalent coordinate pair in the second quadrant, where the sonar grid is represented by data in the BRAM. This module has a 2 clock cycle latency for the BRAM lookup and the coordinate transformation.

3.2.2 Sweeper Module

The sweeper module is composed of an inverse tangent lookup table in the BRAM and transformation logic for `(hcount, vcount)` pairs. The BRAM contains inverse tangent data for the first quadrant of the sonar scope. Given a `(hcount, vcount)` pair, the module checks to see if the coordinates are within an angular bound. To generate the pie-shaped sweeper, the module takes the current `(hcount, vcount)` pair, performs a coordinate transformation to the first quadrant, and feed them into an inverse tangent lookup table.

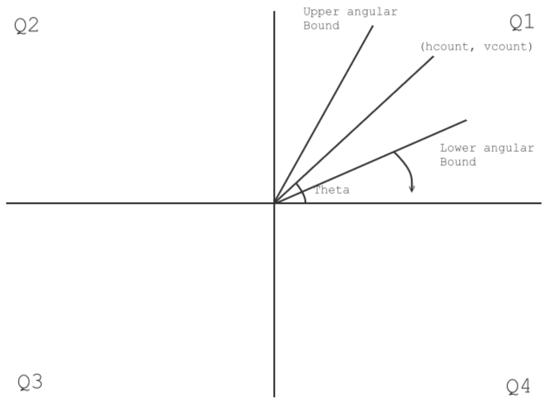


Figure 2. Inverse lookup for bound checking

The angular bound is a time-evolving parameter. The lower angular bound denotes the leading edge of the sweeper pie, and the upper angular bound denotes the trailing edge of the sweeper pie. The upper bound and the lower bound increments slowly with time until either bound reaches 360 degrees, at which point the bound is reset to zero. If the inverse tangent lookup of a (hcount, vcount) coordinate pair indicates that the coordinate lies between the lower bound and the upper bound, a green RGB pixel is generated as the output of the module.

The BRAM module stores an integer angle for each `(x, y)` pair in the first quadrant. The memory look-up address is determined by a mapping `ADDR = y*1024 + x`. This module has a 2 clock cycle latency for the BRAM inverse tangent lookup and for the coordinate transformation.

3.2.3 Object Representation Module

The object representation module generates the pixel to indicate the presence of objects within the detectable range of the sonar sensor. Each object is represented by a pulsating circle on the sonar sensor. The color of the circle changes depending on the location of the object. If the circle is within the 220 pixel range, the color becomes yellow. If the circle is within the 110 pixel range, the color becomes red. In addition, the object representation module asserts a logical high on the `warn` signal, which tells the alarm generation module to produce a warning audio output.

The pulsating effect on the object is generated by a finite state machine. The FSM has a total of six states. The six states presents different rate of change of the `rsquared` register value. The `rsquared` register indicates the radial size of the object. The following diagram provides an overview of the FSM.

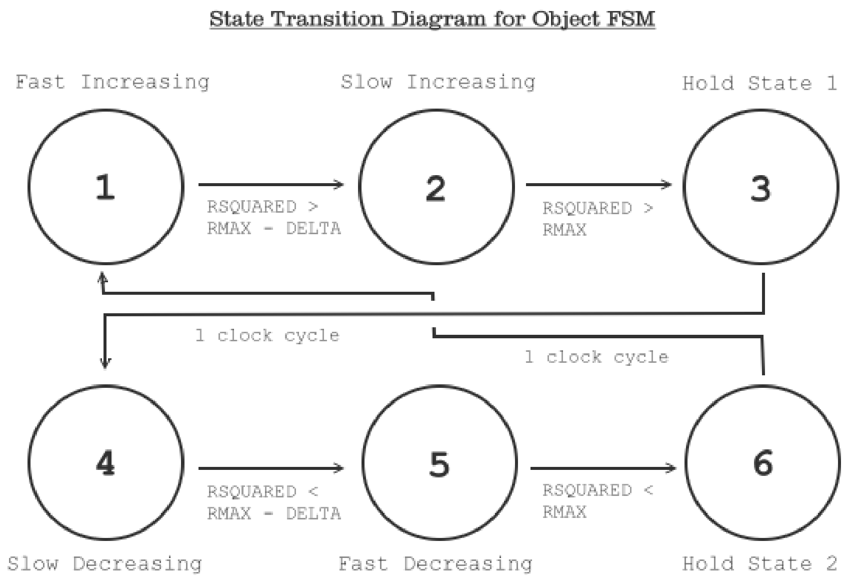


Figure 3. State Transition Diagram for FSM

The prototype sonar system provides representation of up to three different objects. However, once the signal processing units improve their accuracy of differentiating objects within range, the modular design enables the user interface to display as many objects as possible.

3.2.4 Siren Generation Module

The siren generation module generates a 1,030Hz audio signal whenever the sweeping line intersects with an object on the sonar sensor. The siren generation module is triggered by a `sound_on` signal. The `sound_on` signal is simply a logical `AND` of the `final_object_pixel_p[30]` and the `final_sweeper_pixel_p[30]`. These two values are 24-bit pipelined registers that hold the RGB pixel value of an object and the sweeper pixel. If both are asserted, that implies an intersection of the two. The siren audio signal is asserted for about 150 milliseconds. The choice for the short siren assertion is because if there are multiple objects within detectable range, then long siren assertion may potentially create ambiguous audio signal.

The module contains an 18-bit registered counter that increments by one at every positive clock edge of the 65MHz global clock. When the counter reaches a count of 60,000, the audio output signal switches from a logical high to a logical low, or vice versa. This translates into a 1,030 Hz signal on the output of the siren generation module.

3.2.5 Alarm Generation Module

The alarm generation module generates a 400Hz/700Hz alternating audio signal to indicate one or more object is within a close distance of the sensor. The mechanism for generating the 400Hz and the 700Hz signal is very similar to that of the siren generation module, except with different timing parameters. The alternating frequency feature is provided by a scalar `chooseA` register that alternates the frequency generated.

3.2.6 Binary to ASCII Converter Module

The binary to ASCII converter module converts a binary representation of number into an ASCII representation of number. This module provides the input to the text generation module, which requires its inputs to be ASCII based.

The module limits integer conversion to a maximum of three digits, i.e. the largest integer represented is 999. This is an arbitrary choice. If there is a need for conversions of larger numbers, it is a trivial process to add support for more digits. Nevertheless, the current sonar system does not have data to display that requires more than three digits.

This module first converts binary numbers to a base-10 number by applying a recursive `(modulo 10)` process on the binary number. Initially, the binary number to be converted is placed in an internal 10-bit register. At each positive edge of the 65Mhz clock, the register is decremented by one. Simultaneously, one is added to the least significant digit of the base-10 representation of the binary number. If the least significant digit is a nine, than we increment the next higher-order digit by one. Similarly, we apply the same process to all other base-10 representation of the digits.

When the module has decremented the internal register to zero, it performs a bit concatenation to turn the base-10 number into the equivalent ASCII character. For an arbitrary digit, `d`, the module takes the 4 lowest order bits and outputs `{4'b0011, d[3:0]}`. This module has a latency that depends on how big the input integer is. However, the data ready latency of the module is much faster than the update frequency of incoming data.

3.2.7 Menu Interface Module

The menu interface module generates the 24-bit RGB pixel for the user menu located on the right side of the screen. The module interacts with the mouse inputs to provide an intuitive user menu for the sonar system. The menu provides three items to choose from – “Calibration Mode”, “Hide Menu Mode”, and the “Hide All Mode”. These items can be chosen by moving the mouse over one of the three menu items, and performing a simple left click.

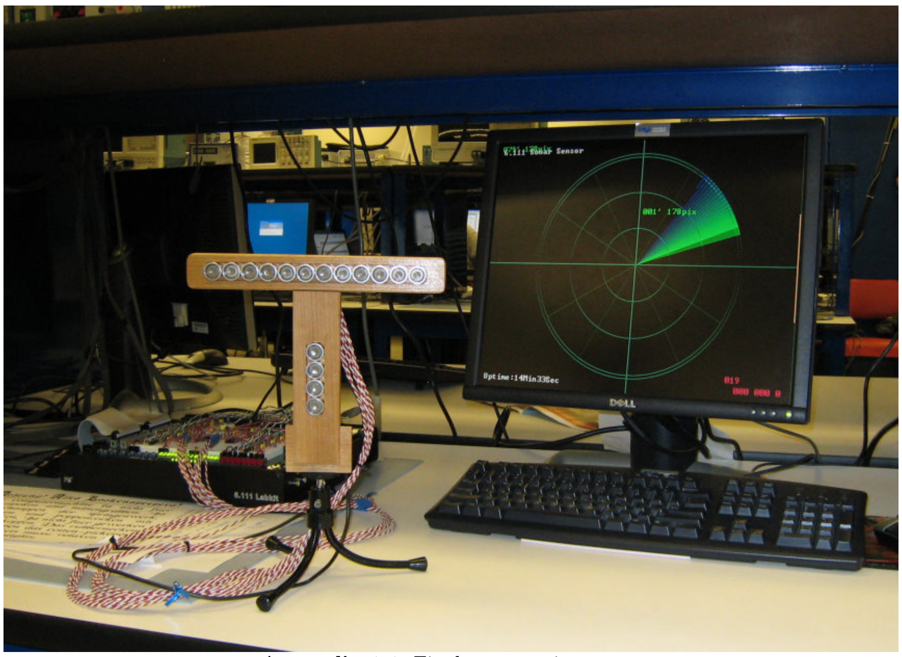
The calibration mode changes the output to the basic sonar grid background, but with an additional line that requests the user to stand 3 meters away from the sensor. This mode enables the signal processing module to fine tune its parameters. The hide menu mode hides the menu for a clean display, but retains all the key information on the screen, including distance, angle, and speed estimation data. The hide all mode hides all text displays. The user should only observe the sonar grid, the sweeping line, and the objects within detectable range.

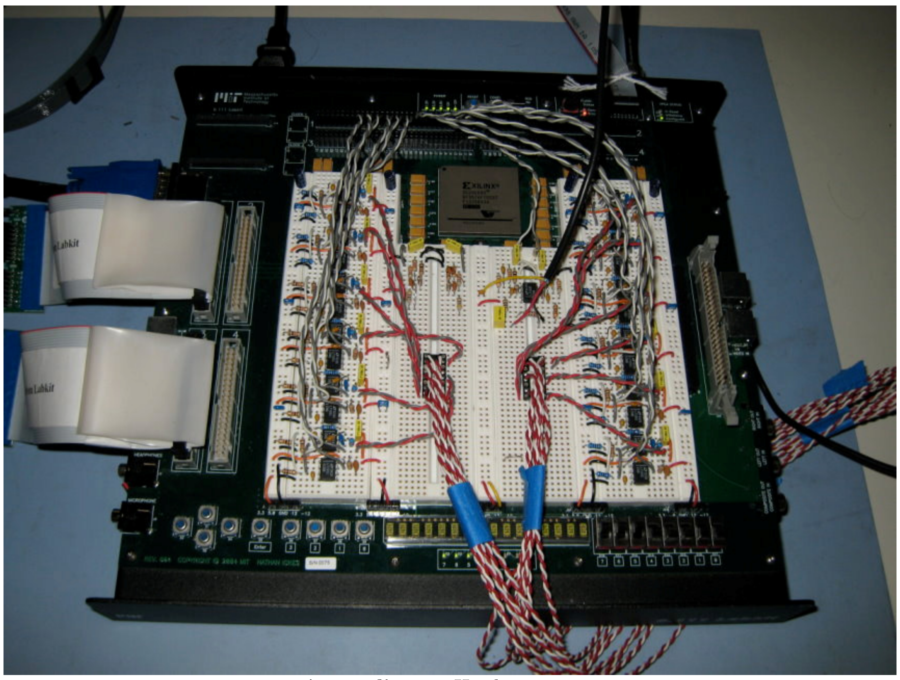
A mouse right click brings the user from one of the three modes back to the normal mode, where all features of the sonar sensor are displayed on the screen.

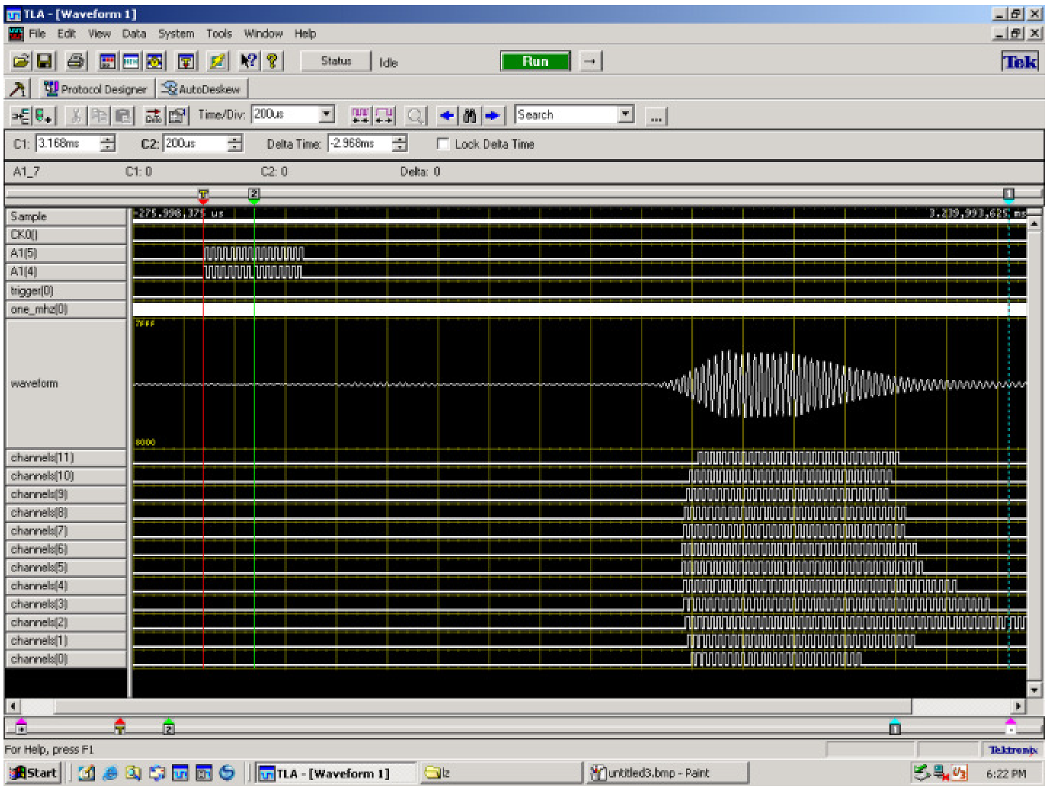
3.2.8 Text Generation Module

The text generation module generates the text in the user interface. This module is provided by previous terms of the 6.111 course. The module takes ASCII character inputs and provides a 24-bit pixel output. The module utilizes a block RAM module to generate the character on the display.

