Virtual Pet

6.111 Introductory Digital Systems Laboratory

Cheng Hau Tong

Abstract

A virtual pet is a digital creature that lives inside a LCD display. This virtual pet, in particular, loves to listen to music, and it dances to the beat detected from the microphone input jack. Like a real pet, the virtual pet needs attention and care from the owner. While it does not need to be fed with real food, the virtual pet needs music in order to stay alive and happy. It has two states—dancing state and idle state. Once it has been in idle state for more than 2 minutes, the virtual pet feel sad its health level decreases. It features lights that flash according to the music beat when it is in dancing state.

Table of Contents

Page

Abstract ... i
List of Tables and Figures ... ii
Overview ... 1
Module Descriptions ... 3
  Beat Detector ... 4
    Beat Detection Algorithms ... 4
    Beat Detector Implementation ... 5
      FFT ... 6
      History BRAM ... 6
      Beat Detector Controller ... 6
  Video Display ... 7
    Dog ... 8
    Health Meter ... 8
    Light ... 8
    Video Controller ... 9
Testing and Debugging ... 10
Conclusion ... 11
Appendix: Verilog code ... 12

List of Tables and Figures

Page

Figure 1: The pattern of light in idle state, happy emotion ... 1
Figure 2: The flashing light in idle state, sad emotion ... 2
Figure 3: Images used to generate a sequence of movement in dancing state ... 2
Figure 4: The pattern of flashing light in dancing state ... 2
Figure 5: Higher lvel Block Diagram of the Virtual Pet ... 3
Figure 6: Spectrum of an example audio signal at time t0 and t1 ... 4
Figure 7: Block Diagram of the Beat Detector ... 5
Figure 8: Block Diagram of the Video Display ... 7
Figure 9: State Transition Diagram of the Dog Module ... 8
Figure 10: State Transition Diagram of Video Controller ... 9
Table 1: Light Flashing Pattern with Different Inputs from Video Controller ... 9

Overview

A virtual pet is an electronic device that simulates the behavior of a pet in real life, and requires a lot of attention and care from the owner. For instance, one of well known virtual pets, tamagotchi, is a pet that lives inside a display sized smaller than a kid's palm. Similar to a real pet, tamagotchi needs to be fed, entertained, and taken care of and its life span depends on the amount the attention the owner has put into. Inspired by tamagotchi, I decided to implement a virtual pet that lives within a color LCD display and loves to listen to music. One of the interesting features of the virtual pet includes its ability to detect a musical beat, and changes it emotion, health, and action according to the beats it percepts.

The user interface is a LCD screen of 1024x768 pixels that contains three video modules. On the left is a dog video that displays a sequence of images of the dog. There are seven circles on the right side of the screen, serving as light to indicate the current emotion of the pet or the beat patterns. Above the seven lights is a health meter that shows the health level of the pet.

When resets, the dog video shows a still image of a dog that is facing front (i.e, the user). The pet starts in idle state and happy emotion. As illustrated in Figure 1, seven lights at the right flash in six phases to show that it is happy.

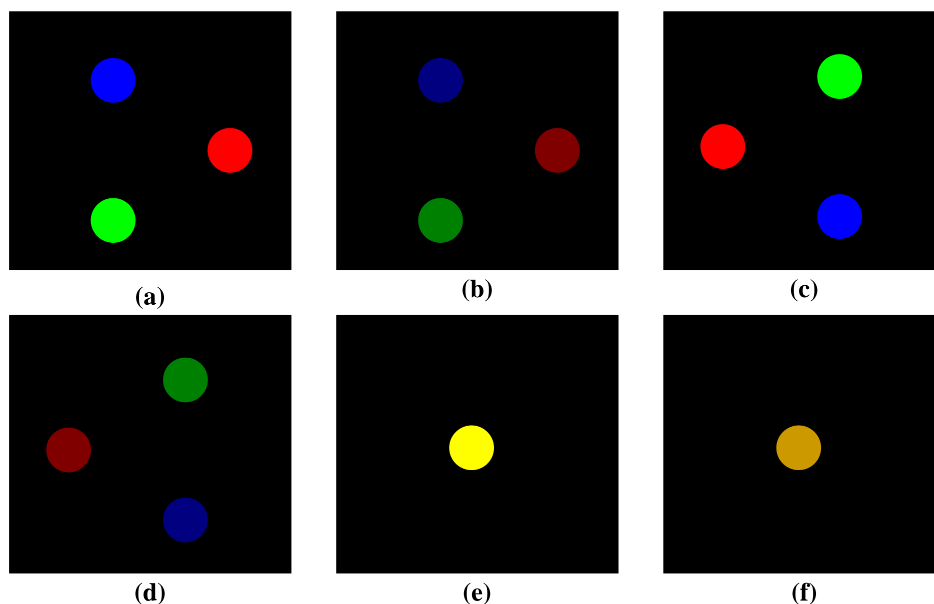


Figure 1The pattern of light in idle state, happy emotion. (a), (b): Three lights of red, green and blue brighten and fade. (c), (d): Three other lights of red, green and blue at different location brighten and fade. (e), (f): A yellow light located at the center of previous lights brightens and fades. The phases always repeat as long as the pet remains in the same state and emotion.

The pet remains in idle state as long as there are less than 10 beats detected in the past one minute. However, once the pet has been in idle state for more than 2 minutes, its emotion changes from happy to sad. This sudden change in emotion is indicated by a different flashing

pattern of the seven lights previously mentioned. Figure 2 shows two phases of the flashing pattern.

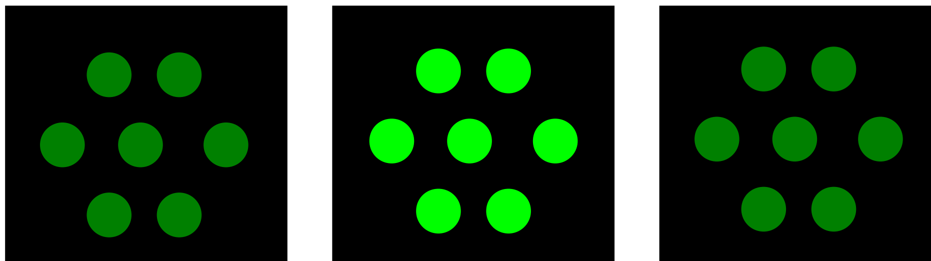
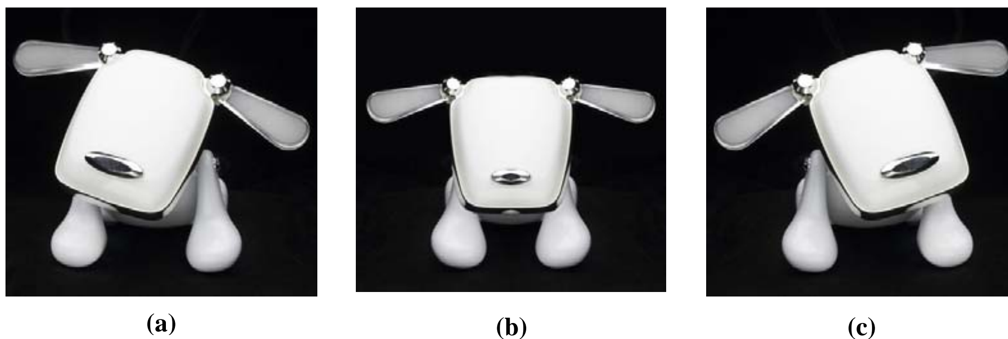


Figure 2 The flashing light in idle state, sad emotion. Seven lights brighten and fade continuously, at the rate slower than when the pet is in happy emotion.

Once the last and the last 10th beats detected are less than 1 minute apart from each other, the virtual pet switches its state to dancing state. The dog video is now showing a sequence of images, instead of a still image, to show the movement of the dog. Figure 3 shows three images used in the dancing state of the pet.



(a) **(b)** **(c)**

Figure 3 Images used to generate a sequence of movement in dancing state. A complete cycle of animation uses the images in the sequence (b)→(a)→(b)→(c)→(b).

Note that in dancing state, the lights no longer show emotion; rather, the lights flash according to the incoming beat, as illustrated in Figure 4.

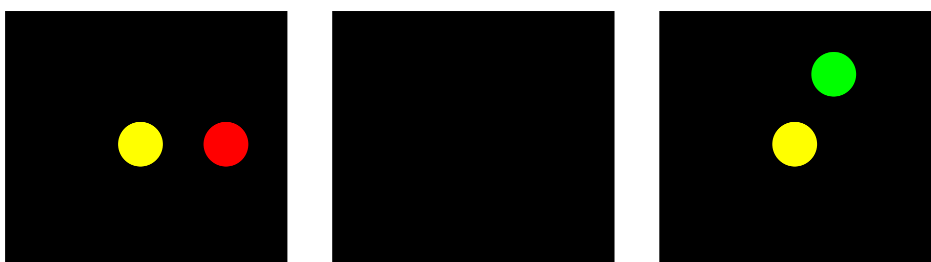


Figure 4The pattern of flashing light in dancing state. Whenever a beat is detected, one light of the six surrounding lights, together with the yellow at the center, flashes once. The lights remain dark until the next beat is detected.