Appendix


Appendix 1.1. Final sonar project setup


Appendix 1.2. Hardware setup



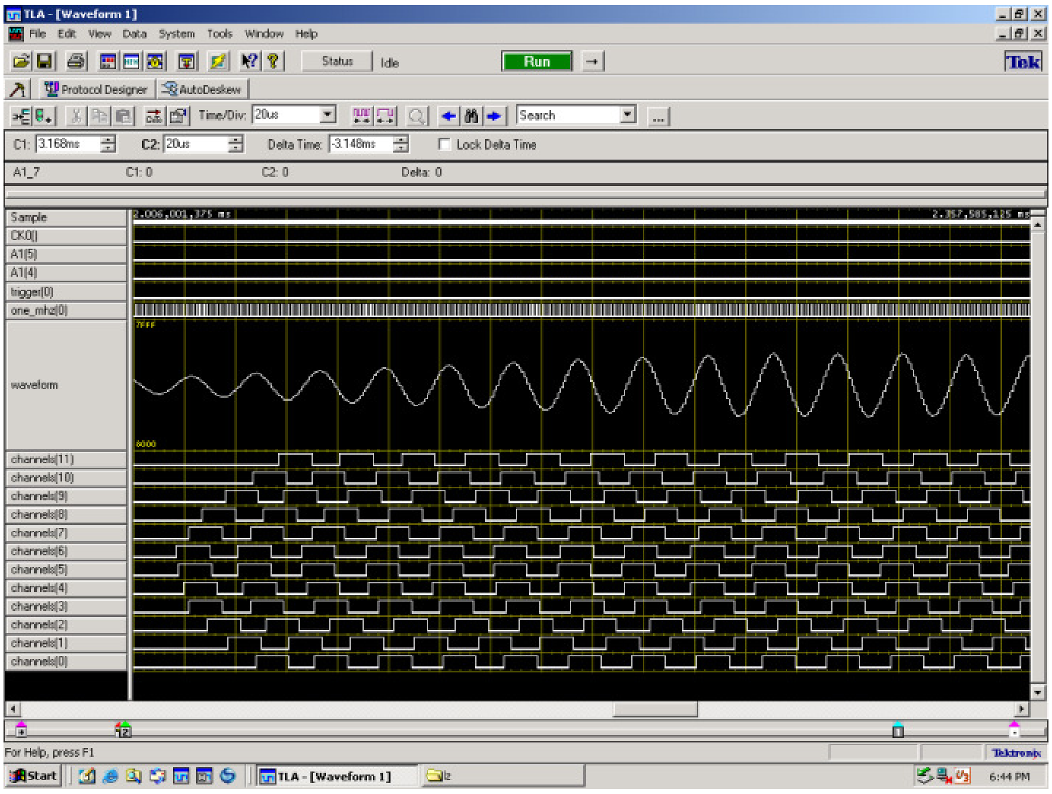
Idle

Time/Div: 200us

C1: 3.168ms C2: 200us Delta Time: -2.968ms Lock Delta Time

A1_7 C1: 0 C2: 0 Delta: 0

-275,998,375 us 3,239,993,625 ms



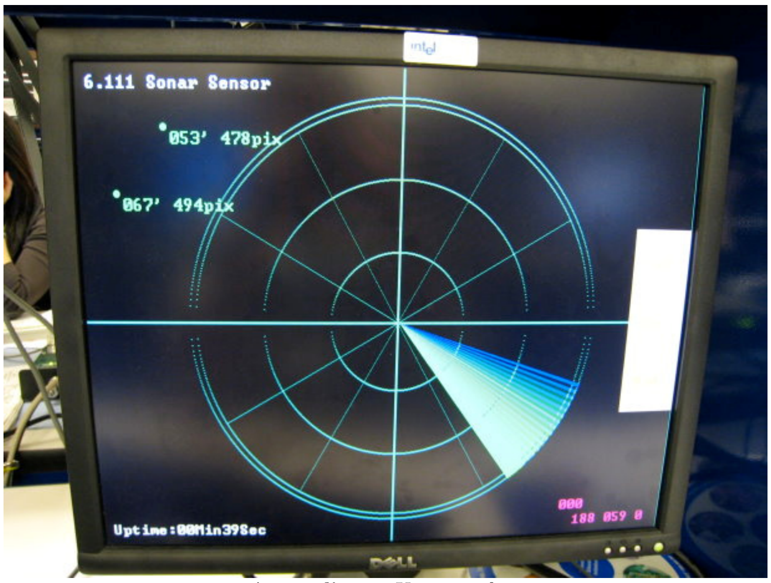
Idle

Time/Div: 20us

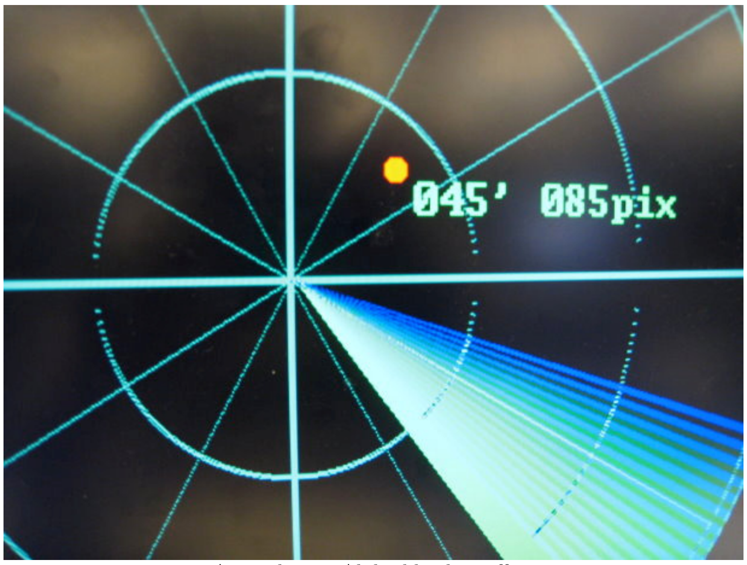
C1: 3.168ms C2: 20us Delta Time: -3.148ms Lock Delta Time

A1_7 C1: 0 C2: 0 Delta: 0

2,006,001,375 ms 2,357,585,125 ms



Appendix 1.5 User interface



Appendix 1.6 Alpha-blending effect

Verilog Code

Binary to ASCII Converter

```
Module binarytodec (clock,input_num,digit_3_ascii,digit_2_ascii,digit_1_ascii);
        input clock;
        input[9:0] input_num; // 0-360
        output[7:0] digit_3_ascii;
        output[7:0] digit_2_ascii;
        output[7:0] digit_1_ascii;

        reg[7:0] digit_3_ascii;
        reg[7:0] digit_2_ascii;
        reg[7:0] digit_1_ascii;
        reg[7:0] digit_3;
        reg[7:0] digit_2;
        reg[7:0] digit_1;
        reg[8:0] input_angle;
        always @ (posedge clock) begin
                if (input_angle == 0) begin
                        input_angle <= input_num;
                        digit_3_ascii <= {4'b0011, digit_3[3:0]};
                        digit_2_ascii <= {4'b0011, digit_2[3:0]};
                        digit_1_ascii <= {4'b0011, digit_1[3:0]};
                        digit_3 <= 0;
                        digit_2 <= 0;
                        digit_1 <= 0;
                end else begin
                        input_angle <= input_angle - 1;

                        if (digit_1 == 9) begin
                                digit_1 <= 0;
                                if (digit_2 == 9) begin
                                        digit_2 <= 0;
                                        digit_3 <= digit_3 + 1;
                                end else
                                        digit_2 <= digit_2 + 1;
                        end else
                                digit_1 <= digit_1 + 1;
                end
        end
endmodule
```

Basic Object Representation

```
module basicobject(clock,hcount,vcount,
                                x1,y1,on1,
                                warn1,
                                pixel);
        input clock;
        input[10:0] hcount;
        input[9:0] vcount;
        input[11:0] x1,y1;
        input on1;
        output warn1;
        output[23:0] pixel;

        reg[5:0] rsquared = 16; // RANGE: 16 - 36
        reg[24:0] counter = 0;
        reg[2:0] state = 0;    // Eight states
        reg[22:0] radial_dist;
        reg[22:0] point_dist;
        reg warn;
        reg safe;
        reg pixel_on;

        parameter WARN_DIST = 12100;  // WARN if within 110 pixel range
        parameter SAFE_DIST = 48400;  // SAFE if outside 220 pixel range
        parameter TERMKEY = 4194303;
        parameter RMAX = 49;
        parameter RMIN = 16;
        parameter DELTA = 8;
        parameter HPIXELC = 512;
        parameter VPIXELC = 384;
        parameter COLOR_WARN = 24'b111111110000000000000000; // RED
        parameter COLOR_SAFE = 24'b000000001111111100000000;  // GREEN
        parameter COLOR = 24'b111111111111111100000000;      // YELLOW
        parameter BLACK = 24'b000000000000000000000000;      // BLACK

        // **************************
        // 2 STAGE PIPELINING!!!!
        // **************************

        always @ (posedge clock) begin
                if (counter == TERMKEY)         begin // About quarter of a second on 65MHz
clock
                        case (state)
                                // Increasing rsquared at 2 units/cycle
                                0: begin
                                                if (rsquared < RMAX - DELTA)
                                                        rsquared = rsquared + 2;
                                                else
                                                        state <= 1;
                                        end
                                // Increasing rsquared at 1 units/cycle
                                1: begin
                                                if (rsquared < RMAX)
                                                        rsquared = rsquared + 1;
                                                else
                                                        state <= 2;
                                        end
                                // Hold state for 0.5 second
                                2: begin
                                                state <= 3;
                                        end
                                // Decreasing rsquared at 1 units/cycle
                                3: begin
                                                if (rsquared > RMAX - DELTA)
                                                        rsquared = rsquared - 1;
                                                else
                                                        state <= 4;
                                        end
                                // Decreasing rsquared at 2 units/cycle
                                4: begin
                                                if (rsquared > RMIN)
                                                        rsquared = rsquared - 2;
                                                else
                                                        state <= 5;
                                        end
                                // Hold state 0.5 second
                                5: begin
                                                state <= 0;
                                        end
                        endcase
                        counter <= 0;
                end else
                        counter <= counter + 1;

                if (x1>=HPIXELC && y1<VPIXELC) begin
                        radial_dist <= ((x1-HPIXELC)*(x1-HPIXELC) + (VPIXELC-y1)*(VPIXELC-
y1));
                        warn <= radial_dist<=WARN_DIST ? 1'b1 : 1'b0;
                        safe <= radial_dist>=SAFE_DIST ? 1'b1 : 1'b0;
                end else if (x1<HPIXELC && y1<VPIXELC) begin
                        radial_dist <= ((HPIXELC-x1)*(HPIXELC-x1) + (VPIXELC-y1)*(VPIXELC-
y1));
                        warn <= radial_dist<=WARN_DIST ? 1'b1 : 1'b0;
                        safe <= radial_dist>=SAFE_DIST ? 1'b1 : 1'b0;
                end else if (x1<HPIXELC && y1>=VPIXELC) begin
                        radial_dist <= ((HPIXELC-x1)*(HPIXELC-x1) + (y1-VPIXELC)*(y1-
VPIXELC));
                        warn <= radial_dist<=WARN_DIST ? 1'b1 : 1'b0;
                        safe <= radial_dist>=SAFE_DIST ? 1'b1 : 1'b0;
                end else if (x1>=HPIXELC && y1>=VPIXELC) begin
                        radial_dist <= ((x1-HPIXELC)*(x1-HPIXELC) + (VPIXELC-y1)*(VPIXELC-
y1));
                        warn <= radial_dist<=WARN_DIST ? 1'b1 : 1'b0;
                        safe <= radial_dist>=SAFE_DIST ? 1'b1 : 1'b0;
                end

                if (hcount>=x1 && vcount>=y1) begin
                        point_dist <= ((hcount-x1)*(hcount-x1) + (vcount-y1)*(vcount-y1));
                        pixel_on <= (point_dist<=rsquared && on1) ? 1 : 0;
                end else if (hcount>=x1 && vcount<y1) begin
                        point_dist <= ((hcount-x1)*(hcount-x1) + (y1-vcount)*(y1-vcount));
                        pixel_on <= (point_dist<=rsquared && on1) ? 1 : 0;
                end else if (hcount<x1 && vcount>=y1) begin
                        point_dist <= ((x1-hcount)*(x1-hcount) + (vcount-y1)*(vcount-y1));
                        pixel_on <= (point_dist<=rsquared && on1) ? 1 : 0;
                end else if (hcount<x1 && vcount<y1) begin
                        point_dist <= ((x1-hcount)*(x1-hcount) + (y1-vcount)*(y1-vcount));
                        pixel_on <= (point_dist<=rsquared && on1) ? 1 : 0;
                end
        end

        assign pixel = pixel_on ? (warn ? COLOR_WARN : (safe ? COLOR_SAFE : COLOR)) :
BLACK;
        assign warn1 = warn;
endmodule
```

User Interface Menu

```
module guimenu(clock,hcount,vcount,mx,my,mouse_click,mode,pixel);
       input clock;
       input[10:0] hcount;
       input[9:0] vcount;
       input[11:0] mx,my;
       input[2:0] mouse_click;
       output[2:0] mode;
       output[23:0] pixel;

       reg[10:0] hcount_p[1:0];
       reg[9:0] vcount_p[1:0];

       reg[2:0] mode = 0;
       reg[23:0] pixel_out;
       parameter BUTTON_COLOR = 24'b111111111000000000000000;
       parameter TABLE_COLOR = 24'b111111111100000010000000;
       parameter MOUSE_COLOR = 24'b111111110000000000000000;
       parameter BACKGROUND_COLOR = 24'b111111111101111110111111;
       parameter BLACK = 24'b000000000000000000000000;
       parameter MAX_DISPLACEMENT = 100;
       parameter MAX_DISP_COUNTER = 300000;
       reg mouse_clicked = 0;
       reg[18:0] displacement_counter = MAX_DISP_COUNTER;
       reg[6:0] displacement = 0;

       wire[23:0] cstring_calibrate_button = "Cal";
       wire[23:0] cal_pixel;
       reg[23:0] cal_pixel_p[2:1];
       char_string_display calibrate_button(clock,hcount,vcount,
                           cal_pixel,cstring_calibrate_button,940+displacement,275);
       defparam calibrate_button.COLOR = BUTTON_COLOR;
       defparam calibrate_button.NCHAR = 3;
       defparam calibrate_button.NCHAR_BITS = 2;

       wire[23:0] cstring_set_button = "Set";
       wire[23:0] set_pixel;
       reg[23:0] set_pixel_p[2:1];
       char_string_display set_button(clock,hcount,vcount,
                           set_pixel,cstring_set_button,940+displacement,375);
       defparam set_button.COLOR = BUTTON_COLOR;
       defparam set_button.NCHAR = 3;
       defparam set_button.NCHAR_BITS = 2;

       wire[23:0] cstring_hid_button = "Hid";
       wire[23:0] hid_pixel;
       reg[23:0] hid_pixel_p[2:1];
       char_string_display hid_button(clock,hcount,vcount,
                           hid_pixel,cstring_hid_button,940+displacement,475);
       defparam hid_button.COLOR = BUTTON_COLOR;
       defparam hid_button.NCHAR = 3;
       defparam hid_button.NCHAR_BITS = 2;

       wire [7:0] d13,d12,d11,d21,d22,d23,d31,d32,d33;
       binarytodec my_conv1 (clock,{6'b000000,mode},d13,d12,d11);
       binarytodec my_conv2 (clock,mx[8:0],d23,d22,d21);
       binarytodec my_conv3 (clock,my[8:0],d33,d32,d31);
       wire[71:0] cstring_mode = {d33,d32,d31,
                                                                 8'b00000000,
                                                                 d23,d22,d21,
                                                                 8'b00000000,
                                                                 d11};
       wire[23:0] mode_pixel;
       reg[23:0] mode_pixel_p[2:1];
       char_string_display mode_display(clock,hcount,vcount,
                           mode_pixel,cstring_mode,850,730);
       defparam mode_display.COLOR = 24'b111001010000000001100110;
```

```
        defparam mode_display.NCHAR = 9;
        defparam mode_display.NCHAR_BITS = 4;

        wire[11:0] LINE_0_X = 920+displacement;
        wire[11:0] LINE_1_X = 925+displacement;
        parameter LINE_TOP = 230;
        parameter LINE_0_Y = 235;
        parameter LINE_1_Y = 335;
        parameter LINE_2_Y = 435;
        parameter LINE_3_Y = 535;
        parameter LINE_BOTTOM = 540;

        reg in_table;
        reg use_font;
        reg fill_bdg;

        always @ (posedge clock) begin
                cal_pixel_p[1] <= cal_pixel;
                set_pixel_p[1] <= set_pixel;
                hid_pixel_p[1] <= hid_pixel;
                mode_pixel_p[1] <= mode_pixel;

                cal_pixel_p[2] <= cal_pixel_p[1];
                set_pixel_p[2] <= set_pixel_p[1];
                hid_pixel_p[2] <= hid_pixel_p[1];
                mode_pixel_p[2] <= mode_pixel_p[1];

                hcount_p[0] <= hcount;
                vcount_p[0] <= vcount;
                hcount_p[1] <= hcount_p[0];
                vcount_p[1] <= vcount_p[0];

                in_table <= ((hcount_p[1]>=LINE_0_X && hcount_p[1]<=LINE_1_X &&
vcount_p[1]>=LINE_TOP && vcount_p[1]<=LINE_BOTTOM) ||
                                                        (hcount_p[1]>=LINE_0_X &&
vcount_p[1]>=LINE_TOP && vcount_p[1]<=LINE_0_Y) ||
                                                        (hcount_p[1]>=LINE_0_X &&
vcount_p[1]>=LINE_3_Y && vcount_p[1]<=LINE_BOTTOM) ||
                                                        ((vcount_p[1]==LINE_1_Y ||
vcount_p[1]==LINE_2_Y) && hcount_p[1]>=LINE_0_X));

                use_font <= (cal_pixel_p[1] || set_pixel_p[1] || hid_pixel_p[1] ||
mode_pixel_p[1]);

                fill_bdg <= (hcount_p[1]>LINE_1_X && vcount_p[1]>LINE_0_Y &&
vcount_p[1]<LINE_3_Y);

                pixel_out <=  in_table ?
                                                                        TABLE_COLOR :
                                                                        use_font ?

        (cal_pixel_p[2] | set_pixel_p[2] | hid_pixel_p[2] | mode_pixel_p[2]) :

        fill_bdg ?

        BACKGROUND_COLOR :

        BLACK;

                if (mouse_clicked) begin
                        if (displacement_counter==MAX_DISP_COUNTER) begin
                                displacement_counter <= 0;
                                if (displacement!=MAX_DISPLACEMENT)
                                        displacement <= displacement + 1;
                        end else
                                displacement_counter <= displacement_counter + 1;
                end else begin
                        if (displacement_counter==MAX_DISP_COUNTER) begin
                        displacement_counter <= 0;
                        if (displacement!=0)
                              displacement <= displacement - 1;
                  end else
                        displacement_counter <= displacement_counter + 1;
            end

            if (mouse_click[2]) begin
                  if (mx>=LINE_1_X && my>=LINE_0_Y && my<=LINE_1_Y)
                        mode <= 1;
                  else if (mx>=LINE_1_X && my>=LINE_1_Y && my<=LINE_2_Y)
                        mode <= 2;
                  else if (mx>=LINE_1_X && my>=LINE_2_Y && my<=LINE_3_Y)
                        mode <= 3;
                  else
                        mode <= mode;

                  mouse_clicked <= 1'b1;
            end else if (mouse_click[0]) begin
                  mode <= 0;
                  mouse_clicked <= 1'b0;
            end
      end

      assign pixel = pixel_out;
endmodule
```

Siren Generation

```
module sirengenerator(clock,sound_on,sound_output);
        input clock;
        input sound_on;
        output sound_output;

        reg[17:0] duration;
        reg[17:0] counter;
        reg signal;
        parameter MAXCOUNT = 60000; //1300 Hz
        parameter MAXDURATION = 150; // Fraction of a second

        // Generates a 700Hz signal
        always @ (posedge clock) begin
                if (sound_on) begin
                        duration <= 0;
                        counter <= 0;
                end else begin
                        if (counter==MAXCOUNT) begin
                                if (duration!=MAXDURATION) begin
                                        signal <= ~signal;
                                        counter <= 0;
                                        duration <= duration + 1;
                                end
                        end else
                                counter <= counter + 1;
                end
        end

        assign sound_output = signal;
endmodule
```

Sweeper Line Module

```
module sweeper(vsync,clock,hcount,vcount,rspeed_in,stop,debug,
                                        x1,y1,
                                        angle_out_1,
                                        x2,y2,
                                        angle_out_2,
                                        pie,angle_loc,
                                        pixel);

        input[10:0] hcount;
        input[9:0] vcount;
        input[3:0] rspeed_in;  // angular speed
        input vsync;                 // vertical sync 65MHz
        input stop;                         // stop spinning signal
        input debug;                 // debug signal
        input clock;                 // 65MHz clock

        input[11:0] x1,y1;
        output[6:0] angle_out_1;      // Angle output

        input[11:0] x2,y2;
        output[6:0] angle_out_2;      // Angle output

        output pie;                          // Boolean: in pie?
        output[6:0] angle_loc; // Indicates location in pie

        output[23:0] pixel;    // output pixel

        parameter COLOR = 24'b000000001111111100000000;      // GREEN
        parameter BLACK = 24'b000000000000000000000000;      // BLACK
        parameter HPIXELC = 512;
        parameter VPIXELC = 384;
        parameter RSQUARED = 116964;
        parameter DELTA_THETA = 5;
        parameter DELTA_PIE = 35;

        reg[6:0] angle_out_1;                              // Angle out for X1,Y1 pair
        reg[6:0] angle_out_2;
        reg[18:0] counter = 0;                             // RANGE: 0-524287
        reg[17:0] addr = 0;                                        // RANGE: 0-196608
        reg[8:0] thetalead = 0;                                    // RANGE: 0-360
degrees
        reg[8:0] thetamiddle = DELTA_THETA;   // RANGE: 0-360 degrees
        reg[8:0] thetafollow = DELTA_PIE;     // RANGE: 0-360 degrees
        reg pixel_out;                                                     // pixel_out
boolean
        reg pie;
        // boolean: in pie?
        reg[6:0] angle_loc;                                        // location in pie

        // BRAM arc tan lookup table
        wire[6:0] dout; // 0-90 degrees output
        bramtriglookup triglookuptable (addr,clock,dout);

        // DEFINITION OF QUADRANTS
        // 2 | 1
        // - - -
        // 3 | 4

        always @ (posedge clock) begin
                if (hcount==x1 && vcount==y1) begin
                        if (hcount>=HPIXELC && vcount<VPIXELC)
                                angle_out_1 <= (90 - dout);
                        else if (hcount<HPIXELC && vcount<VPIXELC)
                                angle_out_1 <= (90 - dout);
                        else if (hcount<HPIXELC && vcount>=VPIXELC)
                                angle_out_1 <= (90 + dout);
```

```
			else if (hcount>=HPIXELC && vcount>=VPIXELC)
				angle_out_1 <= (90 + dout);
		end

		if (hcount==x2 && vcount==y2) begin
			if (hcount>=HPIXELC && vcount<VPIXELC)
				angle_out_2 <= (90 - dout);
			else if (hcount<HPIXELC && vcount<VPIXELC)
				angle_out_2 <= (90 - dout);
			else if (hcount<HPIXELC && vcount>=VPIXELC)
				angle_out_2 <= (90 + dout);
			else if (hcount>=HPIXELC && vcount>=VPIXELC)
				angle_out_2 <= (90 + dout);
		end

		if (counter==524287) begin
			if(!stop) begin
			      thetalead <= (thetalead>rspeed_in) ? (thetalead-rspeed_in) :
(360+thetalead-rspeed_in);
				thetamiddle <= (thetamiddle>rspeed_in) ? (thetamiddle-
rspeed_in) : (360+thetamiddle-rspeed_in);
				thetafollow <= (thetafollow>rspeed_in) ? (thetafollow-
rspeed_in) : (360+thetafollow-rspeed_in);
			end
			counter <= 0;
		end else
			counter <= counter + 1;

		// Quadrant 1
		if (hcount>=HPIXELC && vcount<VPIXELC) begin
			addr <= (hcount - HPIXELC) + vcount*HPIXELC;

			pixel_out <= (thetalead>thetafollow) ?
								((dout>=0 &&
dout<=thetafollow) &&

	((hcount-HPIXELC)*(hcount-HPIXELC)+(VPIXELC-vcount)*(VPIXELC-vcount))<=RSQUARED) :

	((dout>=thetalead && dout<=thetafollow) &&

	((hcount-HPIXELC)*(hcount-HPIXELC)+(VPIXELC-vcount)*(VPIXELC-vcount))<=RSQUARED);

			pie <= (thetamiddle>thetafollow) ?
								((dout>=0 &&
dout<=thetafollow) &&

	((hcount-HPIXELC)*(hcount-HPIXELC)+(VPIXELC-vcount)*(VPIXELC-vcount))<=RSQUARED) :

	((dout>=thetamiddle && dout<=thetafollow) &&

	((hcount-HPIXELC)*(hcount-HPIXELC)+(VPIXELC-vcount)*(VPIXELC-vcount))<=RSQUARED);
			angle_loc <= (thetafollow - dout);

		// Quadrant 2
		end else if (hcount<HPIXELC && vcount<VPIXELC) begin
			addr <= (HPIXELC - (hcount + 1)) + vcount*HPIXELC;
			pixel_out <= (thetalead>thetafollow) ?
								((180 -
dout)>=0 && (180-dout)<=thetafollow &&

	((HPIXELC-hcount)*(HPIXELC-hcount)+(VPIXELC-vcount)*(VPIXELC-vcount))<=RSQUARED) :
								((180 -
dout)>=thetalead && (180 - dout)<=thetafollow &&

	((HPIXELC-hcount)*(HPIXELC-hcount)+(VPIXELC-vcount)*(VPIXELC-vcount))<=RSQUARED);
			pie <= (thetamiddle>thetafollow) ?
								((180 -
dout)>=0 && (180 - dout)<=thetafollow &&
        ((HPIXELC-hcount)*(HPIXELC-hcount)+(VPIXELC-vcount)*(VPIXELC-vcount))<=RSQUARED) :
                                                                                ((180 -
dout)>=thetamiddle && (180 - dout)<=thetafollow &&

        ((HPIXELC-hcount)*(HPIXELC-hcount)+(VPIXELC-vcount)*(VPIXELC-vcount))<=RSQUARED);
                        angle_loc <= (thetafollow - (180 - dout));

                // Quadrant 3
                end else if (hcount<HPIXELC && vcount>=VPIXELC) begin
                        addr <= (HPIXELC - (hcount + 1)) + (2*VPIXELC - (vcount +
1))*HPIXELC;
                        pixel_out <= (thetalead>thetafollow) ?
                                                                                ((180 +
dout)>=0 && (180 + dout)<=thetafollow &&

        ((HPIXELC-hcount)*(HPIXELC-hcount)+(vcount-VPIXELC)*(vcount-VPIXELC))<=RSQUARED) :
                                                                                ((180 +
dout)>=thetalead && (180 + dout)<=thetafollow &&

        ((HPIXELC-hcount)*(HPIXELC-hcount)+(vcount-VPIXELC)*(vcount-VPIXELC))<=RSQUARED);
                        pie <= (thetamiddle>thetafollow) ?
                                                                                ((180 +
dout)>=0 && (180 + dout)<=thetafollow &&

        ((HPIXELC-hcount)*(HPIXELC-hcount)+(vcount-VPIXELC)*(vcount-VPIXELC))<=RSQUARED) :
                                                                                ((180 +
dout)>=thetamiddle && (180 + dout)<=thetafollow &&

        ((HPIXELC-hcount)*(HPIXELC-hcount)+(vcount-VPIXELC)*(vcount-VPIXELC))<=RSQUARED);
                        angle_loc <= (thetafollow - (180 + dout));

                // Quadrant 4
                end else if (hcount>=HPIXELC && vcount>=VPIXELC) begin
                        addr <= (hcount - HPIXELC) + (2*VPIXELC - (vcount + 1))*HPIXELC;
                        pixel_out <= (thetalead>thetafollow) ?
                                                                                ((360 -
dout)>=0 && (360 - dout)<=thetafollow &&

        ((hcount-HPIXELC)*(hcount-HPIXELC)+(vcount-VPIXELC)*(vcount-VPIXELC))<=RSQUARED) :
                                                                                ((360 -
dout)>=thetalead && (360 - dout)<=thetafollow &&

        ((hcount-HPIXELC)*(hcount-HPIXELC)+(vcount-VPIXELC)*(vcount-VPIXELC))<=RSQUARED);
                        pie <= (thetamiddle>thetafollow) ?
                                                                                ((360 -
dout)>=0 && (360 - dout)<=thetafollow &&

        ((hcount-HPIXELC)*(hcount-HPIXELC)+(vcount-VPIXELC)*(vcount-VPIXELC))<=RSQUARED) :
                                                                                ((360 -
dout)>=thetamiddle && (360 - dout)<=thetafollow &&

        ((hcount-HPIXELC)*(hcount-HPIXELC)+(vcount-VPIXELC)*(vcount-VPIXELC))<=RSQUARED);
                        angle_loc <= (thetafollow - (360 - dout));
                end else begin
                        addr <= 18'b000000000000000000;
                        pixel_out <= 1'b0;
                end
        end

        assign pixel = pixel_out ? COLOR : BLACK;
endmodule
```

String display module

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

   parameter NCHAR = 8;       // number of 8-bit characters in cstring
   parameter NCHAR_BITS = 3; // number of bits in NCHAR
       parameter COLOR = 24'b000000001111111100000000;

   input vclock;      // 65MHz clock
   input [10:0] hcount;       // horizontal index of current pixel (0..1023)
   input [9:0]        vcount; // vertical index of current pixel (0..767)
   output [23:0] pixel;       // char display's pixel
   input [NCHAR*8-1:0] cstring;       // character string to display
   input [10:0] cx;
   input [9:0]        cy;

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

   wire [10:0]        hoff = hcount-1-cx;
   wire [9:0]  voff = vcount-cy;
   wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+4:4];  // < NCHAR
   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 @ (posedge vclock) begin
     for (n=0 ; n<8 ; n = n+1 )              // 8 bits per character (ASCII)
       char[n] <= cstring[column*8+n];
        end

   // 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 [23:0] cpixel = (font_byte[7 - h] ^ reverse) ? COLOR : 0;
   wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <= cx+NCHAR*16) & (vcount <
cy + 24));
   wire [23:0] pixel = dispflag ? cpixel : 0;

endmodule
```

Alarm Generation Module

```
module alarm_gen(clock,alarm_on,alarm_out);
        input clock;
        input alarm_on;
        output alarm_out;

        reg chooseA;
        reg signal;
        reg[17:0] counter;
        reg[10:0] duration;
        parameter MAX_DURATION = 300;
        parameter freqA = 162500;
        parameter freqB = 92000;

        always @ (posedge clock) begin
                if(alarm_on) begin
                        if (counter == (chooseA ? freqA : freqB)) begin
                                signal <= ~signal;
                                counter <= 0;
                                duration <= duration + 1;
                                if (duration == MAX_DURATION) begin
                                        duration <= 0;
                                        chooseA <= ~chooseA;
                                end else
                                        duration <= duration + 1;
                        end else
                                counter <= counter + 1;
                end
        end

        assign alarm_out = signal;
endmodule
```

Sonar Data Acquisition

```
module sonar_data_acquisition (clock_65mhz, reset,
                               thirty_hz_enable, one_mhz_enable, // start was
thirty_hz_enable
                               adc_in, adc_out,
                               pulse_output, switch,
                               hex_display_data,
                               analyzer1_data, analyzer1_clock,
                               analyzer3_data, analyzer3_clock,
                               bram_address, bram_data, bram_write_enable);

   input clock_65mhz, reset;
   output thirty_hz_enable;
   output one_mhz_enable;
   input [11:0] adc_in;
   output [11:0] adc_out;
   output [1:0]  pulse_output;
   input  [7:0]  switch;
   output reg [63:0] hex_display_data;
   output [15:0] analyzer1_data;
   output        analyzer1_clock;
   output [15:0] analyzer3_data;
   output        analyzer3_clock;

   output reg [14:0] bram_address;
   output [11:0] bram_data;
   output reg bram_write_enable;

   wire        trigger;
   wire signed [20:0] signal_threshold;
   wire        [11:0] echo_detect;
   wire signed [15:0] waveform_data [11:0];
   wire        out_p, out_m;


   reg signed [15:0] analyzer_waveform;
   reg [3:0]     analyzer_index;

   assign     trigger = thirty_hz_enable && switch[7];
   assign     analyzer1_data = {thirty_hz_enable, one_mhz_enable, out_p, out_m,
echo_detect};
   assign     analyzer1_clock = clock_65mhz;
   assign     analyzer3_data = analyzer_waveform;
   assign     analyzer3_clock = clock_65mhz;
   assign     bram_data = echo_detect;
   assign     pulse_output = {out_p, out_m};

   // Generating 30Hz enable signal:  64,800,000 / 2,160,000 = 30
   clock_divider clkdiv1(clock_65mhz, reset, 25'd2_160_000, thirty_hz_enable);
   // Generating 1MHz enable signal:  64,800,000 / 64 = 1,0125,000
   // (not really 1MHz, but we can compensate within the distance and sampling routines)
   clock_divider clkdiv2(clock_65mhz, reset, 25'd64, one_mhz_enable);

   threshold_generation thld0(clock_65mhz, reset, trigger, one_mhz_enable,
signal_threshold);

   pulse_40khz_out ch1_out(clock_65mhz, reset, trigger, out_p, out_m);

   channel_pipeline scp0 (clock_65mhz, reset, one_mhz_enable, adc_in[0],  adc_out[0],
                          signal_threshold, switch[5:4], echo_detect[0],
waveform_data[0][15:0]);

   channel_pipeline scp1 (clock_65mhz, reset, one_mhz_enable, adc_in[1],  adc_out[1],
                          signal_threshold, switch[5:4], echo_detect[1],
waveform_data[1][15:0]);

   channel_pipeline scp2 (clock_65mhz, reset, one_mhz_enable, adc_in[2],  adc_out[2],
                           signal_threshold, switch[5:4], echo_detect[2],
waveform_data[2][15:0]);

   channel_pipeline scp3 (clock_65mhz, reset, one_mhz_enable, adc_in[3],  adc_out[3],
                           signal_threshold, switch[5:4], echo_detect[3],
waveform_data[3][15:0]);

   channel_pipeline scp4 (clock_65mhz, reset, one_mhz_enable, adc_in[4],  adc_out[4],
                           signal_threshold, switch[5:4], echo_detect[4],
waveform_data[4][15:0]);

   channel_pipeline scp5 (clock_65mhz, reset, one_mhz_enable, adc_in[5],  adc_out[5],
                           signal_threshold, switch[5:4], echo_detect[5],
waveform_data[5][15:0]);

   channel_pipeline scp6 (clock_65mhz, reset, one_mhz_enable, adc_in[6],  adc_out[6],
                           signal_threshold, switch[5:4], echo_detect[6],
waveform_data[6][15:0]);

   channel_pipeline scp7 (clock_65mhz, reset, one_mhz_enable, adc_in[7],  adc_out[7],
                           signal_threshold, switch[5:4], echo_detect[7],
waveform_data[7][15:0]);

   channel_pipeline scp8 (clock_65mhz, reset, one_mhz_enable, adc_in[8],  adc_out[8],
                           signal_threshold, switch[5:4], echo_detect[8],
waveform_data[8][15:0]);

   channel_pipeline scp9 (clock_65mhz, reset, one_mhz_enable, adc_in[9],  adc_out[9],
                           signal_threshold, switch[5:4], echo_detect[9],
waveform_data[9][15:0]);

   channel_pipeline scp10(clock_65mhz, reset, one_mhz_enable, adc_in[10], adc_out[10],
                           signal_threshold, switch[5:4],
echo_detect[10],waveform_data[10][15:0]);

   channel_pipeline scp11(clock_65mhz, reset, one_mhz_enable, adc_in[11], adc_out[11],
                           signal_threshold, switch[5:4],
echo_detect[11],waveform_data[11][15:0]);


   always @ (switch[3:0]) begin
        case (switch[3:0])
          0:  begin analyzer_waveform = waveform_data[0][15:0];  analyzer_index =
switch[3:0]; end
          1:  begin analyzer_waveform = waveform_data[1][15:0];  analyzer_index =
switch[3:0]; end
          2:  begin analyzer_waveform = waveform_data[2][15:0];  analyzer_index =
switch[3:0]; end
          3:  begin analyzer_waveform = waveform_data[3][15:0];  analyzer_index =
switch[3:0]; end
          4:  begin analyzer_waveform = waveform_data[4][15:0];  analyzer_index =
switch[3:0]; end
          5:  begin analyzer_waveform = waveform_data[5][15:0];  analyzer_index =
switch[3:0]; end
          6:  begin analyzer_waveform = waveform_data[6][15:0];  analyzer_index =
switch[3:0]; end
          7:  begin analyzer_waveform = waveform_data[7][15:0];  analyzer_index =
switch[3:0]; end
          8:  begin analyzer_waveform = waveform_data[8][15:0];  analyzer_index =
switch[3:0]; end
          9:  begin analyzer_waveform = waveform_data[9][15:0];  analyzer_index =
switch[3:0]; end
          10: begin analyzer_waveform = waveform_data[10][15:0]; analyzer_index =
switch[3:0]; end
          11: begin analyzer_waveform = waveform_data[11][15:0]; analyzer_index =
switch[3:0]; end
          default: begin analyzer_waveform = signal_threshold[20:5]; analyzer_index =
switch[3:0]; end
```