A virtual pet remains in dancing state as long as the last and the last 10th beats detected are still less than 1 minute apart from each other—implying that in dancing state, the dog keeps dancing for 1 minute since the last beat detected.

In order to make the virtual pet behave closer to the pet in real world, the health level of the pet changes over the time, depending on the amount of care it has from the owner. More specifically, for every minute the pet remains in dancing state, its health level increase by 1 unit, until it reaches maximum health level. On the other hand, its health level decreases by 1 unit for every minute it remains in idle state, until it reaches zero. It is impossible to revive a dead dog—once the health level falls to zero, the dog can never recover. At this stage, the only possible thing a user can do is to press the reset button to regenerate life (playing God.)

Modules Description

There are two main parts in this project: Beat detector and Video Display. The beat detector is responsible to take audio data from AC97 chip, and determines if there is a beat in the audio data. It contains a Fast Fourier Transform (**`FFT`**) module, a **`history RAM`**, and a **`beat detector controller`**. If the beat detector detects a beat, it outputs a high pulse **`beat`** to the Video Display. The **`video controller`** module of Video Display observes the pattern of the **`beat`** coming in, and changes the state, emotion and health level of the virtual pet. Consequently, three video modules of the Video Display—**`Dog`**, **`Health_Meter`**, and **`Light`**—outputs appropriate video data to ADV7215 according to the state, emotion and health level determined by **`video controller`**. Please refer to Figure 5 for a higher level block diagram of the virtual pet.

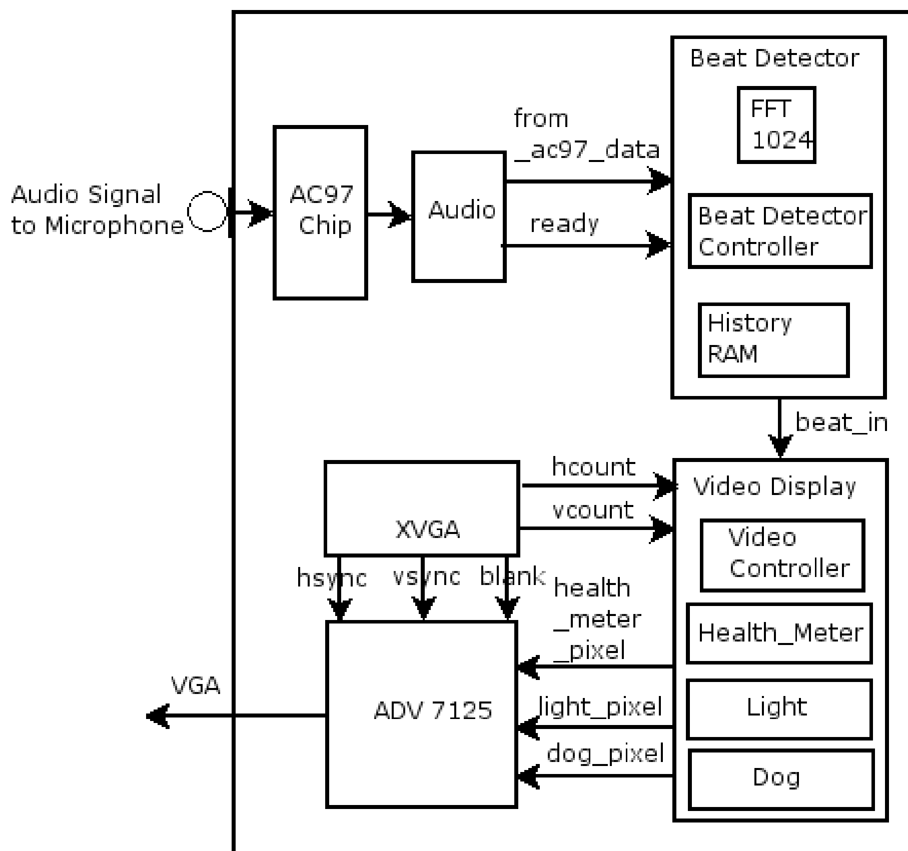


Figure 5 Higher level Block Diagram of the Virtual Pet.

In the following sections, the functionality and implementation of each module are discussed in details.

Beat Detector

Beat Detection Algorithms

Several real time beat detection algorithms that works for different context of audio signal and at different level of accuracy and sophistication have been widely available. One of the simplest algorithms is to observe if there is a sudden increase in the audio signal strength relative to the average signal strength in past one second. The sudden increase has to be a short instant of time such that the interval is sufficient to capture the energy level associated with all the frequencies above minimum human's perception frequency. By setting the short interval to be roughly 0.02s, we are able to capture the energy level at frequencies above 50 Hz. Human's hearing ranges from 20 Hz to 20 kHz so 0.02s interval is fine. With this simple algorithm, a beat is detected when current energy level of an interval 0.02s of the audio signal is greater than a threshold times the average signal energy level in past one second. Note that this algorithm involves summing and comparing signal energy level in time domain only.

However, the beat detection obtained by performing only time domain analysis on audio signal can be less accurate than one that is obtained by observing the spectrogram of the signal—an analysis of the signal in both time and frequency domain. Consider a make-up signal whose spectrum at time t_0 and t_1 is as shown in Figure 6.

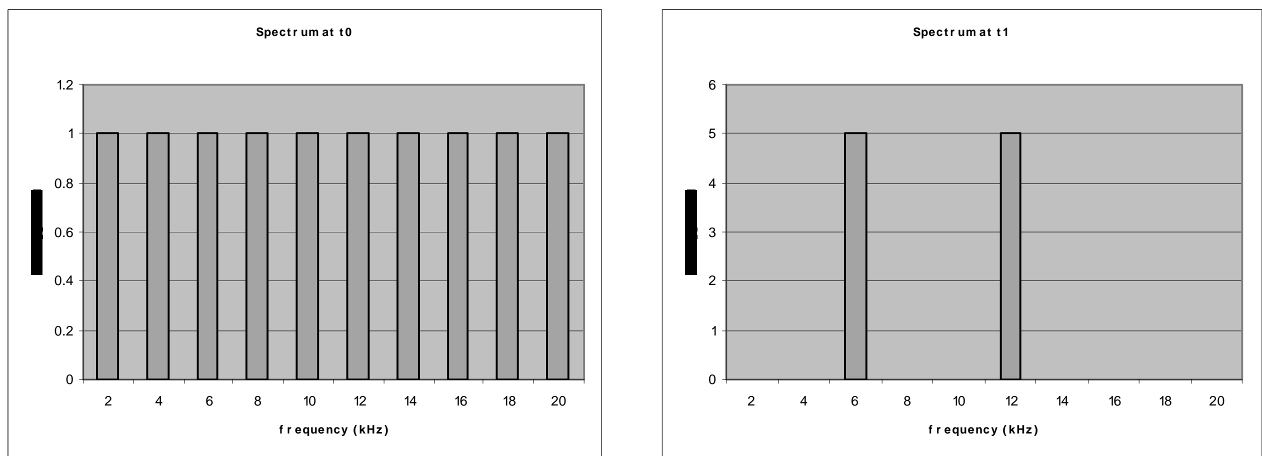


Figure 6 Spectrum of an example audio signal at time t0 and t1.

In time domain, the energy level of the signal at both time instants (obtained by calculating the total area of the graph) is identical—10 units. However, note that there is an abrupt increase in the energy level at frequency band 6 kHz and 12 kHz from time t_0 to t_1.The beat detection done in time domain only is not able to detect this phenomenon because the change in energy level is zero. However, a beat detection performed in frequency domain at different time instant is able to detect this case fairly well.

Below details the algorithm:

1. Transform the data into frequency domain to obtain the spectrum (energy level) of each frequency band for an interval of 0.02s.
2. For each frequency band, compared the energy level of the interval to the average energy level of the signal in past one second.
3. If there is an abrupt increase in energy level, declare a beat at that frequency band.

So now we have beat detection at each frequency band. But how do we declare a beat as a whole? Meaning, given that we see a beat at two frequency band, say 6 kHz and 12 kHz, do we declare that there was a beat for that 0.02s interval? It is inaccurate to declare a beat as a whole whenever we detect a beat at any frequency band. Musical audio signal often has not only instruments rhythms, but also human's singing, which can be considered as a noise to the musical beats. One possible way to declare a beat as a whole given the beat detection at each frequency band is to observe only the frequency below 100 Hz, which is roughly the frequency range of a drum. However, this method explicitly assumes that the rhythms contain drum beats. A more general algorithm is to declare a beat if more than a certain percentage, say 25%, of the frequency bands detects a beat.