```
        endcase // case(switch[3:0])
   end // always @ (switch[3:0])

   always @ (posedge clock_65mhz)
     if (reset) bram_address <= 0;
     else begin
        if (trigger) begin
           hex_display_data <= {40'b0, analyzer_index, 4'b0, analyzer_waveform};
           bram_address <= 0;
        end else if (one_mhz_enable && bram_address < 15'h7FFF) begin
           bram_address <= bram_address + 1;
           bram_write_enable <= 1;
        end else bram_write_enable <= 0;
     end

endmodule // sonar_data_acquisition //////////////
///////////////////////////////////////////////////

module pulse_40khz_out(clock_65mhz, reset_sync, trigger, out_p, out_m);
   // This module's output frequency is customized to generate a 40kHz output when
   // driven by a 64.8MHz clock; the 25us period is equal to 1620 clock cycles.
   // out_p and out_m are differential signals that are 120 degrees out of phase, their
   // difference is a simple 2-bit stepped aproximation of a sinewave: (0,1,1,0,-1,-
1,repeat)
   input clock_65mhz, reset_sync;
   input trigger;
   output reg out_p, out_m;
   reg [10:0] wave_counter;
   reg [4:0] pulse_counter;
   always @ (posedge clock_65mhz) begin
      if (reset_sync) begin
           wave_counter <= 0;
           pulse_counter <= 0;
           out_p <= 0;
           out_m <= 0;
      end else if (trigger) begin
           pulse_counter <= 16;
           wave_counter <= 0;
      end else if (pulse_counter !=0) begin
         if (wave_counter == 1619)
           begin
              wave_counter <= 0;
              pulse_counter <= pulse_counter - 1;
           end else begin
              wave_counter <= wave_counter + 1;
              if      ( wave_counter == 0   ) begin out_p <= 0; out_m <= 1; end
              else if ( wave_counter == 540 ) begin out_p <= 1; out_m <= 1; end
              else if ( wave_counter == 810 ) begin out_p <= 1; out_m <= 0; end
              else if ( wave_counter == 1350) begin out_p <= 0; out_m <= 0; end
              else begin out_p <= out_p; out_m <= out_m; end
           end // else: !if(wave_counter == 1619)
      end // if (pulse_counter !=0)
   end // always @ (posedge clock_65mhz)
endmodule // pulse_40khz_out

module clock_divider (clock, reset, divisor, pulse_enable);
   input clock, reset;
   input [24:0] divisor;
   output reg pulse_enable;
   reg [24:0] count;  // 25-bit register, values up to 2^25 (>32,000,000)
   always @ (posedge clock)
     if (reset) count <= divisor;
     else
       if (count == 0) begin count <= divisor;   pulse_enable <= 1; end
       else            begin count <= count - 1; pulse_enable <= 0; end
endmodule // clock_divider

module delta_sigma_buffer(clock, input_pin, output_pin, buffer_reg);
```

```
   input clock;
   input input_pin;
   output reg output_pin, buffer_reg;
   reg    a, b;
   always @ (posedge clock) begin
      a <= input_pin;
      b <= a;
      output_pin <= b;
      buffer_reg <= b;
   end
endmodule // delta_sigma_buffer

module adc_channel (clock, reset, adc_input_pin, adc_output_pin, data_out);
   /// In the current form (12-6-07) the module is agnostic about the clock rate
   /// It should work as well at 64.8 as at 27MHz
   input clock, reset, adc_input_pin;
   output adc_output_pin;
   output signed [15:0] data_out;
   wire   adc_sample;

   reg signed [31:0] running_sum;
   reg signed [31:0] epsilon;
   reg signed [16:0] up_down_counter;
   reg  [15:0] count_16bit;
   wire [16:0] up_down_delta;
   wire [31:0] running_sum_delta;

   delta_sigma_buffer dsbuf1(clock, adc_input_pin, adc_output_pin, adc_sample);

   assign      up_down_delta = adc_sample ? 17'h00001 : 17'hFFFFF;
   assign      running_sum_delta = adc_sample ? 32'h0001_0000 : 32'hFFFF_0000;
   assign      data_out = running_sum[26:11]; // cut off the 5 high order bits, there is
no data there

   always @ (posedge clock)
     begin
        if (reset)
          begin
             running_sum <= 0;
             epsilon <= 0;
             up_down_counter <= 0;
          end else begin
             running_sum <= running_sum + running_sum_delta - (epsilon >>> 6) -
(running_sum >>> 16); // unaliased
             count_16bit <= count_16bit + 1;
             if (count_16bit == 0)
               begin
                  epsilon <= (epsilon - (epsilon >>> 6)) + up_down_counter;
                  up_down_counter <= up_down_delta;
               end
             else up_down_counter <= up_down_counter + up_down_delta;
          end
     end // always @ (posedge clock)
endmodule // adc_channel

module lpf_iir ( clock, reset, enable, data_in, data_out);
   // Simple one pole infinite impulse response low-pass filter
   parameter SHIFT_FACTOR = 6;
   input  clock, reset, enable;
   input  signed [15:0] data_in;
   output signed [15:0] data_out;
   reg    signed [31:0] xminus0;
   reg    signed [31:0] yminus0;
   reg    signed [31:0] yminus1;
   // data_out truncates the top bit, effectively scaling the output by 2:
   assign data_out = yminus0[30:15];
   always @ (posedge clock)
     if (enable) begin
```

```
        xminus0 <= {data_in, 16'b0};
        yminus1 <= yminus0;
        yminus0 <= yminus1 - ( yminus1 >>> SHIFT_FACTOR) + ( xminus0 >>> SHIFT_FACTOR);
     end
endmodule // lpf_iir

module hpf_iir ( clock, reset, enable, data_in, data_out);
   // Simple one pole infinite impulse response high-pass filter
   parameter SHIFT_FACTOR = 10;
   input  clock, reset, enable;
   input  signed [15:0] data_in;
   output signed [15:0] data_out;
   reg    signed [31:0] xminus0;
   reg    signed [31:0] xminus1;
   reg    signed [31:0] yminus0;
   reg    signed [31:0] yminus1;
   // data_out truncates the top bit, effectively scaling the output by 2:
   assign data_out = yminus0[30:15];
   always @ (posedge clock)
     if (enable) begin
        xminus0 <= {data_in, 16'b0};
        xminus1 <= xminus0;
        yminus1 <= yminus0;
        yminus0 <= yminus1 - ( yminus1 >>> SHIFT_FACTOR) + ( xminus0 - xminus1);
     end
endmodule // hpf_iir

module triangle_window_fir( clock, reset, enable, data_in, data_out);
   // triangle_window_fir produces an output value that is scaled up
   // by a factor of 32 from the input value ( 1+3+5+7+7+5+3+1 = 32).
   // It is left to the specific instance to scale the output down or to
   // keep the full 21-bit value (the sonar code does both)
   input  clock, reset, enable;
   input  signed [15:0] data_in;
   output signed [20:0] data_out;
   reg    signed [20:0] shift_buffer [7:0]; // 21 bit-wide, 8 word buffer
   wire   signed [20:0] a_x [3:0]; // current input value multiplied by the FIR
coefficients

   // sign extend data_in from 16 to 21 bits and assign to a_x[0]:
   assign a_x[0][20:0] = $signed(data_in[15:0]);
   assign a_x[1][20:0] = (a_x[0][20:0] <<< 1) + a_x[0][20:0]; // data_in * 3
   assign a_x[2][20:0] = (a_x[0][20:0] <<< 2) + a_x[0][20:0]; // data_in * 5
   assign a_x[3][20:0] = (a_x[0][20:0] <<< 3) - a_x[0][20:0]; // data_in * 7
   assign data_out = shift_buffer[7][20:0];
   always @ (posedge clock)
     if (enable) begin
        shift_buffer[0][20:0] <= a_x[0][20:0];                         // 1 * data_in
        shift_buffer[1][20:0] <= a_x[1][20:0] + shift_buffer[0][20:0]; // 3 * data_in +
buffer
        shift_buffer[2][20:0] <= a_x[2][20:0] + shift_buffer[1][20:0]; // 5 * data_in +
buffer
        shift_buffer[3][20:0] <= a_x[3][20:0] + shift_buffer[2][20:0]; // 7 * data_in +
buffer
        shift_buffer[4][20:0] <= a_x[3][20:0] + shift_buffer[3][20:0]; // 7 * data_in +
buffer
        shift_buffer[5][20:0] <= a_x[2][20:0] + shift_buffer[4][20:0]; // 5 * data_in +
buffer
        shift_buffer[6][20:0] <= a_x[1][20:0] + shift_buffer[5][20:0]; // 3 * data_in +
buffer
        shift_buffer[7][20:0] <= a_x[0][20:0] + shift_buffer[6][20:0]; // 1 * data_in +
buffer
     end
endmodule // triangle_window_fir

module bpf_65mhz (clock, reset, data_in, data_out);
   // The bpf_65mhz module prepares raw delta-sigma output for 1MHz downsampling:
   // blocking DC bias components and low-pass anti-aliasing by
   // chaining 2 iir filters and one fir (low-high-low)
   input  clock, reset;
   input  signed [15:0] data_in;
   output signed [15:0] data_out;
   wire [15:0] a, b;
   wire [20:0] fir_data;

   assign      data_out[15:0] = fir_data[20:5];
   lpf_iir filter1(clock, reset, 1'b1, data_in, a);
   hpf_iir filter2(clock, reset, 1'b1, a, b);
   triangle_window_fir filter4(clock, reset, 1'b1, b, fir_data);
endmodule // bpf_65mhz

module downsample(clock, reset, latch_sample, data_in, data_out);
   // Downsampling filter:  this module samples the filtered adc data at 1MHz,
   // then applies one additional stage each of DC blocking and anti-aliasing
   // to condition the data before threshold detection logic.
   input  clock, reset, latch_sample;
   input  signed [15:0] data_in;
   output signed [20:0] data_out;
   wire [15:0] a;

   hpf_iir #(6) filter1(clock, reset, latch_sample, data_in, a);
   // all 21 bits from fir are passed to the output:
   triangle_window_fir filter3(clock, reset, latch_sample, a, data_out);
endmodule // downsamp

module threshold_generation(clock, reset, thirty_hz_reset, one_mhz_tick, threshold);
   // threshold generation works as of version 11 of this file:  labkit_sonar_11.v
   parameter MAX_THRESHOLD = 131072; // the upper bound threshold value -  2^17 = 131072
   parameter MIN_THRESHOLD = 4096;   // the lower bound threshold value -  2^11 = 2048
   parameter TXRX_DELAY =  512;  // account for the direct-field transmitter - receiver
distance: 512usec
   parameter T_MAX_THLD = 4096;  // time in usec when threshold falls below maximum

   input  clock, reset;
   input  thirty_hz_reset;
   input  one_mhz_tick;
   output reg signed [20:0] threshold;
   reg  [15:0] mhz_tcounter;   // tick_counter can go up to 2^16, but will always be
reset just above 2^15
   wire [15:0] threshold_time_parameter;
   wire [31:0] threshold_tempvar;
   // Coregen generated divider module:
   // module threshold_divider( clk, dividend, divisor, quotient, remainder, rfd);
   //   input  clk;
   //   input  [31:0] dividend;
   //   input  [15:0] divisor;
   //   output [31:0] quotient;
   //   output [15:0] remainder;
   //   output        rfd;
   wire [31:0] dividend;
   wire [15:0] remainder;
   wire [15:0] divisor;
   wire        rfd;
   assign dividend = MAX_THRESHOLD * (T_MAX_THLD >> 7) * (T_MAX_THLD >> 7);   // This
line should be executed at compile time
   assign divisor = threshold_time_parameter * threshold_time_parameter;  // The single
multiplier in the code?
   assign threshold_time_parameter = (mhz_tcounter > (TXRX_DELAY + T_MAX_THLD)) ?
                                    ((mhz_tcounter - TXRX_DELAY) >> 7) : (T_MAX_THLD >>
7);

   // Divisor is the only variable input to the divider module.
   // It is generated by a continuous assignment with an input that updates every 64
clock cylces.
   // The value is then held for 64 clock cycles before the next mhz tick,
   // at which point the quotient is latched and a new divisor value is presented.
   // remainder and rfd values are discarded
   threshold_divider divider0( clock, dividend, divisor, threshold_tempvar, remainder,
rfd);

   always @ (posedge clock)
     if (reset || thirty_hz_reset) begin
        mhz_tcounter <= 0;
        threshold <= MAX_THRESHOLD;
     end else if (one_mhz_tick ) begin
        if ( mhz_tcounter < 16'hFFFF )
          mhz_tcounter <= mhz_tcounter + 1;
        if ( threshold_tempvar > MIN_THRESHOLD )
          threshold <= $signed(threshold_tempvar[20:0]);
        else threshold <= MIN_THRESHOLD;
     end
endmodule // threshold_generation

module threshold_detect(clock, reset, one_mhz_enable, threshold, data_in, detect);
   input clock, reset;
   input one_mhz_enable;
   input signed [20:0] threshold;
   input signed [20:0] data_in;
   output detect;
   reg [3:0] counter;

   assign detect = (counter > 0);

   always @ (posedge clock)
     if (reset) counter <= 0;
     else if (one_mhz_enable)
       if ( (counter == 0) && (data_in - threshold > 0) )
          counter <= 13;
       else if (counter > 0) counter <= counter - 1;

endmodule // threshold_detect

// Need two modules:  sonar_channel_pipeline and sonar_data_acquistion, which groups all
12 channel pipelines
// together with the control and interface logic


module channel_pipeline (clock_65mhz, reset, mhz_enable, adc_input_pin, adc_output_pin,
                         threshold, waveform_select, pipeline_out, waveform_out);
   input clock_65mhz, reset;
   input mhz_enable;
   input adc_input_pin;
   output adc_output_pin;
   input [20:0] threshold;
   input [1:0]  waveform_select;
   output reg pipeline_out;
   output reg [15:0] waveform_out;

   wire [15:0]   deltasig_raw;
   wire [15:0]   deltasig_filt;
   wire [20:0]   downsamp_data;  // 21-bit output from the downsampling filter
   wire          signal_detect;

   // module adc_channel (clock, reset, adc_input_pin, adc_output_pin, data_out);
   // module bpf_65mhz (clock, reset, data_in, data_out);
   // module downsample(clock, reset, latch_sample, data_in, data_out);
   // module threshold_detect(clock, reset, one_mhz_enable, threshold, data_in, detect);

   adc_channel      adc (clock_65mhz, reset, adc_input_pin, adc_output_pin, deltasig_raw);
   bpf_65mhz        bpf (clock_65mhz, reset, deltasig_raw,  deltasig_filt);
   downsample       dsmp(clock_65mhz, reset, mhz_enable,    deltasig_filt,
downsamp_data);
   threshold_detect dtct(clock_65mhz, reset, mhz_enable,    threshold,      downsamp_data,
signal_detect);
```

```
   always @ (waveform_select)
     case (waveform_select)
       // pipline out gets the inverse sign bit, so that it is 1 when value is positive:
       0 : begin waveform_out = deltasig_raw;         pipeline_out = ~deltasig_raw[15];
end
       1 : begin waveform_out = deltasig_filt;        pipeline_out = ~deltasig_filt[15];
end
       2 : begin waveform_out = downsamp_data[20:5];  pipeline_out = ~downsamp_data[20];
end
       3 : begin waveform_out = downsamp_data[20:5];  pipeline_out =  signal_detect;
end
       // Cases are exhaustive, default shouldn't be necessary:
       default : begin waveform_out = downsamp_data[20:5]; pipeline_out = signal_detect;
end
     endcase // case(waveform_select)
endmodule // channel_pipeline
```

Angle Retriever

```
module angle_retriever(clk, reset, start, t1, t2, r,
       cos_theta, done_ar);

input clk, reset, start;
input [9:0] t1, t2, r;
output [9:0] cos_theta;
output done_ar;
reg done;

reg signed [9:0] dt, cos_theta_est;
reg signed [19:0] cos1;
reg [3:0] count;

// parameters
// parameter d = 10'd117; // in mm
parameter signed a2 = 10'd446;  //a2=(v/2d)*512*512
parameter cycle = 4'd8;
parameter bits=9;



// start pulse
wire start_pulse;
reg start_old;
always @ (posedge clk)
begin
start_old <= start;
end
assign start_pulse = (~start_old) & start;

// main
always @ (posedge clk)
begin //1
if (reset)
       begin
       cos_theta <= 0;
       end
else if (start)
       begin

       dt <= t2-t1;  // change here to flip x axis
       cos1 <= a2*dt;
       cos_theta_est <= (cos1 >> bits);

//      dt <= t2-t1;
//      cos11 <= one_over_d*dt;
//      cos1 <= cos11 >> bits;
//      dt_square1 <= dt * dt;
//      dt_square <= dt_square1 >> bits;
//      cos211 <= (dt_square >> 1) * one_over_d;
//      cos21 <= cos211 >> bits;
//      cos22 <= cos21 * one_over_r;
//      cos2 <= cos22 >> bits;
//      cos31 <= (d>>1) * one_over_r;
//      cos3 <= cos31 >> bits;
//      cos_theta <= cos1 + cos2 - cos3;
       end
end //-1

// look-up table
cal_lut callut(clk, r, cos_theta_est, cos_theta);