Beat Detector Implementation

Given the algorithm described above, there are two hardwares required to perform the beat detection: a `Fast Fourier Transform` to transform the signal into frequency domain, and a memory to remember the sum of energy level in past one second for each frequency band. In order to update the sum of energy level in past one second whenever the new data is available, we need to subtract the expired data (the energy level of the audio signal one second ago) from the current sum of energy level in past one second and add the energy level of the new data. This step needs to be performed on all frequency bands. Therefore, we also need another memory `History BRAM` to remember the 0.02s-interval energy level of each frequency band for the past one second. Figure 7 shows the block diagram of the beat detector.

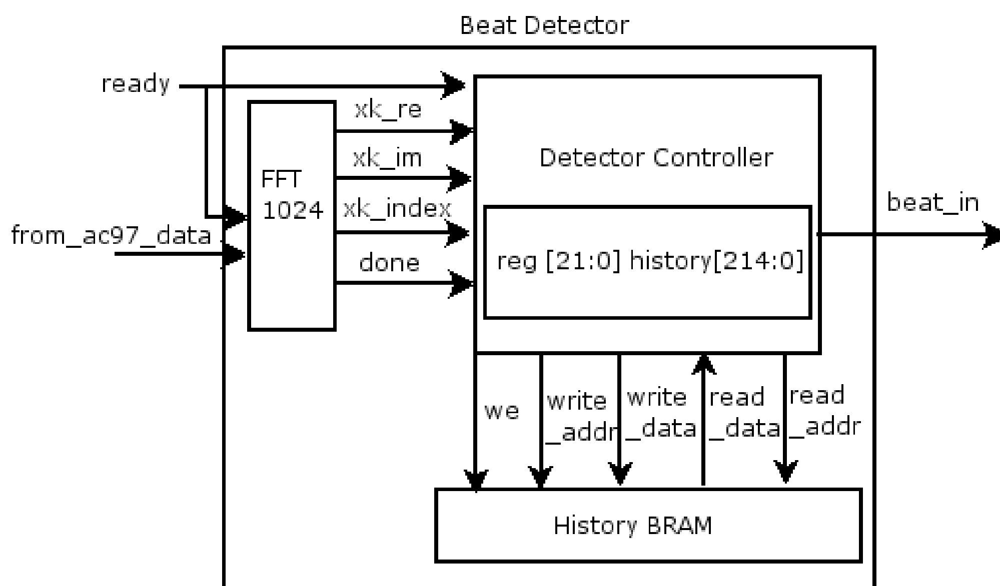


Figure 7 Block Diagram of the Beat Detector.

Fast Fourier Transform FFT

The 0.02s interval and the fact that AC97 samples the data at 48 kHz tell us that we need a 1024-point **FFT**:

$$\frac{1024 \text{ points}}{48000} \sim 0.02 \text{ second}$$

FFT takes 8-bit signed audio data **from_ac97_data** as input, transform the data, and output the real part of the data in frequency domain **xk_re**, the imaginary part of the data in frequency domain **xk_im** for the corresponding transform index **xk_index**. The output data from **FFT** is in natural order. **FFT** runs synchronously with the global **clock_65mhz**, but **from_ac97_data** is only sampled at 48 kHz. Therefore, **FFT** has a pin **CE** to enable the clock only when there is a new audio data (i.e. at the rising edge of **ready** signal).

History BRAM

History BRAM is a dual-port BRAM that has a read-only port and a write-only port. The spectrum of each frequency band is calculated by adding the square of **xk_re** and **xk_im**. The spectrum is symmetric about **xk_index** 513 and **xk_index** 0 to 513 corresponds to the frequency band 0 to 48 kHz, with each index representing a frequency band of ~94 Hz wide. Since human's perception range is from 20Hz to 20 kHz, only the spectrum of index 0 to 214 is relevant to the beat detection. The size of the memory to store of the 0.02s-interval spectrum (energy level) of each frequency band for past one second is therefore:

Width: 16 bit

The square of **xk_re** and **xk_im** has 16 bit signed each, which is equivalent to 15 bit unsigned each. The sum of them is hence 16 bit unsigned.

Depth: 10320 = 48x215

$\frac{48000}{1024 \text{ points}}$ ~48. There are 215 relevant frequency bands.

Beat Detector Controller

Beat Detector Controller mainly controls how the data is read from and written to the memories, determines if there is a beat at each frequency band, and eventually declares a beat (as a whole) by observing the number of frequency band that detects a beat.

Beat Detector Controller contains a register memory array **history** of width 22 bit and depth 215 to store the sum of energy level in past one second for each of the 215 frequency bands. Whenever new data is ready from **FFT**, the controller calculates the spectrum of the corresponding **xk_index,** and store the spectrum into History BRAM, provided that the **xk_index** is between 0 and 214.

At the rising edge of **ready** signal, for the corresponding **xk_index** between 0 and 214, the controller updates the memory history[**xk_index**] by subtracting the expired data from history[**xk_index**] and adding spectrum to history[**xk_index**]. Expired data is obtained from **History BRAM**. There is a clock cycle delay for reading data from BRAM. Therefore, at the

rising edge of **ready** signal, I need to assign the read address so that it retrieves data for the coming **xk_index**, not for the current **xk_index.**
Also at the rising edge of **ready** signal, the controller compares if 48 times the current spectrum for the corresponding **xk_index** between 0 and 214 is larger than a threshold constant times the sum of energy level in past one second, history[**xk_index**]. If so, it detects a beat at that particular **xk_index**. Currently, the threshold constant is set to 1.5.

At the rising edge of the **ready** signal, the controller counts the number of frequency bands that have detected beats. If more than 50 of the 215 bands detected beats, the beat detector controller finally declares a beat (as a whole).

Video Display

Video Display controls all the video output to the LCD display. The video modules work on a 1024x768 display, at clock frequency 65 MHz. There are 3 video modules: **Dog**, **Health Meter**, and **Light**, and a **Video Controller**, as shown in the block diagram in Figure 8.

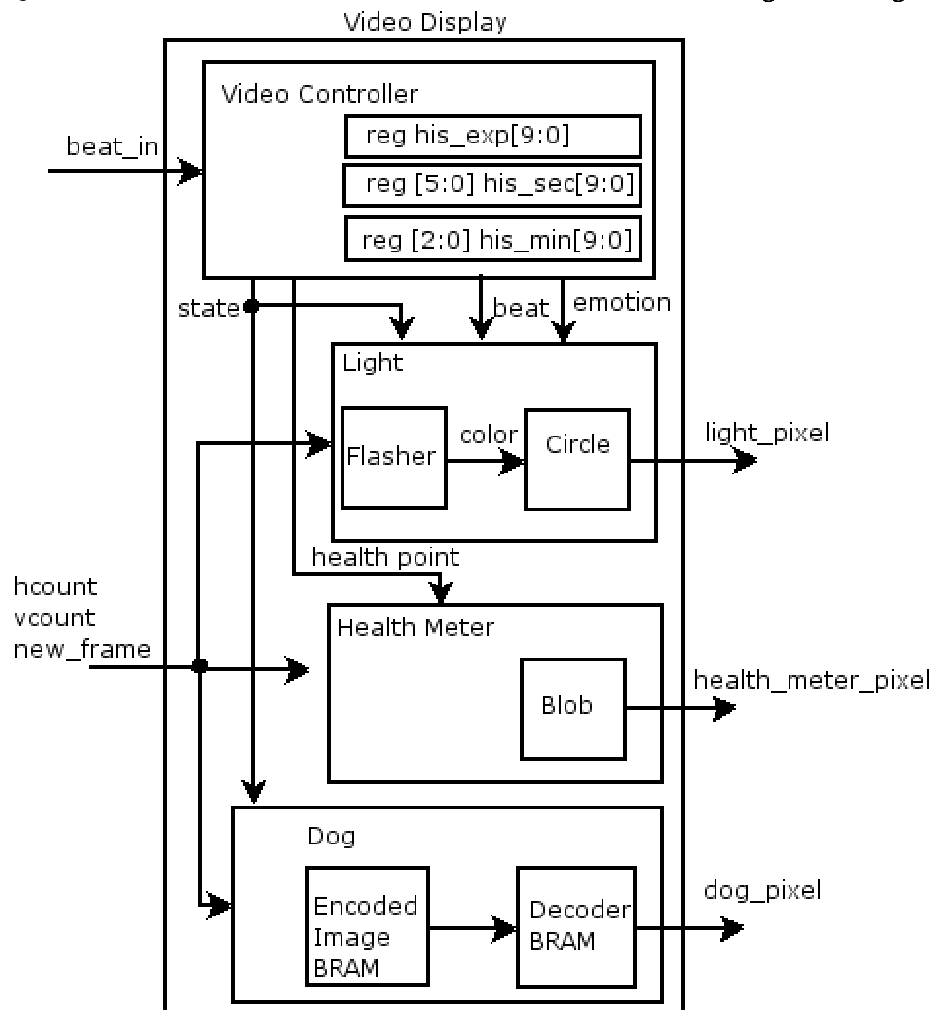


Figure 8 Block Diagram of the Video Display.

Dog
As its name suggest, `Dog` module displays the video of the pet. In order to make the dog dances, three 380x310 pixels images are used. These images are encoded in 5 bits of colors and stored in three different ROMs, each of width 5 bits and depth 117800. To retrieve the RGB value of the 32 colors, the read data from the image ROM feeds into the memory address of the decoders ROMs. The output of the decoder ROM is 24 bit RGB pixel values. Because of cascaded ROMs, there are delays from reading the pixel values. This was fixed by reading the data from the ROM in advance. In idle state, the dog does not move so this module only read the data from the Center image ROM. In dancing state, the dog turns left, then face center, turns right, and face center again. A counter, which increases its value by 1 at the arrival of new video frame, is used to implement this transition. Whenever the counter reaches its maximum value, it increases `count_rom` by 1. Figure 9 illustrates that the dog video display encoded video data from Center image ROM when the `count_rom` is equal to 0 or 2.

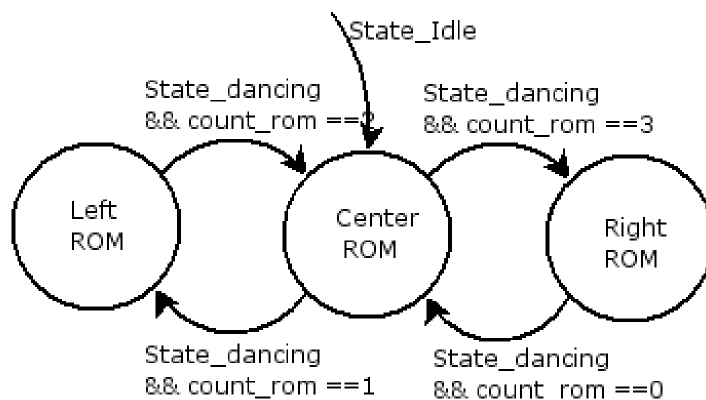


Figure 9 State transition diagram of the Dog module.

Health Meter
This module is responsible to report the current health level of the pet. It takes `health_point` from the `Video Controller` and display a gradient bar of width 16 times the value of the `health_point.` The color of the health meter is green when the `health_point` is greater than 8, yellow when between 8 and 5 inclusively, and red when it falls less than or equal to 4. In order to implement the vertical gradient (brightest at horizontal middle line of the bar, and darkest at the top and bottom of the bar), a parameter increment is declared. For every pixel $2^{increment}$ apart from the y-middle line of the bar, R, G and B values of the color shift right by one unit, and this method decreases the luminance of the R, G and B element of the color.

Light
Light is the unit that indicates several information of the pet: emotion when the pet is idle and beat detection when the pet is dancing. Light module, therefore, requires three inputs from the video controller— `state`, `emotion`, and `beat`, each of which is one bit width.

Following is a table showing the output patterns corresponding to the possible inputs: It shows that the lights are independent of emotion when it is in dancing state, and independent of insignificant beat detection. (By insignificant, it means the last tenth beat and the last beat are more than 1 minute apart from each other.)

Table 1: Light Flashing Pattern with Different Inputs from Video Controller

State	Emotion	Beat	Pattern
Idle	Sad	x*	Pattern 1
	Happy	x*	Pattern 2
Dancing	Sad	0	Blank
	Happy	0	
	Sad	1	Pattern3
	Happy	1	

* Note: As long as the pet remains in idle state, lights do not indicate beat detection.

In order to implement the lights, submodules `circle` and `flasher` were written. Circle is a module that takes coordinate of the center point and color as inputs, and displays a circle on the screen. Flasher is a module that takes `color_in`, `increment`, `increase`, `frame_rate` and output `color_out`. For every `frame_rate` frames, `flasher` increases the value of each of R, G and B elements of `color_in` by `increment` unit, if the input `increase` is high. Otherwise, it decreases the values of each of R, G and B elements of `color_in` by `increment` unit. By setting the `frame_rate` and increment appropriately, I could implement smooth transition of color luminance and make the circles flashing.

Video Controller

Taking only `beat_in` as input from the `Beat Detector`, `Video Controller` is responsible to determine three outputs: `state`, `emotion`, and `health_point`. `Beat Detector` is only responsible to determine in real time if the energy level at different frequency bands has increased abruptly relatively to the average energy level of past one second. To avoid unexpected noise, instead of changing to dancing state everytime the beat detector detects a beat, `Video Controller` observes the interval between last ten beats in order to determine whether to change the states of the virtual pet. If the interval is greater than 1 minute apart, the beats detected by the beat detector are not significant enough, provided that the beat detector has been appropriately calibrated.

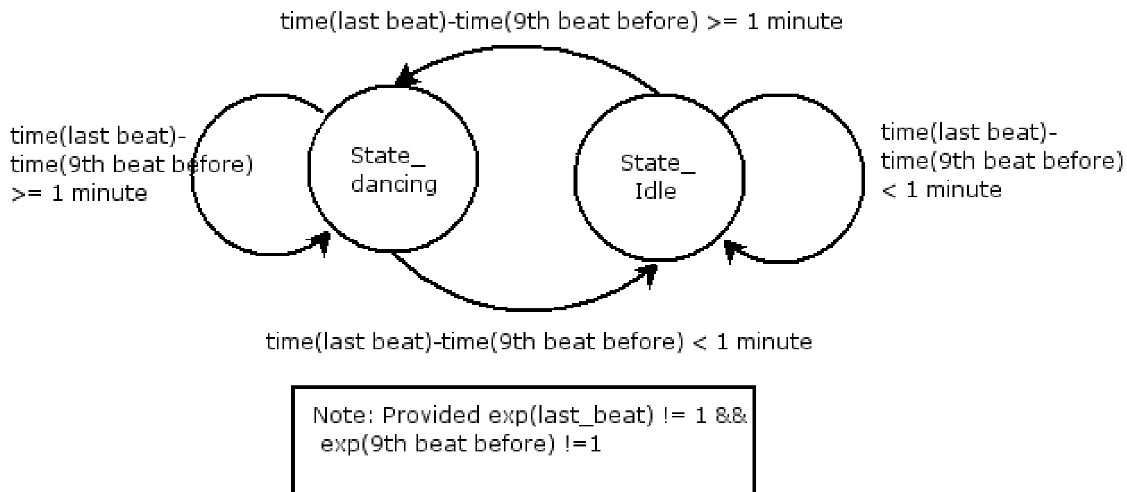


Figure 10 State Transition Diagram of Video Controller.

In order to implement this state transition, I coded a timer/counter that associates each beat received from beat detector with a time value. It is impossible to keep counting the time forever—imagine if the beat never arrives at the input port after the arrival of the first beat, the counter would have to count till infinity. In fact, the counter implemented only counts up to 5 minutes. In order to fix that, each beat is also associated with a register expired. If the value of the register has not changed in past one minute, the beat is expired, and time value associated with that register is no longer valid. The implication of this implementation is that the pet will remain in dancing state for 1 minute even when there is no **`beat_in`** from the beat detector.

While state indicates that the number of beats in past one minute is significant, emotion indicates the length the virtual has been in a particular state. When the state is dancing, the emotion is always happy. When the state switches from dancing to idle, the dog remains happy for 2 minutes and becomes sad as long as it remains in idle. In order to implement this, this module needs a counter to remember the length it has been in a particular state. The counter resets to zero whenever the virtual pet switches its state.

Finally, video controller also needs to determine the health point of the pet. When resets, the health point is at its maximum, which is at 15. For every minute it remains in idle state, the **`health_point`** reduces by 1 point until it reaches zero. Once the **`health_point`** is zero, it can never increase anymore in the future, regardless if it changes its state to dancing. In other words, the pet is now dead. On the other hand, for every minute the pet remains in dancing state, **`health_point`** increases by 1 point until it reaches maximum again.

Testing and Debugging

The testing of the video modules was mostly done by running the labkit and observing if the video output matches the expected behavior. One of the issues I had was with the logic delay. Because of the high clock speed (65MHz), the logic has little time to do complex logic. For instance, the circle module was not able to display a nice circle because it requires multiplication and addition of the multiplication product.