// done signal
always @ (posedge clk)
begin //2
if ((reset) | (start_pulse))
```

```
        begin
        count <= 0;
        done <= 0;
        end
else if (count == cycle)
        begin
        done <= 1'b1;
        count <= 0;
        end
else if (start) count <= count +1;
end //-2

assign done_ar = (start_pulse) ? 0 : done;

endmodule
```

Boundary Retriever

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Create Date:    19:24:46 12/02/2007
// Module Name:    boundary_retriever
//
////////////////////////////////////////////////////////////////////////////////

module boundary_retriever(clk, reset, start, r, t1_0, t2_0, t1_11, t2_11, mem_data,
               t_0, t_11, mem_pointer, done_br);
input clk, reset, start;
input [15:0] t1_0, t2_0, t1_11, t2_11;
input [9:0] r;
input [11:0] mem_data;
output [9:0] t_0, t_11;
output reg [15:0] mem_pointer;
output done_br;
reg done;


reg [11:0] t[11:0];
reg [7:0] count;
reg [11:0] shift_l, shift_r;
reg [11:0] mem_data_old;
//reg [15:0] t_ref;
wire [11:0] t_star[11:0];

// parameters
parameter period = 25;
parameter half_period = 13;
parameter sample_period = 28;
parameter shift_period = sample_period + 20;
parameter cycle = 60;


//assign t_ref = (t1_0>>2) + (t2_0>>2) + (t1_11>>2) + (t2_11>>2);

// start pulse
wire start_pulse;
reg start_old;
always @ (posedge clk)
begin
start_old <= start;
end
assign start_pulse = (~start_old) & start;

// main
always @ (posedge clk)
begin //1
if (reset)
   begin
       mem_pointer <= 0;
       count <= 0;
       shift_l <= 0;
       shift_r <= 0;
   end

else if (start)
   begin //1.2
       if (start_pulse)
              begin
              mem_pointer <= (t1_0>>2) + (t2_0>>2) + (t1_11>>2) + (t2_11>>2);
              count <=0;
              shift_l <= 0;
          shift_r <= 0;
      mem_data_old <= 12'b1111_1111_1111;
       end
```

```
        else if (count <= sample_period)
                // phase 1, read rising edge from mem
                begin //1.2.2
                count <= count + 1;
                mem_pointer <= mem_pointer + 1;
                mem_data_old <= mem_data;
                // catch the rising edge of received signal
                if (mem_data[0] & ~mem_data_old[0]) t[0] <= count + (period<<3);
                if (mem_data[1] & ~mem_data_old[1]) t[1] <= count + (period<<3);
                if (mem_data[2] & ~mem_data_old[2]) t[2] <= count + (period<<3);
                if (mem_data[3] & ~mem_data_old[3]) t[3] <= count + (period<<3);
                if (mem_data[4] & ~mem_data_old[4]) t[4] <= count + (period<<3);
                if (mem_data[5] & ~mem_data_old[5]) t[5] <= count + (period<<3);
                if (mem_data[6] & ~mem_data_old[6]) t[6] <= count + (period<<3);
                if (mem_data[7] & ~mem_data_old[7]) t[7] <= count + (period<<3);
                if (mem_data[8] & ~mem_data_old[8]) t[8] <= count + (period<<3);
                if (mem_data[9] & ~mem_data_old[9]) t[9] <= count + (period<<3);
                if (mem_data[10] & ~mem_data_old[10]) t[10] <= count + (period<<3);
                if (mem_data[11] & ~mem_data_old[11]) t[11] <= count + (period<<3);
                end //-1.2.2
        else if (count <= shift_period)
                // phase 2, determine the shifts needed
                // change this to implement "linear search-direction prediction"
     begin //1.2.3
     if  (t1_0 < t1_11)
                begin //1.2.3.1
                count <= count + 1;
                // here, shift[7] implies that all t[0] to t[7] should be shifted
                if (t[1] > t[0]+half_period) shift_r[0] <= 1; else shift_r[0] <= 0;
                if (t[1]+half_period < t[0]) shift_l[0] <= 1; else shift_l[0] <= 0;
                if (t_star[0]+t[2] > (t[1]<<1)+half_period) shift_r[1] <= 1; else
shift_r[1] <= 0;
               if (t_star[0]+t[2]+half_period <(t[1]<<1)) shift_l[1] <= 1; else shift_l[1]
<= 0;
                if (t_star[1]+t[3] > (t[2]<<1)+half_period) shift_r[2] <= 1; else
shift_r[2] <= 0;
               if (t_star[1]+t[3]+half_period <(t[2]<<1)) shift_l[2] <= 1; else shift_l[2]
<= 0;
                if (t_star[2]+t[4] > (t[3]<<1)+half_period) shift_r[3] <= 1; else
shift_r[3] <= 0;
               if (t_star[2]+t[4]+half_period <(t[3]<<1)) shift_l[3] <= 1; else shift_l[3]
<= 0;
                if (t_star[3]+t[5] > (t[4]<<1)+half_period) shift_r[4] <= 1; else
shift_r[4] <= 0;
               if (t_star[3]+t[5]+half_period <(t[4]<<1)) shift_l[4] <= 1; else shift_l[4]
<= 0;
                if (t_star[4]+t[6] > (t[5]<<1)+half_period) shift_r[5] <= 1; else
shift_r[5] <= 0;
               if (t_star[4]+t[6]+half_period <(t[5]<<1)) shift_l[5] <= 1; else shift_l[5]
<= 0;
                if (t_star[5]+t[7] > (t[6]<<1)+half_period) shift_r[6] <= 1; else
shift_r[6] <= 0;
               if (t_star[5]+t[7]+half_period <(t[6]<<1)) shift_l[6] <= 1; else shift_l[6]
<= 0;
                if (t_star[6]+t[8] > (t[7]<<1)+half_period) shift_r[7] <= 1; else
shift_r[7] <= 0;
               if (t_star[6]+t[8]+half_period <(t[7]<<1)) shift_l[7] <= 1; else shift_l[7]
<= 0;
                if (t_star[7]+t[9] > (t[8]<<1)+half_period) shift_r[8] <= 1; else
shift_r[8] <= 0;
               if (t_star[7]+t[9]+half_period <(t[8]<<1)) shift_l[8] <= 1; else shift_l[8]
<= 0;
                if (t_star[8]+t[10] > (t[8]<<1)+half_period) shift_r[9] <= 1; else
shift_r[9] <= 0;
                if (t_star[8]+t[10]+half_period <(t[9]<<1)) shift_l[9] <= 1; else
shift_l[9] <= 0;
                if (t_star[9]+t[11] > (t[10]<<1)+half_period) shift_r[10] <= 1; else
```

```
shift_r[10] <= 0;
               if (t_star[9]+t[11]+half_period <(t[10]<<1)) shift_l[10] <= 1; else
shift_l[10] <= 0;
               end //-1.2.3.1
         else
          begin //1.2.3.2
          count <= count + 1;
          // here, shift[7] implies all t[7] to t[11] should be shifted
          if (t[10] > t[11]+half_period) shift_r[11] <= 1; else shift_r[11] <= 0;
          if (t[10]+half_period < t[11]) shift_l[11] <= 1; else shift_l[11] <= 0;
         if (t[9]+t_star[11] > (t[10]<<1)+half_period) shift_r[10] <= 1; else shift_r[10]
<= 0;
         if (t[9]+t_star[11]+half_period < (t[10]<<1)) shift_l[10] <= 1; else shift_l[10]
<= 0;
          if (t[8]+t_star[10] > (t[9]<<1)+half_period) shift_r[9] <= 1; else shift_r[9]
<= 0;
          if (t[8]+t_star[10]+half_period < (t[9]<<1)) shift_l[9] <= 1; else shift_l[9]
<= 0;
          if (t[7]+t_star[9] > (t[8]<<1)+half_period) shift_r[8] <= 1; else shift_r[8] <=
0;
          if (t[7]+t_star[9]+half_period < (t[8]<<1)) shift_l[8] <= 1; else shift_l[8] <=
0;
          if (t[6]+t_star[8] > (t[7]<<1)+half_period) shift_r[7] <= 1; else shift_r[7] <=
0;
          if (t[6]+t_star[8]+half_period < (t[7]<<1)) shift_l[7] <= 1; else shift_l[7] <=
0;
          if (t[5]+t_star[7] > (t[6]<<1)+half_period) shift_r[6] <= 1; else shift_r[6] <=
0;
          if (t[5]+t_star[7]+half_period < (t[6]<<1)) shift_l[6] <= 1; else shift_l[6] <=
0;
          if (t[4]+t_star[6] > (t[5]<<1)+half_period) shift_r[5] <= 1; else shift_r[5] <=
0;
          if (t[4]+t_star[6]+half_period < (t[5]<<1)) shift_l[5] <= 1; else shift_l[5] <=
0;
          if (t[3]+t_star[5] > (t[4]<<1)+half_period) shift_r[4] <= 1; else shift_r[4] <=
0;
          if (t[3]+t_star[5]+half_period < (t[4]<<1)) shift_l[4] <= 1; else shift_l[4] <=
0;
          if (t[2]+t_star[4] > (t[3]<<1)+half_period) shift_r[3] <= 1; else shift_r[3] <=
0;
          if (t[2]+t_star[4]+half_period < (t[3]<<1)) shift_l[3] <= 1; else shift_l[3] <=
0;
          if (t[1]+t_star[3] > (t[2]<<1)+half_period) shift_r[2] <= 1; else shift_r[2] <=
0;
          if (t[1]+t_star[3]+half_period < (t[2]<<1)) shift_l[2] <= 1; else shift_l[2] <=
0;
          if (t[0]+t_star[2] > (t[1]<<1)+half_period) shift_r[1] <= 1; else shift_r[1] <=
0;
          if (t[0]+t_star[2]+half_period < (t[1]<<1)) shift_l[1] <= 1; else shift_l[1] <=
0;
      end //-1.2.3.2
         end //-1.2.3
   end //-1.2
end //-1

// t_star[] (locally shifted t[])
assign t_star[0] = (shift_l[0]) ? (t[0]-period) :
                   (shift_r[0]) ? (t[0]+period) :
                   t[0];
assign t_star[1] = (shift_l[1]) ? (t[1]-period) :
                   (shift_r[1]) ? (t[1]+period) :
                   t[1];
assign t_star[2] = (shift_l[2]) ? (t[2]-period) :
                   (shift_r[2]) ? (t[2]+period) :
                   t[2];
assign t_star[3] = (shift_l[3]) ? (t[3]-period) :
                   (shift_r[3]) ? (t[3]+period) :
                   t[3];
```

```
assign t_star[4] = (shift_l[4]) ? (t[4]-period) :
                   (shift_r[4]) ? (t[4]+period) :
                   t[4];
assign t_star[5] = (shift_l[5]) ? (t[5]-period) :
                   (shift_r[5]) ? (t[5]+period) :
                   t[5];
assign t_star[6] = (shift_l[6]) ? (t[6]-period) :
                   (shift_r[6]) ? (t[6]+period) :
                   t[6];
assign t_star[7] = (shift_l[7]) ? (t[7]-period) :
                   (shift_r[7]) ? (t[7]+period) :
                   t[7];
assign t_star[8] = (shift_l[8]) ? (t[8]-period) :
                   (shift_r[8]) ? (t[8]+period) :
                   t[8];
assign t_star[9] = (shift_l[9]) ? (t[9]-period) :
                   (shift_r[9]) ? (t[9]+period) :
                   t[9];
assign t_star[10] = (shift_l[10]) ? (t[10]-period) :
                   (shift_r[10]) ? (t[10]+period) :
                   t[10];
assign t_star[11] = (shift_l[11]) ? (t[11]-period) :
                   (shift_r[11]) ? (t[11]+period) :
                   t[11];

// generate output
wire [4:0] sl, sr;
wire [15:0] t_11_shifted;

assign sl = shift_l[0]+shift_l[1]+shift_l[2]+shift_l[3]+shift_l[4]+shift_l[5]
         +shift_l[6]+shift_l[7]+shift_l[8]+shift_l[9]+shift_l[10]+shift_l[11];
assign sr = shift_r[0]+shift_r[1]+shift_r[2]+shift_r[3]+shift_r[4]+shift_r[5]
      +shift_r[6]+shift_r[7]+shift_r[8]+shift_r[9]+shift_r[10]+shift_r[11];
assign t_11_shifted = (t1_0<t1_11)?(t[11]+period*sl-period*sr):(t[11]+period*sr-
period*sl);

assign t_0 = (t[0] < t_11_shifted) ? 0 : (t[0]-t_11_shifted);
assign t_11 = (t[0] < t_11_shifted) ? (t_11_shifted - t[0]) : 0;

//wire [11:0] t0, t1, t2, t11;
//assign t0 = t[0];
//assign t1 = t[1];
//assign t2 = t[2];
//assign t11 = t[11];

// done signal
reg [7:0] count_done;
always @ (posedge clk)
begin //2
if ((reset) | (start_pulse))
        begin
        count_done <= 0;
        done <= 0;
        end
else if (count_done > cycle)
        begin
        done <= 1'b1;
        count_done <= 0;
        end
else if (start) count_done <= count_done +1;
end //-2

assign done_br = (start_pulse) ? 0 : done;

endmodule
```

Calibration Lookup

```
module cal_lut(clk, r, cos_theta, cos_theta_cal);

input clk;
input [9:0] r, cos_theta;
output [9:0] cos_theta_cal;

wire [7:0] mem_data;

assign cos_theta_cal = cos_theta + mem_data;

ramlut ram32({cos_theta[8:5], r[6:3]}, clk, 8'b0, mem_data, 1'b0);

endmodule
```

Controller

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Create Date:    19:30:54 12/02/2007
// Module Name:    controller
//
////////////////////////////////////////////////////////////////////////////////
module controller(clk, reset, start, done_wpd12, finished_wpd12, done_dr, done_br,
done_ar, done_cr, mem_pointer1, mem_pointer2,
                  restart_wpd12, start_wpd12, start_dr, start_br, start_ar, start_cr,
done, mem_pointer, count, total_obj,
                                             // debug outputs
                                             state);
// debug outputs
output [3:0] state;

input clk, reset, start, done_wpd12, finished_wpd12, done_dr, done_br, done_ar, done_cr;
input [15:0] mem_pointer1, mem_pointer2;
output reg start_wpd12, start_dr, start_br, start_ar, start_cr, done;
output restart_wpd12;
output [15:0] mem_pointer;
output reg [3:0] count, total_obj;

reg [3:0] state;

// parameters
parameter s_wpd = 3'b000;
parameter s_dr = 3'b001;
parameter s_ar = 3'b010;
parameter s_cr = 3'b011;
parameter s_idle = 3'b100;
parameter s_br = 3'b101;
parameter num_obj = 4'd10;

// start pulse
wire start_pulse;
reg start_old;
always @ (posedge clk)
begin
start_old <= start;
end
assign start_pulse = (~start_old) & start;
//


// state machine
always @ (posedge clk)
if (reset)
   begin
   start_wpd12 <= 0;
   start_dr <= 0;
   start_ar <= 0;
        start_br <= 0;
   start_cr <= 0;
   count <=0;
   done <= 0;
        state <= s_idle;
        total_obj <= 0;
   end

else if (start_pulse)
   begin
   done <= 0;
   state <= s_wpd;
   start_wpd12 <= 1;
   count <= 0;
   end
```

```
else if (start)
   begin
   case (state)
       s_wpd:
          begin
          if (done_wpd12)
             begin
             if (finished_wpd12)
                begin
                state <= s_idle;
                                      start_wpd12 <= 0;
                end
             else
                // goto s_dr
                begin
                start_wpd12 <= 0;
                start_dr <= 1;
                state <= s_dr;
                end
             end
          else start_wpd12 <= 1; // stay
          end

       s_dr:
          begin
          if (done_dr)
             //goto s_br
             begin
             start_dr <= 0;
             start_br <= 1;
             state <= s_br;
             end
          else start_dr <= 1; //stay
          end

       s_br:
          begin
          if (done_br)
             //goto s_ar
             begin
             start_br <= 0;
             start_ar <= 1;
             state <= s_ar;
             end
          else start_br <= 1; //stay
          end

       s_ar:
          begin
          if (done_ar)
             //goto s_cr
             begin
             start_ar <= 0;
             start_cr <= 1;
             state <= s_cr;
             end
          else start_ar <= 1; //stay
          end

       s_cr:
          begin
          if (done_cr)
             //go for next object, or stop
             begin
             count <= count + 1;
             start_cr <= 0;
             if (count < num_obj)
```

```
                begin
                start_wpd12 <= 1;
                state <= s_wpd;
                end
             else
                begin
                state <= s_idle;
                end
             end
          else start_cr <= 1; //stay
          end

       s_idle:
          begin
          done <= 1;
          total_obj <= count;
          end

   endcase
   end

// outputs
assign restart_wpd12 = start_pulse;
assign mem_pointer = (state == s_wpd) ? mem_pointer1 : mem_pointer2;

endmodule
```

Cosine to Sine

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Create Date:    21:46:37 12/02/2007
// Module Name:    cos2sin
// Tested, working.
////////////////////////////////////////////////////////////////////////////////
module cos2sin(clk, reset, cos_theta, sin_theta, ready);
input clk, reset;
input [9:0] cos_theta;
output [9:0] sin_theta;
output ready;
reg [9:0] cos_square;
wire [9:0] sin_square;
reg [19:0] cos_square1;

// sin's and cos's are in this format: x.xxxxxxxxx (1QN)
//

parameter bits = 9;

always @ (posedge clk)
begin //1
if (reset)
        begin
        cos_square <= 0;
        end

else
        begin
        cos_square1 <= cos_theta * cos_theta;
        cos_square <= cos_square1 >> bits;
        end

end //-1

//assign sin_square[9] = 1'b0;
//assign sin_square[8:0] = (cos_square[9])? 9'b0 : (~cos_square[8:0]+1'b1);
assign sin_square = 10'b1000000000 - cos_square;

sqrt sqrt1(sin_square, clk, 1'b1, sin_theta, ready);

endmodule
```