At the beginning of the stage, the data from my image ROMs and decoder ROMs seem to show some kind of delay. I figured out that was because the read request need to go through two ROMs before outputting the data, and each ROM itself has one clock cycle delay. This problem was fixed by reading the data from the ROM in advance.

The testing of Beat Detector was mostly done by observing the labkit internal signal on the logic analyzer. At the early stage, I was stuck very long because the FFT was not working. It turned out that the clock enable pin (CE) was not registered enough. I defined CE as

```
always @(posedge vclock)
ready_old <= ready;
assign ce = ready & ~ready_old;
```

This was causing problem because the signal ready from AC97 is not synchronous to vclock. This problem was fixed by adding one more register to the ready signal input:

```
always @(posedge vclock) begin
ready_old <= rdy;
rdy <=ready;
end
assign ce = rdy & ~ready_old;
```

I verified that my FFT worked by feeding in a sin wave of 750 Hz and observed the spectrum of the FFT output. It showed two spikes near the xk_index 16 and 1007 and that was correct. However, the beat detector failed to work as a whole because the beat detector controller was not working. I suspected that it was because of the logic delay for doing several multiplication to determine if the spectrum is greater than a constant times the average energy level in past one minute.

Conclusion

I thought this final project has been a really rewarding and educational experience for me personally, even though I did not have my beat detector working eventually.

I was over worried that the beat detection algorithm was not good enough and my final project would turn out somewhat worthless because the beat detector really was the soul of the entire project. I tested several algorithms on MATLAB but failed to see one that works consistently with different kind of songs.

For future work, I would like to add more features (analog devices) to the virtual pet. Perhaps, an optical device to detect a pat from the user. Or maybe something to do with the video input. I felt that the pet lacks the liveliness I was expecting.

Appendix: Verilog Code

final2.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 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 final2   (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
              ac97_bit_clock,

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

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

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

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

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

clock_feedback_out, clock_feedback_in,

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

rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

mouse_clock, mouse_data, keyboard_clock, keyboard_data,

clock_27mhz, clock1, clock2,

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

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

switch,

led,
```

```
          user1, user2, user3, user4,

          daughtercard,

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

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

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

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

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

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

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

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

inout [43:0] daughtercard;

inout  [15:0] systemace_data;
output [6:0]  systemace_address;
output systemace_ce_b, systemace_we_b, systemace_oe_b;
```

```
input  systemace_irq, systemace_mpbrdy;

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

////////////////////////////////////////////////////////////////////////////
//
// I/O Assignments
    //
    // Modifications made:-
    // 12/1/05: disable default audio input and output assignments.
//
////////////////////////////////////////////////////////////////////////////

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

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

// Video Input
assign tv_in_i2c_clock = 1'b0;
```

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

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

// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 24'h0;
assign flash_ce_b = 1'b1;
assign flash_oe_b = 1'b1;
```

```
assign flash_we_b = 1'b1;
assign flash_reset_b = 1'b0;
assign flash_byte_b = 1'b1;
// flash_sts is an input

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;
// rs232_rxd and rs232_cts are inputs

// PS/2 Ports
// mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

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

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

// User I/Os
assign user1 = 32'hZ;
assign user2 = 32'hZ;
assign user3 = 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 = 16'h0;
//assign analyzer1_clock = 1'b1;
//assign analyzer2_data = 16'h0;
//assign analyzer2_clock = 1'b1;
assign analyzer3_data = 16'h0;
assign analyzer3_clock = 1'b1;
assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;


////////////////////////////////////////////////////////////////////////////
//
// Clock and Video Setup
//
////////////////////////////////////////////////////////////////////////////

// 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;

   // 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);



    ////////////////////////////////////////////////////////////////////////////
//
// Buttons Setup
//
////////////////////////////////////////////////////////////////////////////

     // 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
wire up,down;
debounce db2(reset, clock_65mhz, ~button_up, up);
debounce db3(reset, clock_65mhz, ~button_down, down);


    ////////////////////////////////////////////////////////////////////////////
//
// Audio Setup
//
////////////////////////////////////////////////////////////////////////////
```

```
    wire [7:0] from_ac97_data, to_ac97_data;
wire ready;

// AC97 driver
audio a(clock_65mhz, reset, from_ac97_data, to_ac97_data, ready,
       audio_reset_b, ac97_sdata_out, ac97_sdata_in,
       ac97_synch, ac97_bit_clock);

    wire signed [19:0] pcm_data;
    tone750hz tone(clock_65mhz, ready, pcm_data);

    ///////////////////////////////////////////
    //
// Beat Detector
//
    ///////////////////////////////////////////

    wire [9:0] scale_sch = 10'b1010101010;
    wire overflow;
    wire [7:0] xk_re;
    wire [7:0] xk_im;
    wire [9:0] xk_index;
    wire ce;
    wire [9:0] xn_index;
    wire done;
    wire beat_in;

    wire [13:0] write_addr;
    wire [13:0] read_addr;
    wire [15:0] read_data;
    wire [15:0] write_data;

    wire [15:0] spectrum;
    wire fft_oready;
    wire ram_start;
```

```
	wire data_useful;
	wire [15:0] old_data;
	wire [20:0] history0, history1,history214;


	wire [7:0] band = 8'd50;
	wire [4:0] con = switch[7:3];


	beat_detector
beatbeat(reset,clock_65mhz,ready,/*pcm_data[19:12]*/from_ac97_data,beat_in,overflow,sc
ale_sch, band, con  //final ports


	,xk_re,xk_index,ce,xn_index,done,xk_im   //fft testing outputs


	,write_addr,read_addr,write_data,read_data     //bram testing outputs


	,spectrum,fft_oready,ram_start,data_useful,old_data,history0, history1,
history214   //detector testing outputs


	);
//
	//next testing:- dclk: thinking that dclk is not working.
	//next stage:- write video fsm even if fft doesnt work
	//



	wire beat,state, emotion;
	wire [3:0] health_point;






	video_controller vc(clock_65mhz, reset, beat_in, health_point, state, emotion,
beat);


	///////////////////////////////////////////
	//
   // Dancing_Dog video
   //
```

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

wire [23:0] dancing_dog_pixel;

   wire phsync,pvsync,pblank;

//	wire [23:0] data_in;
//	wire [16:0] addr;
   //dog
dog1(clock_65mhz,reset,hcount,vcount,hsync,vsync,blank,phsync,pvsync,pblank,dancing_do
g_pixel, data_in, addr, switch[7:6]);
	dog
dog1(clock_65mhz,reset,hcount,vcount,hsync,vsync,blank,phsync,pvsync,pblank,dancing_do
g_pixel,/*switch[6]*/state,data_in,addr);
	///////////////////////////////////////////
	//
   // Health_Meter video
   //
	//////////////////////////////////////////

	wire [23:0] health_meter_pixel;
	wire hphsync,hpvsync,hpblank;
	health_meter
h1(clock_65mhz,reset,hcount,vcount,hsync,vsync,blank,hphsync,hpvsync,hpblank,health_me
ter_pixel, health_point);

	//////////////////////////////////////////
	//
   // Light video
   //
	//////////////////////////////////////////

	wire [23:0] light_pixel;
	wire lphsync,lpvsync,lpblank;
//	light
l1(clock_65mhz,reset,hcount,vcount,hsync,vsync,blank,lphsync,lpvsync,lpblank,light_pix
el, emotion, state, beat);
	light_light
l1(clock_65mhz,reset,hcount,vcount,hsync,vsync,blank,lphsync,lpvsync,lpblank,light_pix
el,emotion, state, beat);/*switch[7],switch[6],switch[5]*///);
```

```
	////////////////////////////////////////////
	//
   // Freq video
   //
	////////////////////////////////////////////
  /*
	wire [23:0] freq_pixel;
	wire fphsync,fpvsync,fpblank;
	wire [7:0] spectrumm;

	graph_video
graph(ready,/*pcm_data[19:12]*///from_ac97_data,clock_65mhz,reset,hcount,vcount,hsync,
vsync,blank,fphsync,fpvsync,fpblank,freq_pixel, spectrumm);
	//fre_video
freq1(ready,from_ac97_data,clock_65mhz,reset,hcount,vcount,hsync,vsync,blank,fphsync,f
pvsync,fpblank,freq_pixel);


	//Combine video modules
	wire   [23:0] pixel = light_pixel + dancing_dog_pixel+
health_meter_pixel;//dancing_dog_pixel ;//+ health_meter_pixel + light_pixel;*/


	/////////////////////////////////////////////////////////////////////////////////
///////////
	//
	// switch[1:0] selects which video generator to use:
   //  00: user's virtual-pet
   //  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
      if (switch[1:0] == 2'b01) begin
	 // 1 pixel outline of visible area (white)
	 hs <= hsync;
	 vs <= vsync;
```

```
        b <= blank;
        rgb <= (hcount==0 | hcount==1023 | vcount==0 | vcount==767) ? 16777215 : 0;
      end else if (switch[1:0] == 2'b10) begin
        // color bars
//      hs <= fphsync;
//     vs <= fpvsync;
//      b <=  fpblank;
//      rgb <= freq_pixel;
      end else begin
   // default: virtual-pet
        hs <= phsync;
        vs <= pvsync;
        b <= pblank;
        rgb <= pixel;
      end
   end

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
   assign vga_out_red = 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 = ~{3'b000,up,down,reset,switch[1:0]};

       ////////////////////////////////////////////////////////////////////////////
   //
   // Logic Analyzer Setup
   //
   ////////////////////////////////////////////////////////////////////////////
       // output useful things to the logic analyzer connectors
```

```
   assign analyzer1_clock = ac97_bit_clock;
   assign analyzer1_data[0] = audio_reset_b;
   assign analyzer1_data[1] = ac97_sdata_out;
   assign analyzer1_data[2] = ac97_sdata_in;
   assign analyzer1_data[3] = ac97_synch;
   assign analyzer1_data[15:4] = 0;

   assign analyzer2_clock = ready;
// assign analyzer2_data[15:0] = 0;
  assign analyzer2_data[15:0] = { spectrum[7:0],
history0[6:0]/*xk_index[9:3]*/,/*to_ac97_data,from_ac97_data};*/ beat};

       assign to_ac97_data = from_ac97_data;

       wire [35:0] control0;
       reg rdy;
       reg old_ready;
       always @(posedge clock_65mhz) begin
       rdy <= ready;
       old_ready <= rdy;
       end
       wire dclk = ~old_ready & rdy;

       wire [31:0] test32 = {xk_index, xk_re, xk_im, 6'd0};
       icon ico (.control0(control0));

       ila ila1 (.control(control0), .clk(dclk), .trig0(test32));
       endmodule

       module icon
  (
      control0
  );
  output [35:0] control0;
endmodule
```

```
module ila
  (
    control,
    clk,
    trig0
  );
  input [35:0] control;
  input clk;
  input [31:0] trig0;
endmodule
```

beat_detector.v

```
module
beat_detector(reset,vclock,ready,from_ac97_data,beat,overflow,scale_sch,band,con
       //final ports

,xk_re,xk_index,ce,xn_index,done,xk_im   //fft testing outputs

,write_addr,read_addr,write_data,read_data     //bram testing outputs

,spectrum,fft_oready,ram_start,data_useful,old_data,history0, history1, history214
       //detector testing outputs

);
       input reset;
       input vclock, ready;
       input [7:0] from_ac97_data;
       output beat;
       output overflow;
       input [9:0] scale_sch;     //values tested: 10'b0110101010 causes overflow,
10'b1010101010 works just fine
       input [7:0] band;
       input [4:0] con;


       output [7:0] xk_re;
```

```
output [9:0] xk_index;
output ce;
output done;
output [9:0] xn_index;
output [7:0] xk_im;

output [13:0] write_addr;
output [13:0] read_addr;
output [15:0] read_data;
output [15:0] write_data;

output [15:0] spectrum;
output fft_oready;
output ram_start;
output data_useful;
output [15:0] old_data;
output [20:0] history0, history1, history214;
reg rdy;
reg ready_old;
wire dclk = ~ready_old & rdy; //rising edge of ready
always @(posedge vclock)
begin
ready_old <= rdy;
rdy    <= ready;  end
//wires for fft
wire ce = dclk; //feed fft when there is new audio data
wire rfd;
wire fwd_inv = 1'b1; //never does inverse fourier
wire dv;
wire done;     //done at xk_index=1023
wire busy;
wire edone;
wire overflow;
wire [7:0] xn_im = 8'b0;
wire signed [7:0] xn_re = from_ac97_data;
wire signed [7:0] xk_im;
```

```
    wire signed [7:0] xk_re;
  wire [9:0] xn_index;      //fft input index
    wire [9:0] xk_index; //fft output index

    //initialize fft
    //reset: nothing
    //1st ready: fwd_inv_we
    //2nd ready: scale_sch_we
    //3rd ready: start fft
    reg [1:0] initclk;
  always @(posedge vclock)
   initclk <= reset ? 0 : (((initclk<3) & ce) ? initclk + 1 : initclk);
  wire fwd_inv_we = (initclk==2'd1);
  wire start = (initclk==2'd3);

  // reload scale factors when changed as well as at init time
  reg [7:0]   old_scale_sch;
  always @(posedge vclock) old_scale_sch <= scale_sch;
  assign scale_sch_we = initclk[0] | ~(scale_sch==old_scale_sch);

    wire nfft_we = (initclk==2'd1);
    wire [4:0] nfft = 5'd10;

    fft_1024
fft(xn_re,xn_im,start,fwd_inv,fwd_inv_we,scale_sch,scale_sch_we,ce,vclock,xk_re,xk_im,
xn_index,xk_index,rfd,busy,dv,edone,done,overflow);
//    fft_1024
fft(xn_re,xn_im,start,nfft,nfft_we,fwd_inv,fwd_inv_we,scale_sch,scale_sch_we,ce,clk,xk
_re,xk_im,xn_index,xk_index,rfd,busy,dv,edone,done,overflow);
//    fft_1024
fft(ce,fwd_inv_we,rfd,start,fwd_inv,dv,scale_sch_we,done,vclock,busy,edone,overflow,sc
ale_sch,xn_re,xk_im,xn_index,xk_re,xn_im,xk_index);

    wire [13:0] write_addr;
    wire [13:0] read_addr;
    wire [15:0] read_data;
    wire [15:0] write_data;
    wire [15:0] spectrum;
    wire fft_oready;
```

```
	wire ram_start;

	wire data_useful;

	wire [15:0] old_data;

	wire [20:0] history0, history1, history214;

	wire we;


	history_ram spechis(read_addr,write_addr,vclock,vclock,write_data,read_data,we);


	detector_controller_2
control(reset,vclock,dclk,xk_index,xk_re,xk_im,done,beat,band,con,read_addr,write_addr
,write_data,read_data,we,spectrum,fft_oready,ram_start,data_useful,old_data,history0,
history1, history214);


endmodule
```

beat_detector_controller.v

```
module detector_controller_2(reset,vclock,dclk,xk_index,xk_re,xk_im,fft_done, beat,
band, con


,read_addr,write_addr,write_data,read_data,we


,spectrum,fft_oready,ram_start,data_useful,old_data


,history0, history1, history214


);

	input reset;

	input vclock; //system clock

	input dclk;    //fft-data clock

	input [9:0] xk_index; //for 1024-pt fft

	input [7:0] xk_re, xk_im;

	input fft_done; //when xk-index=1023


	output [13:0] read_addr;

	output [13:0] write_addr;

	output [15:0] write_data;

	input [15:0] read_data;

	output we;
```

```
output [15:0] spectrum;
output fft_oready;
output ram_start;
output data_useful;
output [15:0] old_data;
output [20:0] history0, history1, history214;
output beat;
input [7:0] band;
input [4:0] con;

//0..512 represents frequency 0..sampling freq(48khz for ac97)
//however, human's hearing range is about 0..20khz, there fore index 0..214
//the spectrum = xk_re*xk_im is symetric about index 512.
//therefore data from index 513 onwards is redundant
//and data from index 215 onwards is irrelevant

parameter NX = 10'd1023;
parameter NX_useful = 8'd214;
parameter n = 47; //time index 48*1024 samples ~ 1 seconds
wire data_useful = (xk_index <=NX_useful);

//fft-oready: falling-edge of fft-done
reg old_done;
always @(posedge vclock) old_done <= fft_done;
wire fft_oready = old_done & ~fft_done;

reg ram_start; //the falling edge of the first done from fft
always @(posedge vclock)
ram_start <= reset? 0: (fft_oready? 1: ram_start);

//calculate spectrum
wire signed [7:0] xk_re;
wire signed [7:0] xk_im;
wire signed [16:0] fft_xk_sq = xk_re*xk_re + xk_im*xk_im;
```