Distance Retriever

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Create Date:    19:30:54 12/02/2007
// Module Name:    distance retriever
//
////////////////////////////////////////////////////////////////////////////////
module distance_retriever(clk, reset, start, reprogram, t1, t2,    a1, b1,
   a1_cal, b1_cal, r, done_dr);

input clk, reset, start;
input reprogram;
input [15:0] t1, t2;
input [9:0] a1, b1;
output reg [9:0] a1_cal, b1_cal, r;
output done_dr;
reg done;

reg [25:0] r1, rc1;
reg [9:0] rc;
reg [15:0] t;
reg [2:0] count;

// parameters
parameter cycle = 3'd4;
parameter bits = 14;
parameter r_cal = 10'd300;

// start_pulse, reprogram_pulse
reg start_old, reprogram_old;
wire start_pulse, reprogram_pulse;
always @ (posedge clk)
begin
start_old <= start;
reprogram_old <= reprogram;
end
assign start_pulse = (~start_old) & start;
assign reprogram_pulse = (~reprogram_old) & reprogram;
///

// main
always @ (posedge clk)
begin //1
if (reset)
       begin
       a1_cal <= 10'b0;
       b1_cal <= 10'b0;
       r <= 0;
       rc <= 0;
       rc1 <= 0;
       t <= 0;
       end

else if (reprogram)
       // only b1 is need to be calibrated.
       // a1, a2 depend on the speed of sound, and the distance between receivers; a1, a2
should be calibrated before production
       // b2 can be calibrated more easily by mechanics.
       // 2 clock cycles needed.
       begin
       t <= t1; // the first bit of t1, t2, t is always 0;
       rc1 <= a1*t;
       rc <= rc1>>bits;
       if (rc > r_cal)
              begin
              b1_cal[8:0] <= rc - r_cal;
              b1_cal[9] <= 1'b1;
```

```
                end
        else b1_cal <= r_cal - rc;
        a1_cal <= a1;
        r1 <= a1*t;
        r <= (b1[9])? (r1 >> bits) - b1[8:0] : (r1 >> bits) + b1;
        end

else if (start)
        begin
        t <= t1; // the first bit of t1, t2, t is always 0;
        r1 <= a1*t;
        r <= (b1[9])? (r1 >> bits) - b1[8:0] : (r1 >> bits) + b1;
        end
end //-1


// done signal
always @ (posedge clk)
begin //2
if ((reset) | (start_pulse) | (reprogram_pulse))
        begin
        count <= 0;
        done <= 0;
        end
else if (count == cycle)
        begin
        done <= 1'b1;
        count <= 0;
        end
else if (start | reprogram) count <= count +1;
end //-2

assign done_dr = (start_pulse) ? 0 : done;

endmodule
```

Parameter Manager

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Create Date:    19:27:01 12/02/2007
// Module Name:    parameter_manager
// not tested, but looks so good...:)
////////////////////////////////////////////////////////////////////////////////


module para_manager(clk, reset, reprogram, a1_cal, b1_cal, ready_cal,
       a1, b1);
input clk, reset, reprogram, ready_cal;
input [9:0] a1_cal, b1_cal;
output reg [9:0] a1, b1;

// reprogram take a few clock cycles (see distance_retriever).
// tie the ready_cal to the done signal of distance_retriever, so that
// para_manager takes the calibration values when cr is done

always @ (posedge clk)
begin
if (reset)
       begin
       a1 <= 285;
       b1 <= 0;
       end

else if (reprogram & ready_cal)
       begin
       a1 <= a1_cal;
       b1 <= b1_cal;
       end
end

endmodule
```

Polar to cartesian

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Create Date:    19:26:26 12/02/2007
// Module Name:    polar2cart
// tested, but for some reason i have to buffer the output as soon as it's done
////////////////////////////////////////////////////////////////////////////////

module polar2cart(clk, reset, start, finish, r, cos_theta, obj_num,
        x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, y0, y1, y2, y3, y4, y5, y6, y7, y8, y9,
done_p2c,
        //debug outputs
        sin_theta, cos_abs
        );
//debug outputs
output [9:0] sin_theta, cos_abs;


input clk, reset, start, finish;
input [3:0] obj_num;
input [9:0] r;
input signed [9:0] cos_theta;
output reg [10:0] x0;
output reg [10:0] x1;
output reg [10:0] x2;
output reg [10:0] x3;
output reg [10:0] x4;
output reg [10:0] x5;
output reg [10:0] x6;
output reg [10:0] x7;
output reg [10:0] x8;
output reg [10:0] x9;
output reg [9:0] y0;
output reg [9:0] y1;
output reg [9:0] y2;
output reg [9:0] y3;
output reg [9:0] y4;
output reg [9:0] y5;
output reg [9:0] y6;
output reg [9:0] y7;
output reg [9:0] y8;
output reg [9:0] y9;
output done_p2c;

reg done;

parameter bits =9;
parameter num_obj =10;
parameter cycle = 20;
parameter signed half_x_display = 512;
parameter half_y_display = 384;

reg signed [19:0] x_buf, x_buf1;
reg [19:0] y_buf, y_buf1;
wire [9:0] sin_theta, cos_abs;
// wire signed [9:0] cos_theta;

// main
always @ (posedge clk)
begin //1
if (reset)
        begin
        x0 <= 0;
        x1 <= 0;
        x2 <= 0;
        x3 <= 0;
        x4 <= 0;
        x5 <= 0;
        x6 <= 0;
        x7 <= 0;
        x8 <= 0;
        x9 <= 0;
        y0 <= 0;
        y1 <= 0;
        y2 <= 0;
        y3 <= 0;
        y4 <= 0;
        y5 <= 0;
        y6 <= 0;
        y7 <= 0;
        y8 <= 0;
        y9 <= 0;
        end

else if (start)
        begin
        x_buf1 <= r * cos_theta;
        y_buf1 <= r * sin_theta;
        x_buf <= (half_x_display << bits) + x_buf1;
        y_buf <= (half_y_display << bits) - y_buf1;

        case (obj_num)
            0:
            begin
            x0 <= x_buf >> bits;
            y0 <= y_buf >> bits;
            end
            1:
            begin
            x1 <= x_buf >> bits;
            y1 <= y_buf >> bits;
            end
            2:
            begin
            x2 <= x_buf >> bits;
            y2 <= y_buf >> bits;
            end
            3:
            begin
            x3 <= x_buf >> bits;
            y3 <= y_buf >> bits;
            end
            4:
            begin
            x4 <= x_buf >> bits;
            y4 <= y_buf >> bits;
            end
            5:
            begin
            x5 <= x_buf >> bits;
            y5 <= y_buf >> bits;
            end
            6:
            begin
            x6 <= x_buf >> bits;
            y6 <= y_buf >> bits;
            end
            7:
            begin
            x7 <= x_buf >> bits;
            y7 <= y_buf >> bits;
            end
            8:
            begin
            x8 <= x_buf >> bits;
            y8 <= y_buf >> bits;
            end
            9:
            begin
            x9 <= x_buf >> bits;
            y9 <= y_buf >> bits;
            end
        endcase
        end
end //-1

assign cos_abs = (cos_theta < 0) ? (~cos_theta + 1) : cos_theta;
cos2sin c2s(clk, reset, cos_abs, sin_theta, c2s_ready);

// start pulse
wire start_pulse;
reg start_old;
always @ (posedge clk)
begin
start_old <= start;
end
assign start_pulse = (~start_old) & start;

// done signal
reg [4:0] count;
always @ (posedge clk)
begin //2
if ((reset) | (start_pulse))
        begin
        count <= 0;
        done <= 0;
        end
else if (count == cycle)
        begin
        done <= 1'b1;
        count <= 0;
        end
else if (start) count <= count +1;
end //-2

assign done_p2c = (start_pulse) ? 0 : done;

endmodule
```

Signal Analysis System

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Create Date:    19:27:01 12/02/2007
// Module Name:    signal_analysis_system, toplevel
// Almost just connecting modules..
////////////////////////////////////////////////////////////////////////////////


module signal_analysis_system(clk, reset, reprogram_sw, reprogram_button, start, mem_data,
       x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, y0, y1, y2, y3, y4, y5, y6, y7, y8, y9,
total_obj, done, mem_pointer);

input clk, reset, reprogram_sw, reprogram_button, start;
input [11:0] mem_data;
output [15:0] mem_pointer;
output [10:0] x0, x1, x2, x3, x4, x5, x6, x7, x8, x9;
output [9:0] y0, y1, y2, y3, y4, y5, y6, y7, y8, y9;
output [3:0] total_obj;
output done;

wire [15:0] mem_pointer1, mem_pointer2;
wire [3:0] obj_num;
wire [15:0] t1_0, t1_1, t1_2, t1_3, t1_4, t1_5, t1_6, t1_7, t1_8, t1_9, t1_10, t1_11,
        t2_0, t2_1, t2_2, t2_3, t2_4, t2_5, t2_6, t2_7, t2_8, t2_9, t2_10, t2_11;
wire [15:0] t1_d, t2_d;
wire [9:0] a1, b1, a1_cal, b1_cal;
wire [9:0] cos_theta, r;

// debug wires
wire [3:0] state;
wire [9:0] sin_theta, cos_abs;

controller ctrl(clk, reset, start, done_wpd12, finished_wpd12, done_dr, done_br, done_ar,
done_cr, mem_pointer1, mem_pointer2,
            restart_wpd12, start_wpd12, start_dr, start_br, start_ar, start_cr, done,
mem_pointer, obj_num, total_obj,
                              // debug output
                              state
                              );
wave_package_detector_12 wpd12(clk, reset, restart_wpd12, start_wpd12, mem_data,
            mem_pointer1,
            t1_0, t1_1, t1_2, t1_3, t1_4, t1_5, t1_6, t1_7, t1_8, t1_9, t1_10, t1_11,
            t2_0, t2_1, t2_2, t2_3, t2_4, t2_5, t2_6, t2_7, t2_8, t2_9, t2_10, t2_11,
            done_wpd12, finished_wpd12);
assign t1_d = (t1_4>>2) + (t1_5>>2) + (t1_6>>2) + (t1_7>>2);
assign t2_d = (t2_4>>2) + (t2_5>>2) + (t2_6>>2) + (t2_7>>2);
distance_retriever dr(clk, reset, start_dr, reprogram_sw, t1_d, t2_d, a1, b1,
            a1_cal, b1_cal, r, done_dr);
para_manager pm(clk, reset, reprogram_sw, a1_cal, b1_cal, reprogram_button,
            a1, b1);
boundary_retriever br(clk, reset, start_br, r, t1_0, t2_0, t1_11, t2_11, mem_data,
                     t_0, t_11, mem_pointer2, done_br);
angle_retriever ar(clk, reset, start_ar, t_0, t_11, r,
            cos_theta, done_ar);
polar2cart p2c(clk, reset, start_cr, finish, r, cos_theta, obj_num,
            x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, y0, y1, y2, y3, y4, y5, y6, y7, y8,
y9, done_cr,
                              //debug outputs
                              sin_theta, cos_abs);


endmodule
```

Wave Package Detector

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Create Date:    19:19:57 12/02/2007
// Module Name:    wave_package_detector
// one-channel version
////////////////////////////////////////////////////////////////////////////////
module wave_package_detector(clk, reset, start, mem_pointer, mem_data,
                                      t1, t2, done_wpd);
input clk, reset, start;
input [15:0] mem_pointer;
input mem_data;
output reg [15:0] t1, t2;
//output reg [15:0] mem_pointer;
output done_wpd;

reg done;
reg [2:0] state;
reg [11:0] mem_data_old;
reg [4:0] sum;
reg [4:0] count;
reg [15:0] t_maybe;

// parameters
parameter catch_pos_edge = 3'b000;
parameter verify_pos_edge = 3'b001;
parameter catch_neg_edge = 3'b010;
parameter verify_neg_edge = 3'b011;
parameter idle = 3'b100;

parameter pos_th = 5'd9;
parameter neg_th = 5'd3;
parameter half_period = 5'd12;
parameter period = 5'd25;

// start pulse
wire start_pulse;
reg start_old;
always @ (posedge clk)
begin
start_old <= start;
end
assign start_pulse = (~start_old) & start;

// state machine
always @ (posedge clk)
begin
if (reset)
   begin
   state <= idle;
   t1 <= 0;
   t2 <= 0;
   mem_data_old <= 0;
   sum <= 0;
   count <= 0;
   t_maybe <= 0;
   done <= 0;
   end

else if (start_pulse)
   begin
   state <= catch_pos_edge;
   done <= 0;
        mem_data_old <= 1'b1;
   end

else
```

```
   begin //1
   case (state)
      idle:
         begin
         done <= 1'b1;
         end

      catch_pos_edge:
         begin
         mem_data_old <= mem_data;
//         mem_pointer <= mem_pointer + 1;
         if (mem_data & ~mem_data_old)
            begin
            state <= verify_pos_edge;
            sum <= 0;
            count <= 0;
            t_maybe <= mem_pointer;
            end
         end // catch_pos_edge

      verify_pos_edge:
         begin
         if (count < half_period)
            begin
//            mem_pointer <= mem_pointer + 1;
            sum <= sum + mem_data;
            count <= count + 1;
            end
         else if (sum > pos_th)
            begin
            state <= catch_neg_edge;
            t1 <= t_maybe;
            end
         else
            begin
            state <= catch_pos_edge;
            end
         end // verify_pos_edge

      catch_neg_edge:
         begin
         mem_data_old <= mem_data;
//         mem_pointer <= mem_pointer + 1;
         if (~mem_data)
            begin
            state <= verify_neg_edge;
            sum <= 0;
            count <= 0;
            t_maybe <= mem_pointer;
            end
         end  //catch_neg_edge

      verify_neg_edge:
         begin
         if (count < period)
            begin
//            mem_pointer <= mem_pointer + 1;
            sum <= sum + mem_data;
            count <= count + 1;
            end
         else if (sum < neg_th)
            // get it!
            begin
            state <= idle;
            t2 <= t_maybe;
            end
         else
            begin
            state <= catch_neg_edge;
            end
         end // verify_neg_edge
   endcase
   end //-1
end

assign done_wpd = (start_pulse) ? 0 : done;

endmodule
```

Wave Package Detector 12

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Create Date:    19:19:57 12/02/2007
// Module Name:    wave_package_detector_12
// one-channel version
////////////////////////////////////////////////////////////////////////////////
module wave_package_detector_12(clk, reset, restart, start, mem_data,
                   mem_pointer,
                   t1_0, t1_1, t1_2, t1_3, t1_4, t1_5, t1_6, t1_7, t1_8, t1_9, t1_10,
t1_11,
                   t2_0, t2_1, t2_2, t2_3, t2_4, t2_5, t2_6, t2_7, t2_8, t2_9, t2_10,
t2_11,
                   done_wpd12, finished);
input clk, reset, restart, start;
input [11:0] mem_data;
output [15:0] t1_0, t2_0;
output [15:0] t1_1, t2_1;
output [15:0] t1_2, t2_2;
output [15:0] t1_3, t2_3;
output [15:0] t1_4, t2_4;
output [15:0] t1_5, t2_5;
output [15:0] t1_6, t2_6;
output [15:0] t1_7, t2_7;
output [15:0] t1_8, t2_8;
output [15:0] t1_9, t2_9;
output [15:0] t1_10, t2_10;
output [15:0] t1_11, t2_11;

output reg [15:0] mem_pointer;
output reg finished;
output done_wpd12;
reg done;
wire [11:0] wpddone;

parameter max_mem = 16'b0111_1111_1111_1000; // 32k memory, leaving some blank space in
the end

// start pulse
wire start_pulse;
reg start_old;
always @ (posedge clk)
begin
start_old <= start;
end
assign start_pulse = (~start_old) & start;

//main
always @ (posedge clk)
begin
if (reset | restart)
   begin
   mem_pointer <= 0;
   done <= 0;
   finished <= 0;
   end

else if (start)
   begin
   if (wpddone == 12'b1111_1111_1111)
      begin
      done <= 1;
      end
   else if (mem_pointer == max_mem)
      begin
      done <= 1;
      finished <= 1;
      end
   else
      begin
      mem_pointer <= mem_pointer + 1;
      done <=0;
      end
   end
end


wave_package_detector wpd0(clk, reset, start, mem_pointer, mem_data[0], t1_0, t2_0,
wpddone[0]);
wave_package_detector wpd1(clk, reset, start, mem_pointer, mem_data[1], t1_1, t2_1,
wpddone[1]);
wave_package_detector wpd2(clk, reset, start, mem_pointer, mem_data[2], t1_2, t2_2,
wpddone[2]);
wave_package_detector wpd3(clk, reset, start, mem_pointer, mem_data[3], t1_3, t2_3,
wpddone[3]);
wave_package_detector wpd4(clk, reset, start, mem_pointer, mem_data[4], t1_4, t2_4,
wpddone[4]);
wave_package_detector wpd5(clk, reset, start, mem_pointer, mem_data[5], t1_5, t2_5,
wpddone[5]);
wave_package_detector wpd6(clk, reset, start, mem_pointer, mem_data[6], t1_6, t2_6,
wpddone[6]);
wave_package_detector wpd7(clk, reset, start, mem_pointer, mem_data[7], t1_7, t2_7,
wpddone[7]);
wave_package_detector wpd8(clk, reset, start, mem_pointer, mem_data[8], t1_8, t2_8,
wpddone[8]);
wave_package_detector wpd9(clk, reset, start, mem_pointer, mem_data[9], t1_9, t2_9,
wpddone[9]);
wave_package_detector wpd10(clk, reset, start, mem_pointer, mem_data[10], t1_10, t2_10,
wpddone[10]);
wave_package_detector wpd11(clk, reset, start, mem_pointer, mem_data[11], t1_11, t2_11,
wpddone[11]);

assign done_wpd12 = (start_pulse) ? 0 : done;
endmodule
```