```
	wire [15:0] spectrum = fft_xk_sq[15:0]; //ok because fft_xk_sq is always
positive and not more than 16-bit



	reg [15:0] old_data;//data to delete from history
	always @(posedge vclock)
	old_data <= dclk? read_data : old_data;

	reg [20:0] history[NX_useful:0] ; //the sum of the spectrum of each frequency
band for past
	//wire [20:0] new_history = history[xk_index] + spectrum - old_data;
	integer i;
	always @(posedge vclock)
	begin
		if(reset)
		for(i=0; i<=NX_useful; i=i+1)
		history[i] <= 0;

		else if(data_useful&&ram_start)
		history[xk_index]<= dclk? history[xk_index] + spectrum - old_data:
history[xk_index];
		else begin end
	end

	initial old_data <= 0;
	reg [13:0] read_addr, write_addr;
	reg [15:0] write_data;
	parameter max_addr = 429;//10319;
	wire [13:0] next_read_addr = ~ram_start? 1 :
						(read_addr==max_addr )? (xk_index
== NX)? 1: max_addr: (dclk&&data_useful)? (read_addr+1):
						read_addr;
	wire [13:0] next_write_addr =  ~ram_start? 0:
						(write_addr==max_addr)? (xk_index
== NX)? 0: max_addr: (dclk&&data_useful)? (write_addr+1):
						write_addr;
	always @(posedge vclock) begin
	read_addr <= next_read_addr;
```

```
	write_addr <= next_write_addr;

	write_data <= data_useful? spectrum: write_data;
	end


	//assign write_data = spectrum;
	assign we = data_useful;

	assign history0 = history[0];
	assign history1 = history[1];
	assign history214 = history[214];

	reg beat_temp;

	always @(posedge vclock)
	beat_temp <= (dclk && data_useful)? (48*spectrum >
((con*history[xk_index])>>3)) : 0;

	reg [7:0] beatbuf;

	always @ (posedge vclock)
	beatbuf <= (reset ||fft_done)? 0: beat_temp? beatbuf + 1: beatbuf;

	assign beat = (beatbuf > band);

endmodule
```

video_controller.v

```
module video_controller(clk, reset, beat, health_point, state, emotion, beat_out
/*
, minute, second, counter, latest, oldest, prv_min, dance //for debugging
, his_min3, his_min2, his_min1, his_min0
, his_sec3, his_sec2, his_sec1, his_sec0
, his_exp3, his_exp2, his_exp1, his_exp0
, state_sec, state_counter, state_min
*/
```

```
 );

	input clk, reset, beat;
	output [3:0] health_point; //0..15; to health_meter
	output state;    //to light
	output emotion;  //to light
	output beat_out;  //to light

	//for debugging
/*	output [2:0] minute, prv_min, his_min3, his_min2, his_min1, his_min0, state_min;
	output [5:0] second, his_sec3, his_sec2, his_sec1, his_sec0, state_sec;
	output [25:0] counter, state_counter;
	output [3:0] latest, oldest;
	output dance;
	output his_exp3, his_exp2, his_exp1, his_exp0;
*/

	////////////////////////////////////////////////////////////////////////////////
	//general timer

	parameter max_counter = /*26'd9;*/26'd64999999; //assuming clk 65mhz,
max_counter = 1s
	parameter max_second = /*6'd5;*/6'd59; //1min = 60s
	parameter max_minute = 3'd5; //1 day of virtual pet = 1 minute in real world

	reg [25:0] counter;
	reg [5:0] second;
	reg [2:0] minute;

	wire [25:0] next_counter = (counter==max_counter)? 0: counter+1;
	wire [5:0] next_second = (counter==max_counter)? (second==max_second)? 0:
second+1: second;
	wire [2:0] next_minute = (second==max_second && counter==max_counter)?
(minute==max_minute)? 0: minute+1 : minute;

	always @(posedge clk)
	begin
```

```
		if(reset)
			begin
				counter <= 0;
				second <= 0;
				minute <=0;
			end
		else
			begin
				counter <= next_counter;
				second <= next_second;
				minute <= next_minute;
			end
	end

	////////////////////////////////////////////////////////////////////////////////
	//history of last 10 beats

	parameter count_beat = /*4'd3;*/4'd9; //minimum 10 beat/minute to activate
state_dancing
	integer i;

	reg [3:0] latest; //index to record history of beats
	reg [5:0] his_sec[count_beat:0]; //second in which beat was recorded
	reg [2:0] his_min[count_beat:0]; //minute in which beat was recorded
	reg his_exp[count_beat:0]; //0 if beat recorded within 1 minute from now (ie not
expired)

	wire [3:0] oldest = (latest==count_beat)? 0: latest+1;
	wire [3:0] next_latest = beat? (latest==count_beat)? 0: latest+1: latest;
	wire [2:0] prv_min = (minute==3'd0)? max_minute: minute-1;

	always  @(posedge clk)
	begin

		if (reset)
			begin
				latest <= 0;
```

```
                        for (i=0; i<=count_beat; i=i+1)
                                begin
                                        his_min[i] <= 3'd0;
                                        his_sec[i] <= 6'd0;
                                        his_exp[i] <= 1;
                                end
                end
        else
                begin
                        latest <= next_latest;
                        his_min[latest] <= beat? minute: his_min[latest];
                        his_sec[latest] <= beat? second: his_sec[latest];
                        for (i=0; i<=count_beat; i=i+1)
                                if((beat==1) && (i==latest))
                                        his_exp[i] <=0;
                                else if(his_exp[i]==1)
                                        his_exp[i] <= 1;
                                else if((prv_min==max_minute) && (his_min[i]==0))
                                        his_exp[i] <= 0;
                                else
//note: just added 30 so that deavtivates in 30s
                                        his_exp[i] <= (his_min[i]<prv_min) ||
((his_min[i]==prv_min) && ((his_sec[i]+30)<second));
                end


        end


        ////////////////////////////////////////////////////////////////////////////////
//
        //state activation notes: state_dancing active if the time difference between
        //the latest beat and the last 10th beat is less than 1 minute.


        parameter state_idle = 1'b0;
        parameter state_dancing = 1'b1;


        reg state; //0:idle, 1:dancing
```

```
	reg old_state;
	wire dance = ((his_exp[latest]!=1) && (his_exp[oldest]!=1))? 1: 0;

	always @(posedge clk)
	begin
		old_state <= state;
		state <= reset? 0 : dance? state_dancing : state_idle;
	end

	///////////////////////////////////////////////////////////////////////////////
	//state timer: to time the length of current state

	reg [25:0] state_counter;
	reg [5:0] state_sec;
	reg [2:0] state_min;

	wire [25:0] next_state_counter = (state_counter==max_counter)? 0:
state_counter+1;
	wire [5:0] next_state_sec = (state_counter==max_counter)?
(state_sec==max_second)? 0: state_sec+1: state_sec;
	wire [2:0] next_state_min = (state_sec==max_second &&
state_counter==max_counter)? (state_min==max_minute)? 0: state_min+1 : state_min;

	always @(posedge clk)
	begin
		if(reset|| (old_state!=state))
			begin
				state_min <=0;
				state_sec <=0;
				state_counter <=0;
			end
		else
			begin
				state_counter <= next_state_counter;
				state_sec <= next_state_sec;
				state_min <= next_state_min;
			end
```

```
	end

	/////////////////////////////////////////////////////////////////////////////
	//health-point calculation
	//notes: once health_point becomes 0, it can never increase (ie the pet dies)

	parameter hincrement = 1'b1; //increment of health point
	parameter max_health_point = 4'd15;

	reg [3:0] health_point;

	wire new_state_min = (state_sec==max_second) && (state_counter==max_counter);
	wire [3:0] dance_health = (~new_state_min || health_point==max_health_point
||health_point==0)? health_point: (health_point + hincrement);
	wire [3:0] idle_health = (~new_state_min || health_point==0)? health_point:
(health_point - hincrement);

	always @(posedge clk)
	begin
		if(reset)
			health_point <= max_health_point;
		else case(state)
			state_dancing:		health_point <= dance_health;
			state_idle:		health_point <= idle_health;
		endcase
	end

	////////////////////////////////////////////////////////////////////////////
	//emotion calculation

	parameter happy = 1'b1;
	parameter sad = 1'b0;
	parameter happy_interval = 3'd2; //when it is idle, happy emotion only lasts for
5 minute

	reg emotion;
	//when idle, once it is sad, the only way to make it happy is to switch to
state_dancing
```

```
	//ie feed it with beats

	wire idle_emotion = (emotion==sad)? sad: ~(new_state_min &&
(state_min==happy_interval));


	always @(posedge clk)

	begin

		if(reset)

			emotion <= happy;

		else case(state)

			state_dancing:		emotion <= happy;

			state_idle:		emotion <= idle_emotion;

		endcase

	end



	assign beat_out =   beat;

	//debugging outputs


/*	assign his_exp3 = his_exp[3];

	assign his_exp2 = his_exp[2];

	assign his_exp1 = his_exp[1];

	assign his_exp0 = his_exp[0];


	assign his_min3 = his_min[3];

	assign his_min2 = his_min[2];

	assign his_min1 = his_min[1];

	assign his_min0 = his_min[0];


	assign his_sec3 = his_sec[3];

	assign his_sec2 = his_sec[2];

	assign his_sec1 = his_sec[1];

	assign his_sec0 = his_sec[0];