Sonar Project Top level Verilog

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

module debounce (reset, clock_65mhz, noisy, clean);
   input reset, clock_65mhz, noisy;
   output clean;

   reg [19:0] count;
   reg new, clean;

   always @(posedge clock_65mhz)
     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 -- 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 basiclayout   (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;

    inout mouse_clock, mouse_data;
input  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
     wire sound_output;
     wire sound_on;
     //assign user1[0] = sound_output;
     //assign user1[1] = sound_on;
//assign user1[31:2] = 30'hZ;
assign user2 = 32'hZ;
//assign user3 = 32'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 = {15'h0,sound_on};
//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;

////////////////////////////////////////////////////////////////////////////
//
// lab5 : a simple pong game
//
////////////////////////////////////////////////////////////////////////////

// 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(power_on_reset, clock_65mhz, ~button_enter, user_reset);
   assign reset = user_reset | power_on_reset;

   // UP and DOWN buttons for pong paddle
   wire up,down,left,right;
   debounce db2(reset, clock_65mhz, ~button_up, up);
   debounce db3(reset, clock_65mhz, ~button_down, down);
   debounce db4(reset, clock_65mhz, ~button_left, left);
   debounce db5(reset, clock_65mhz, ~button_right, right);

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

   wire [11:0] mx,my;
   wire [2:0]  btn_click;
   ps2_mouse_xy m1(clock_65mhz, reset, mouse_clock, mouse_data, mx, my, btn_click);
   defparam m1.MAX_X = 1023;  // max - blob size
   defparam m1.MAX_Y = 767;

// ***************** BRYAN MORRISSEY ********************


   /// Sonar connection definitions:
   //wire   power_on_reset; // remain high for first 16 clocks
   wire   reset_sync, sonar_reset;
   wire [11:0] adc_inputs;
   wire [11:0] adc_outputs;
   wire [1:0]  pulse_output;
   wire        thirty_hz_enable;
   // wire sonar_start,
   wire        sonar_done;
   wire        one_mhz_enable;
   wire [63:0] hex_display_data;

   wire [14:0] bram_sonar_addr;
   wire [11:0] bram_sonar_data;
   wire [14:0] bram_analyzer_addr;
   wire [11:0] bram_analyzer_data;
   wire        bram_write_enable;
   wire [7:0]  switch_db;
   wire [35:0] hex_data; // Zhen

   assign reset_sync = user_reset || power_on_reset;
   assign sonar_reset = reset_sync || ~switch_db[6];
   assign led = ~switch_db;

   // Assigning user port pins to adc input and output:

   assign user3 = {thirty_hz_enable, one_mhz_enable, 28'hZ, pulse_output[1],
pulse_output[0]};  // 40kHz pulse output

//   assign bram_analyzer_addr = 0;
//   assign bram_write_enable = 1;
```

```
   // Debounce button_enter to be user reset
   //debounce db11(power_on_reset, clock_65mhz, ~button_enter, user_reset);
   // Switch Debouncing:

//   debounce sdb0(power_on_reset, clock_65mhz, switch[0], switch_db[0]);
//   debounce sdb1(power_on_reset, clock_65mhz, switch[1], switch_db[1]);
//   debounce sdb2(power_on_reset, clock_65mhz, switch[2], switch_db[2]);
//   debounce sdb3(power_on_reset, clock_65mhz, switch[3], switch_db[3]);
//   debounce sdb4(power_on_reset, clock_65mhz, switch[4], switch_db[4]);
//   debounce sdb5(power_on_reset, clock_65mhz, switch[5], switch_db[5]);
//   debounce sdb6(power_on_reset, clock_65mhz, switch[6], switch_db[6]);
//   debounce sdb7(power_on_reset, clock_65mhz, switch[7], switch_db[7]);
   assign switch_db = switch;

   // Instantiate sonar_data_acquisition module:
   sonar_data_acquisition sdacq0(clock_65mhz, sonar_reset,
             thirty_hz_enable, one_mhz_enable,
                              adc_inputs, adc_outputs,
                              pulse_output, switch_db,
                              hex_display_data,
                              analyzer1_data, analyzer1_clock,
                              analyzer3_data, analyzer3_clock,
                              bram_sonar_addr, bram_sonar_data, bram_write_enable);

   // Hexadecimal LED display for debugging information:
   display_16hex hex_display1(reset_sync, clock_65mhz, {hex_data[31:0], hex_data[35:32],
hex_display_data[27:0]},
                             disp_blank, disp_clock, disp_rs, disp_ce_b,
                             disp_reset_b, disp_data_out);

   bufferwr_12b32k bram0(clock_65mhz, bram_sonar_data, bram_sonar_addr, bram_write_enable,
                         clock_65mhz, bram_analyzer_addr, bram_analyzer_data);

   assign user1 = {sound_output, 7'hZ,
                  adc_outputs[11], 1'hZ, adc_outputs[10], 1'hZ, adc_outputs[9], 1'hZ,
                  adc_outputs[8],  1'hZ, adc_outputs[7],  1'hZ, adc_outputs[6], 1'hZ,
                  adc_outputs[5],  1'hZ, adc_outputs[4],  1'hZ, adc_outputs[3], 1'hZ,
                  adc_outputs[2],  1'hZ, adc_outputs[1],  1'hZ, adc_outputs[0], 1'hZ};

   assign adc_inputs[11:0] = {user1[22], user1[20], user1[18], user1[16], user1[14],
user1[12],
                            user1[10], user1[8],  user1[6],  user1[4],  user1[2],
user1[0]};

   wire done;
   assign sonar_done = (bram_sonar_addr >= 15'h7000);
   // level_to_pulse ltop0(clk_65mhz, done, sonar_start);
   /// assign sonar_start = done;


// **************** END OF BRYAN CODE *********************



// **************** ZHEN CODE *********************
        wire start;
        wire [10:0] x0, x1, x2, x3, x4, x5, x6, x7, x8, x9;
        wire [9:0] y0, y1, y2, y3, y4, y5, y6, y7, y8, y9;
        wire [3:0] total_obj;
        wire [11:0] mem_data;
        wire [15:0] mem_pointer;
        wire we;
        assign we = 1'b0;
        assign mem_data = bram_analyzer_data;
   assign bram_analyzer_addr = mem_pointer[14:0];
   assign reprogram_sw = switch_db[0];
   assign start = sonar_done;
   debounce bdb1(power_on_reset, clock_65mhz, reprogram_button_in, reprogram_button);
   assign reprogram_button_in = ~button0;

       signal_analysis_system sas(clock_65mhz, reset, reprogram_sw, reprogram_button,
start, mem_data,
       x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, y0, y1, y2, y3, y4, y5, y6, y7, y8, y9,
total_obj, done, mem_pointer);

//      ram32x12 ram32(mem_pointer[14:0], clock_65mhz, 12'b0, mem_data, we);

// one_hz
reg one_Hz_enable;
reg [31:0] one_hz_count;
reg [2:0] one_hz_sel;

parameter Freq = 65_000_000; // #counts, = clock frequency

always @ (posedge clock_65mhz)
begin
if (one_hz_count == Freq)
   begin
   one_Hz_enable <= 1'b1;
   one_hz_count <= 0;
   one_hz_sel <= one_hz_sel + 1;
   end
else
   begin
   one_Hz_enable <= 1'b0;
   one_hz_count <= one_hz_count + 1;
       end
end
///

// display_16hex display_led(reset, clock_27mhz, data, disp_blank, disp_clock, disp_rs,
disp_ce_b,     disp_reset_b, disp_data_out);
       assign hex_data[15:0] = (one_hz_sel == 3'b000)? y0:
                                                                      (one_hz_sel ==
3'b001)? y1:
                                                                      (one_hz_sel ==
3'b010)? y2:
                                                                      (one_hz_sel ==
3'b011)? y3:
                                                                      (one_hz_sel ==
3'b100)? y4:
                                                                      (one_hz_sel ==
3'b101)? y5:
                                                                      (one_hz_sel ==
3'b110)? y6:
                                                                      (one_hz_sel ==
3'b111)? y7:
                                                                      y8;
       assign hex_data[31:16] = (one_hz_sel == 3'b000)? x0:
                                                                 (one_hz_sel ==
3'b001)? x1:
                                                                 (one_hz_sel ==
3'b010)? x2:
                                                                 (one_hz_sel ==
3'b011)? x3:
                                                                 (one_hz_sel ==
3'b100)? x4:
                                                                 (one_hz_sel ==
3'b101)? x5:
                                                                 (one_hz_sel ==
3'b110)? x6:
                                                                 (one_hz_sel ==
3'b111)? x7:
                                                                 x8;
   assign hex_data[35:32] = one_hz_sel;
```

```
//{done_wpd12, finished_wpd12, done_dr, done_br,
//                                                  done_ar,      done_cr,  restart_wpd12,
start_wpd12,
//                                                  start_dr, start_br, start_ar,
start_cr};


// **************** END OF ZHEN CODE *********************




// **************** BRIAN CODE *********************

   wire [23:0] pixel;
   wire phsync,pvsync,pblank;
   layout_sim pg(clock_65mhz,clock_27mhz,reset,up,down,left,right,
                                             4'b0010,~switch[6],1'b0,1'b0,
                                             mx,my,btn_click,
                                             x0,y0,
                                             x1,y1,
                  x2,y2,
                  x3,y3,
                  x4,y4,
                                             hcount,vcount,hsync,vsync,blank,
                                             phsync,pvsync,pblank,
                                             sound_output,sound_on,
                                             pixel);

// **************** END OF BRIAN CODE *********************



   // switch[1:0] selects which video generator to use:
   //  00: user's pong game
   //  01: 1 pixel outline of active video area (adjust screen controls)
   //  10: color bars
   reg [23:0] rgb;
   reg b,hs,vs;
   always @ (posedge clock_65mhz) begin
             hs <= phsync;
             vs <= pvsync;
             b <= pblank;
             rgb <= pixel;
   end

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
   assign vga_out_red = {rgb[23:16]};
   assign vga_out_green = {rgb[15:8]};
   assign vga_out_blue = {rgb[7:0]};
   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;

   //assign led[7:0] = ~{4'b0000,switch[7:4]};

endmodule

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

```
module xvga(vclock,hcount,vcount,hsync,vsync,blank);
   input vclock;
   output [10:0] hcount;
   output [9:0] vcount;
   output      vsync;
   output      hsync;
   output      blank;

   reg    hsync,vsync,hblank,vblank,blank;
   reg [10:0]   hcount;    // pixel number on current line
   reg [9:0] vcount;    // line number

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   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

////////////////////////////////////////////////////////////////////////////////
//
// pong_game: the game itself!
//
////////////////////////////////////////////////////////////////////////////////

module layout_sim (vclock,clock_27mhz,reset,up,down,left,right,
                 mem_sel,stop_spin,debug,calibration_mode,
                 mx,my,mouse_button_click,
                 x_in_1,y_in_1,
                 x_in_2,y_in_2,
        x_in_3,y_in_3,
        x_in_4,y_in_4,
        x_in_5,y_in_5,
                 hcount,vcount,hsync,vsync,blank,
                 phsync,pvsync,pblank,
                 sound_output,sound_on,
                 pixel);
   input vclock;       // 65MHz clock
       input clock_27mhz;     // 27MHz clock
   input reset;               // 1 to initialize module
   input up;           // 1 when paddle should move up
   input down;         // 1 when paddle should move down
```

```
	input left;
	input right;
input [3:0] mem_sel;  // puck speed in pixels/tick
	input stop_spin;
	input debug;
	input calibration_mode;
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[11:0] mx,my;
	input[2:0] mouse_button_click;

	input[10:0] x_in_1,x_in_2,x_in_3,x_in_4,x_in_5;
	input[9:0] y_in_1,y_in_2,y_in_3,y_in_4,y_in_5;

output phsync;      // pong game's horizontal sync
output pvsync;      // pong game's vertical sync
output pblank;      // pong game's blanking
output [23:0] pixel;       // pong game's pixel
	output sound_output;
	output sound_on;

	// PIPELINING REGISTERS
	parameter PIPELINE_STAGE = 32;
	reg[11:0] dist_memory[9:0];
	reg pie_p[31:2];
	reg[6:0] angle_loc_p[31:2];
	reg[22:0] dist_squared_p[2:1];
	reg[11:0] dist_p_1[6:6];
	reg[11:0] dist_p_2[6:6];
	reg[23:0] grid_pixel_p[31:2];
	reg[23:0] sweeper_pixel_p[31:2];
	//reg[23:0] object_pixel_p[10:1][31:2];
	reg[23:0] object_pixel_p_1[31:2];
	reg[23:0] object_pixel_p_2[31:2];
	reg[23:0] object_pixel_p_3[31:2];
	reg[23:0] object_pixel_p_4[31:2];
	reg[23:0] object_pixel_p_5[31:2];
	reg[23:0] text_1_pixel_p[31:2];
	reg[23:0] text_2_pixel_p[31:2];
	reg[23:0] logo_pixel_p[31:2];
	reg[23:0] menu_pixel_p[31:2];
	reg[23:0] uptime_pixel_p[31:2];
	reg[23:0] calibration_pixel_p[31:2];
	reg[23:0] sweeper_divider_quotient_p[31:28];
	reg[23:0] grid_divider_quotient_p[31:28];
	reg[23:0] alpha_blended_p[31:29];
	reg[23:0] final_object_pixel_p[30:30];
	reg[23:0] final_text_1_pixel_p[30:30];
	reg[23:0] final_background_pixel_p[30:30];
	reg[23:0] pixel_p[31:31];
	reg vsync_p[31:0];
	reg hsync_p[31:0];
	reg blank_p[31:0];
	integer i,j,k,p,q;


	// ****************** STAGE 1 PIPELINE *********************

	// *** Grid generator ***
	wire[23:0] grid_pixel;
	sonargrid my_grid (hcount,vcount,vclock,
	                                            grid_pixel);

	wire[23:0] menu_pixel;
	wire[2:0] mode;
	guimenu my_menu (vclock,hcount,vcount,mx,my,mouse_button_click,
	                                            mode,menu_pixel);

	// *** Object generator ***
	/*
	wire[23:0] object_pixel[10:1];        // 24bit RGB
	*/
	//reg sound_on;
	reg[11:0] x[10:1];     // Center of mass X
	reg[11:0] y[10:1];     // Center of mass Y
	wire on[10:1];
	/*
	assign on[10] = 1'b1;
	assign on[9] = 1'b1;
	assign on[8] = 1'b1;
	assign on[7] = 1'b1;
	assign on[6] = 1'b1;
	*/
	assign on[5] = 1'b1;
	assign on[4] = 1'b1;
	assign on[3] = 1'b1;
	assign on[2] = 1'b1;
	assign on[1] = 1'b1;

	wire[23:0] object_pixel_1, object_pixel_2, object_pixel_3, object_pixel_4,
object_pixel_5;
	wire warn[10:1];
	//basicobject my_object_1
(vclock,hcount,vcount,x[1],y[1],on[1],warn[1],object_pixel_1);
	//basicobject my_object_2
(vclock,hcount,vcount,x[2],y[2],on[2],warn[2],object_pixel_2);

	basicobject my_object_1 (vclock,hcount,vcount,x[1],y[1],on[1],object_pixel_1);
	basicobject my_object_2 (vclock,hcount,vcount,x[2],y[2],on[2],object_pixel_2);
	basicobject my_object_3 (vclock,hcount,vcount,x[3],y[3],on[3],object_pixel_3);
	basicobject my_object_4 (vclock,hcount,vcount,x[4],y[4],on[4],object_pixel_4);
	basicobject my_object_5 (vclock,hcount,vcount,x[5],y[5],on[5],object_pixel_5);
	/*
	basicobject my_object_6 (vclock,hcount,vcount,x[6],y[6],on[6],object_pixel[6]);
	basicobject my_object_7 (vclock,hcount,vcount,x[7],y[7],on[7],object_pixel[7]);
	basicobject my_object_8 (vclock,hcount,vcount,x[8],y[8],on[8],object_pixel[8]);
	basicobject my_object_9 (vclock,hcount,vcount,x[9],y[9],on[9],object_pixel[9]);
	basicobject my_object_10
(vclock,hcount,vcount,x[10],y[10],on[10],object_pixel[10]);
	*/

	// *** Sweeper generator ***
	wire[6:0] angle_out_1;               // angle output of current hcount, vcount
	wire[6:0] angle_out_2;
	wire pie;                                       // in the pie?
	wire[6:0] angle_loc;                 // RANGE:0-90 location of (hcount,vcount) in
the pie
	wire[23:0] sweeper_pixel;     // 24 bit RGB
	wire[6:0] DELTA_PIE = 30;     // The pie is 30 degrees
	sweeper my_sweeper (vsync,vclock,hcount,vcount,mem_sel,stop_spin,debug,
	                                                x[1],y[1],angle_out_1,
	                                                x[2],y[2],angle_out_2,
	                                                pie,angle_loc,
	                                                sweeper_pixel);

	// *** Sound generator ***
	wire sound_output_1, sound_output_2;
	wire warn_signal = warn[1] | warn[2];
	wire sound_output = warn_signal ? sound_output_2 : sound_output_1;
	sirengenerator object_siren (vclock,sound_on,sound_output_1);
	alarm_gen my_alarm_gen (vclock,warn_signal,sound_output_2);

	// *** Vector generator ***
```

```
       //wire[11:0] x2 = 100;
       //wire[11:0] y2 = 100;
       wire[23:0] vector_pixel;
       //vector_gen my_vector (vclock,x1,y1,x2,y2,hcount,vcount,vector_pixel);


       // *** TEXT 1 ***
       parameter TEXT_DELTA_X = 15;
       parameter TEXT_DELTA_Y = 5;
       wire[7:0] d1_3, d1_2, d1_1;
       wire[7:0] d1_3a, d1_2a, d1_1a;
       binarytodec my_btod_converter_11 (vclock,{3'b000,angle_out_1},d1_3,d1_2,d1_1);
       binarytodec my_btod_converter_12 (vclock,dist_p_1[6][9:0],d1_3a,d1_2a,d1_1a);
       wire[87:0] cstring_1 = {d1_3,d1_2,d1_1,8'b00100111,
                                                                8'b00001101,

       d1_3a,d1_2a,d1_1a,

       8'b01110000,8'b01101001,8'b01111000};
   wire[23:0] text_1_pixel;
   char_string_display text1(vclock,hcount,vcount,
                        text_1_pixel,cstring_1,(x[1] + TEXT_DELTA_X),(y[1] +
TEXT_DELTA_Y));
       defparam text1.NCHAR = 11;
       defparam text1.NCHAR_BITS = 4;


       // *** TEXT 2 ***
       wire[7:0] d2_3, d2_2, d2_1;
       wire[7:0] d2_3a, d2_2a, d2_1a;
       binarytodec my_btod_converter_21 (vclock,{2'b00,angle_out_2},d2_3,d2_2,d2_1);
       binarytodec my_btod_converter_22 (vclock,dist_p_2[6][9:0],d2_3a,d2_2a,d2_1a);
       wire[87:0] cstring_2 = {d2_3,d2_2,d2_1,8'b00100111,
                                                                8'b00001101,

       d2_3a,d2_2a,d2_1a,

       8'b01110000,8'b01101001,8'b01111000};
   wire[23:0] text_2_pixel;
   char_string_display text2(vclock,hcount,vcount,
                        text_2_pixel,cstring_2,(x[2] + TEXT_DELTA_X),(y[2] +
TEXT_DELTA_Y));
       defparam text2.NCHAR = 11;
       defparam text2.NCHAR_BITS = 4;


       // *** Logo generator ***
       wire[143:0] cstring_logo = "6.111 Sonar Sensor";
       wire[23:0] logo_pixel;
       char_string_display logo(vclock,hcount,vcount,
                               logo_pixel,cstring_logo,15,15);
       defparam logo.COLOR = 24'b111111111111111111111111;
       defparam logo.NCHAR = 18;
       defparam logo.NCHAR_BITS = 5;


       // *** Uptime calculator ***
       reg[6:0] minutes;
       reg[5:0] seconds;
       wire[7:0] m3_ascii,m2_ascii,m1_ascii,s3_ascii,s2_ascii,s1_ascii;
       binarytodec my_btod_minutes (vclock,{3'b000,minutes},m3_ascii,m2_ascii,m1_ascii);
       binarytodec my_btod_seconds (vclock,{4'b0000,seconds},s3_ascii,s2_ascii,s1_ascii);
       wire[151:0] cstring_uptime =
{8'b01010101,8'b01110000,8'b01110100,8'b01101001,8'b01101101,8'b01100101,8'b00111010,

       m2_ascii,m1_ascii,

       8'b01001101,8'b01101001,8'b01101110,
       s2_ascii,s1_ascii,

       8'b01010011,8'b01100101,8'b01100011};
       wire[23:0] uptime_pixel;
       char_string_display uptime(vclock,hcount,vcount,
                             uptime_pixel,cstring_uptime,15,730);
       defparam uptime.COLOR = 24'b111111111111111111111111;
       defparam uptime.NCHAR = 17;
       defparam uptime.NCHAR_BITS = 5;


       // *** Calibration Display ***
       wire[23:0] calibration_pixel;
       wire[215:0] cstring_calibrate = "Please stand 3m from sensor";
       char_string_display calibrate(vclock,hcount,vcount,
                             calibration_pixel,cstring_calibrate,285,20);
       defparam calibrate.COLOR = 24'b111111111111111111111111;
       defparam calibrate.NCHAR = 27;
       defparam calibrate.NCHAR_BITS = 5;


       // *** AVERAGE SPEED Display ***
       reg[23:0] average_speed;
       wire[23:0] avg_speed_pixel;
       wire [7:0] davg3,davg2,davg1;
       binarytodec my_avg_conv (vclock,average_speed[9:0],davg3,davg2,davg1);
       wire[23:0] cstring_avg_speed = {davg3,davg2,davg1};
       char_string_display avg_speed(vclock,hcount,vcount,
                             avg_speed_pixel,cstring_avg_speed,850,700);
       defparam avg_speed.COLOR = 24'b111001010000000001100110;
       defparam avg_speed.NCHAR = 3;
       defparam avg_speed.NCHAR_BITS = 2;


       // ******************************************************




       // ***************** STAGE 2 PIPELINE ********************

       // Alpha blending dividers
       wire[23:0] sweeper_divider_quotient;
       int_divider sweeper_divider (vclock,sweeper_pixel_p[2],DELTA_PIE,

       sweeper_divider_quotient,s_remainder,s_rfd);

       wire[23:0] grid_divider_quotient;
       int_divider grid_divider (vclock,grid_pixel_p[2],DELTA_PIE,

       grid_divider_quotient,g_remainder,g_rfd);

       wire[23:0] avg_speed_quotient;
       reg[23:0] location_sum;
       int_divider avg_speed_divider (vclock,location_sum,7'b0001010,

       avg_speed_quotient,a_remainder,a_rfd);

       wire[11:0] sqrt_output_1;
       sqrtcomp my_sqrtcomp_1 (dist_squared_p[1],vclock,sqrt_output_1);

       wire[11:0] sqrt_output_2;
       sqrtcomp my_sqrtcomp_2 (dist_squared_p[2],vclock,sqrt_output_2);

       // ******************************************************
```

```
        wire[23:0] alpha_blended;
        reg[11:0] prev_location;
        reg[25:0] timer_counter;
        parameter HPIXELC = 512;
        parameter VPIXELC = 384;
        parameter MAX_TIMER_COUNT = 65000000; // 65MHz Clock
        parameter SPEED_MAX_TIMER_COUNT = 20000000; // Sampling 3 times a second

        always @ (posedge vclock) begin
                /*
                x[10] <= 750;
                y[10] <= 450;
                x[9] <= 750;
                y[9] <= 350;
                x[8] <= 750;
                y[8] <= 250;
                x[7] <= 750;
                y[7] <= 150;
                x[6] <= 750;
                y[6] <= 50;
                */
                x[5] <= {1'b0,x_in_5};
                y[5] <= {2'b00,y_in_5};
                x[4] <= {1'b0,x_in_4};
                y[4] <= {2'b00,y_in_4};
                x[3] <= {1'b0,x_in_3};
                y[3] <= {2'b00,y_in_3};
                x[2] <= {1'b0,x_in_2};
                y[2] <= {2'b00,y_in_2};
                x[1] <= {1'b0,x_in_1};
                y[1] <= {2'b00,y_in_1};

                if (timer_counter == MAX_TIMER_COUNT) begin
                        if (seconds != 59)
                                seconds <= seconds + 1;
                        else begin
                                seconds <= 0;
                                minutes <= minutes + 1;
                        end

                        timer_counter <= 0;
                end else
                        timer_counter <= timer_counter + 1;

                // AVERAGE SPEED FOR OBJECT 2 !!!
                if (timer_counter == 20000000 || timer_counter==40000000 ||
timer_counter==60000000) begin
                        for (i=1; i<10; i=i+1) begin
                                dist_memory[i] <= dist_memory[i-1];
                        end

                        if(prev_location>dist_p_2[6])
                                dist_memory[0] <= (prev_location - dist_p_2[6]);
                        else
                                dist_memory[0] <= (dist_p_2[6] - prev_location);

                        prev_location <= dist_p_2[6];

                        location_sum <= (dist_memory[9] + dist_memory[8] + dist_memory[7] +
dist_memory[6] +
                                                                dist_memory[5] +
dist_memory[4] + dist_memory[3] + dist_memory[2] +
                                                                dist_memory[1] +
dist_memory[0]);
                end

                hsync_p[0] <= hsync;
```

```
               vsync_p[0] <= vsync;
               blank_p[0] <= blank;
               for (i=1; i<PIPELINE_STAGE; i=i+1) begin
                      hsync_p[i] <= hsync_p[i-1];
                      vsync_p[i] <= vsync_p[i-1];
                      blank_p[i] <= blank_p[i-1];
               end

               average_speed <= avg_speed_quotient;
               object_pixel_p_1[2] <= object_pixel_1;
               object_pixel_p_2[2] <= object_pixel_2;
               object_pixel_p_3[2] <= object_pixel_3;
               object_pixel_p_4[2] <= object_pixel_4;
               object_pixel_p_5[2] <= object_pixel_5;
               /*
               for (q=1; q<=10; q=q+1) begin
                      object_pixel_p[q][2] <= object_pixel[q];
               end
               */

               if (x[1]>=HPIXELC && y[1]<VPIXELC)
                      dist_squared_p[1] <= (x[1]-HPIXELC)*(x[1]-HPIXELC) + (VPIXELC-
y[1])*(VPIXELC-y[1]);
               else if (x[1]<HPIXELC && y[1]<VPIXELC)
                      dist_squared_p[1] <= (HPIXELC-x[1])*(HPIXELC-x[1]) + (VPIXELC-
y[1])*(VPIXELC-y[1]);
               else if (x[1]<HPIXELC && y[1]>=VPIXELC)
                      dist_squared_p[1] <= (HPIXELC-x[1])*(HPIXELC-x[1]) + (y[1]-
VPIXELC)*(y[1]-VPIXELC);
               else if (x[1]>=HPIXELC && y[1]>=VPIXELC)
                      dist_squared_p[1] <= (x[1]-HPIXELC)*(x[1]-HPIXELC) + (y[1]-
VPIXELC)*(y[1]-VPIXELC);

               if (x[2]>=HPIXELC && y[2]<VPIXELC)
                      dist_squared_p[2] <= (x[2]-HPIXELC)*(x[2]-HPIXELC) + (VPIXELC-
y[2])*(VPIXELC-y[2]);
               else if (x[2]<HPIXELC && y[2]<VPIXELC)
                      dist_squared_p[2] <= (HPIXELC-x[2])*(HPIXELC-x[2]) + (VPIXELC-
y[2])*(VPIXELC-y[2]);
               else if (x[2]<HPIXELC && y[2]>=VPIXELC)
                      dist_squared_p[2] <= (HPIXELC-x[2])*(HPIXELC-x[2]) + (y[2]-
VPIXELC)*(y[2]-VPIXELC);
               else if (x[2]>=HPIXELC && y[2]>=VPIXELC)
                      dist_squared_p[2] <= (x[2]-HPIXELC)*(x[2]-HPIXELC) + (y[2]-
VPIXELC)*(y[2]-VPIXELC);

               grid_pixel_p[2] <= grid_pixel;
               sweeper_pixel_p[2] <= sweeper_pixel;
               logo_pixel_p[2] <= logo_pixel;
               uptime_pixel_p[2] <= uptime_pixel;
               calibration_pixel_p[2] <= calibration_pixel;
               pie_p[2] <= pie;
               angle_loc_p[2] <= angle_loc;
               menu_pixel_p[4] <= menu_pixel;

               dist_p_1[6] <= sqrt_output_1;
               text_1_pixel_p[7] <= text_1_pixel;
               dist_p_2[6] <= sqrt_output_2;
               text_2_pixel_p[7] <= text_2_pixel;

               for (j=3; j<PIPELINE_STAGE; j=j+1) begin
                      grid_pixel_p[j] <= grid_pixel_p[j-1];
                      sweeper_pixel_p[j] <= sweeper_pixel_p[j-1];
                      logo_pixel_p[j] <= logo_pixel_p[j-1];
                      uptime_pixel_p[j] <= uptime_pixel_p[j-1];
                      calibration_pixel_p[j] <= calibration_pixel_p[j-1];
                      pie_p[j] <= pie_p[j-1];
                      angle_loc_p[j] <= angle_loc_p[j-1];
```

```
                    object_pixel_p_1[j] <= object_pixel_p_1[j-1];
                    object_pixel_p_2[j] <= object_pixel_p_2[j-1];
                    object_pixel_p_3[j] <= object_pixel_p_3[j-1];
                    object_pixel_p_4[j] <= object_pixel_p_4[j-1];
                    object_pixel_p_5[j] <= object_pixel_p_5[j-1];

                    if (j>=5)
                           menu_pixel_p[j] <= menu_pixel_p[j-1];

                    if (j>=8) begin
                           text_1_pixel_p[j] <= text_1_pixel_p[j-1];
                           text_2_pixel_p[j] <= text_2_pixel_p[j-1];
                    end

                    /*
                    for (q=1; q<=10; q=q+1) begin
                           object_pixel_p[q][j] <= object_pixel_p[q][j-1];
                    end
                    */
             end

             sweeper_divider_quotient_p[28] <= sweeper_divider_quotient;
             grid_divider_quotient_p[28] <= grid_divider_quotient;
             alpha_blended_p[29] <= sweeper_divider_quotient_p[28]*angle_loc_p[28] +

       grid_divider_quotient_p[28]*(DELTA_PIE-angle_loc_p[28]);

             for (k=30; k<PIPELINE_STAGE; k=k+1) begin
                    alpha_blended_p[k] <= alpha_blended_p[k-1];
             end

             final_object_pixel_p[30] <= (object_pixel_p_1[29] | object_pixel_p_2[29] |

       object_pixel_p_3[29] | object_pixel_p_4[29] |

       object_pixel_p_5[29]);
             /*
             final_object_pixel_p[30] <= (object_pixel_p[1][29] | object_pixel_p[2][29]
| object_pixel_p[3][29] |

       object_pixel_p[4][29] | object_pixel_p[5][29] | object_pixel_p[6][29] |

       object_pixel_p[7][29] | object_pixel_p[8][29] | object_pixel_p[9][29] |

       object_pixel_p[10][29]);
             */

             final_text_1_pixel_p[30] <= (text_1_pixel_p[29] | text_2_pixel_p[29] |
avg_speed_pixel |

       logo_pixel_p[29] | uptime_pixel_p[29] | menu_pixel_p[29]);

             final_background_pixel_p[30] <= pie_p[29] ?

             alpha_blended_p[29] :

             sweeper_pixel_p[29] ?

                    sweeper_pixel_p[29] :

                    grid_pixel_p[29];

             pixel_p[31] <= mode==1 ?
                                                           (calibration_pixel_p[30] ?
calibration_pixel_p[30] : grid_pixel_p[30]) :
                                                           (final_text_1_pixel_p[30] &&
mode!=3) ?
```

```
        final_text_1_pixel_p[30] :

        final_object_pixel_p[30] ?

        final_object_pixel_p[30] :

        final_background_pixel_p[30];

        end

        /*
        always @ (posedge vsync) begin
                x[1] <= right ? (x[1] + 1) : (left ? (x[1] - 1) : x[1]);
                y[1] <= down ? (y[1] + 1) : (up ? (y[1] - 1) : y[1]);
        end
        */
        assign sound_on = (final_object_pixel_p[30] && sweeper_pixel_p[30]);
   assign phsync = hsync_p[31];
   assign pvsync = vsync_p[31];
   assign pblank = blank_p[31];
        assign pixel = pixel_p[31];

endmodule

//////////////////////////////////////////////////////////////////////
//
// sonargrid: generate circular grid on screen
//
//////////////////////////////////////////////////////////////////////

module sonargrid (hcount,vcount,clock,pixel);
   input [10:0] hcount;
   input [9:0] vcount;
        input clock;
   output [23:0] pixel;

   parameter COLOR = 24'b001011101000101101010111;  // default RGB: 00CC00 GREEN
        parameter BLACK = 24'b000000000000000000000000;      // BLACK
        parameter HPIXELC = 512;
        parameter VPIXELC = 384;
        reg[17:0] addr;
        wire dout;

        brambackground myram(addr, clock, dout);

   //reg [2:0] pixel;
   always @ (hcount or vcount) begin
      if (hcount<HPIXELC && hcount>=0 && vcount<VPIXELC && vcount>=0) begin
                        addr <= (hcount + vcount*HPIXELC);
                end else if (hcount<HPIXELC && vcount>=VPIXELC) begin
                        addr <= (hcount + (VPIXELC - (vcount - VPIXELC + 1))*HPIXELC);
                end else if (hcount>=HPIXELC && vcount<VPIXELC) begin
                        addr <= ((HPIXELC - (hcount - HPIXELC + 1)) + vcount*HPIXELC);
                end else if (hcount>=HPIXELC && vcount>=VPIXELC) begin
                        addr <= ((HPIXELC - (hcount - HPIXELC + 1)) + (VPIXELC - (vcount -
VPIXELC + 1))*HPIXELC);
      end else
                        addr <= 18'b000000000000000000;
   end

        assign pixel = dout ? COLOR : BLACK;
endmodule
```