*/

 Endmodule
```

Health.v

```
module health_meter
(vclock,reset,hcount,vcount,hsync,vsync,blank,phsync,pvsync,pblank,pixel,
health_point);

       input vclock; // 65MHz clock
   input reset;           // 1 to initialize module
   input [10:0] hcount;    // horizontal index of current pixel (0..1023)
   input [9:0]      vcount; // vertical index of current pixel (0..767)
   input hsync;           // XVGA horizontal sync signal (active low)
   input vsync;           // XVGA vertical sync signal (active low)
   input blank;           // XVGA blanking (1 means output black pixel)

       output phsync;      // pong game's horizontal sync
   output pvsync;   // pong game's vertical sync
   output pblank;   // pong game's blanking

       output [23:0] pixel;


       input [3:0] health_point;


       parameter meter_height = 10'd64;

       parameter meter_x = 11'd720;

       parameter meter_y = 10'd76;


       parameter yellow = 24'hFECA10;

       parameter red = 24'hFF0000;

       parameter green = 24'h00FF00;


       wire [10:0] next_meter_width = health_point<<4;

       wire [23:0] next_meter_color =   (health_point>8)? green:

       (health_point<=8 && health_point>4)?    yellow:
                                                                       red;




       reg [10:0] meter_width;

       reg [23:0] meter_color;


       reg vsync_old;             //to store the value of vsync before the rising edge
of the clock

       wire new_frame = vsync_old & ~vsync; //falling edge of vsync */


       parameter ge = 1'b1;

       blob health(meter_x, meter_y, hcount, vcount, meter_color,
meter_width,meter_height,ge,pixel );
```

```
	always @(posedge vclock)
	begin

		vsync_old <= vsync;

		if(reset)
		begin
			meter_width <= 11'd240;
			meter_color <= green;
		end
		else if(new_frame)
		begin
			meter_width <= next_meter_width;
			meter_color <= next_meter_color;
		end	else begin end
	end

		assign phsync = hsync;
	assign pvsync = vsync;
	assign pblank = blank;

endmodule
```

blob.v

```
module blob(x,y,hcount,vcount,color,width,height,ge,pixel);

//a module to generate a rectangle with gradient
//gradient light at y-middle, dark at y-edges

	input ge;					//gradient-enable
	input [10:0] x,hcount;		//x: x-coordinate of top-left corner
	input [9:0] y,vcount;		//y: y-coordinate of top-left-corner
	input [10:0] width;
	input [9:0] height;
	input [23:0] color;		//color at y-middle
```

```
	output [23:0] pixel;

	parameter increment = 3;   //darken the gradient every 2^increment pixels from
y-middle

	reg [23:0] pixel;

	wire [9:0] mid_y = y + (height>>1); //y-middle point of the rectangle
	wire [9:0] delta_y = (vcount> mid_y)? vcount - mid_y: mid_y - vcount;

	wire [7:0] r = color[23:16];
	wire [7:0] g = color[15:8];
	wire [7:0] b = color[7:0];

	wire [7:0] newr = ge? r>>(delta_y>>increment) : r;
	wire [7:0] newg = ge? g>>(delta_y>>increment) : g;
	wire [7:0] newb = ge? b>>(delta_y>>increment) : b;

	wire [23:0] newcolor = {newr, newg, newb};

	always @ (x or y or hcount or vcount) begin
		if ((hcount >= x && hcount < (x+width)) && (vcount >= y && vcount <
(y+height)))
			pixel = newcolor;
		else pixel = 0;
	end

endmodule
```

dog.v

```
module dog
(vclock,reset,hcount,vcount,hsync,vsync,blank,phsync,pvsync,pblank,pixel,state,data_in
,addr);

	input vclock;// 65MHz clock
   input reset;            // 1 to initialize module
   input [10:0] hcount;    // horizontal index of current pixel (0..1023)
   input [9:0] vcount; // vertical index of current pixel (0..767)
   input hsync;            // XVGA horizontal sync signal (active low)
   input vsync;            // XVGA vertical sync signal (active low)
```

```
   input blank;            // XVGA blanking (1 means output black pixel)

       input state;      //determines whether the dog dances


       output phsync;      // output horizontal sync
   output pvsync;   // output vertical sync
   output pblank;   // output blanking
   output [23:0] pixel;    // output pixel



       //debugging

       output [23:0] data_in;

       output [16:0] addr;


       parameter max_addr = 117799; //max_addr in the image roms : 380x310 pixels

       //parameters of dog
       parameter dog_width = 11'd380;

       parameter dog_height = 10'd310;

       parameter dog_x = 11'd102; //top-left corner corrdinates

       parameter dog_y = 10'd229;


       parameter hold_frames = 6'd19; //each image holds for 20 frames before
transitting to next image

       parameter state_idle = 1'b0;

       parameter state_dancing = 1'b1;


       reg [16:0] addr;

       reg [1:0] count_rom;

       reg [5:0] count_frame;


       wire [4:0] encoded_data_left;

       wire [4:0] encoded_data_center;

       wire [4:0] encoded_data_right;


       wire [23:0] data_in_left;

       wire [23:0] data_in_center;

       wire [23:0] data_in_right;
```

```
       reg vsync_old;             //to store the value of vsync before the rising edge
of the clock


       assign phsync = hsync;
       assign pvsync = vsync;
       assign pblank = blank;


       //to determine which the address to read///////////////////////////////

       wire new_frame = vsync_old & ~vsync; //falling edge of vsync

       wire rom_within_region = ((hcount+3) >= dog_x && (hcount+3) < (dog_x+dog_width))
&& (vcount >= dog_y && vcount < (dog_y+dog_height));


       wire within_region = (hcount >= dog_x && hcount < (dog_x+dog_width)) && (vcount
>= dog_y && vcount < (dog_y+dog_height));

       wire [16:0] next_addr = (addr==0)? max_addr: addr - 1;



       always @(posedge vclock)

       begin

              if((hcount+3) == dog_x && vcount == dog_y)           //at top left corner
                     addr <= max_addr;
       //start getting data from rom

              else if(hcount && rom_within_region)                              //for
next pixel that is within the display region

                     addr <= next_addr;
       //next rom address

              else begin end
       end



      wire lrom_within_region = ((hcount+4) >= dog_x && (hcount+4) < (dog_x+dog_width))
&& (vcount >= dog_y && vcount < (dog_y+dog_height));


              reg [16:0] laddr;

    reg [10:0] count_mem;

    parameter max_count_mem = dog_width-1;

    wire [16:0] next_laddr = (laddr==max_count_mem)? max_addr-max_count_mem:
(count_mem==max_count_mem)? laddr - ((max_count_mem<<1) + 1): laddr + 1;

    wire [10:0] next_count_mem = (count_mem==max_count_mem)? 0: count_mem+1;

    always @(posedge vclock)

    begin

        if((hcount+4) == dog_x && vcount == dog_y)        //at top left corner

            begin     laddr <= max_addr- max_count_mem;
//start getting data from rom

                        count_mem <= 0; end
```

```
        else if(hcount && lrom_within_region)                     //for next pixel
that is within the display region

            begin        laddr <= next_laddr;                               //next
rom address

                        count_mem <= next_count_mem;    end

        else begin end

    end




       //to determine which bram to read////////////////////////////////////////

       always @(posedge vclock)

       begin

              vsync_old <= vsync;

              if(reset)

                     begin

                            count_rom <= 1;            //read the center-rom because the
initial state is always idle dog

                            count_frame <= 0;

                     end

              else if(new_frame)

              case(state)

                            state_idle:

                                   count_rom <= 1; //always show image from center_rom

                            state_dancing:

                                   begin

                                          count_frame <= (count_frame==hold_frames)? 0:
count_frame+1;

                                          count_rom <= (count_frame==hold_frames)?
count_rom+1: count_rom;

                                   end

                     endcase

       end



       wire [16:0] xaddr = (count_rom==0)? laddr: addr;

       //instances of read-only brams

       //they all share the same read address
```

```
//     left_rom boo_left(addr, vclock, encoded_data_left);
       //left_lut foo_left(encoded_data_left, vclock, data_in_left);
       right_rom boo_right(xaddr, vclock, encoded_data_right);
       right_lut foo_right(encoded_data_right, vclock, data_in_right);
       center_rom boo_center(addr, vclock, encoded_data_center);
       center_lut foo_center(encoded_data_center, vclock, data_in_center);



       assign data_in =    (count_rom==0)? data_in_right://data_in_left:
                                                (count_rom==1 || count_rom==3)?
data_in_center:
                                                data_in_right;



       assign pixel = within_region? data_in : 0;
endmodule
```

light.v

```
module
light_light(vclock,reset,hcount,vcount,hsync,vsync,blank,phsync,pvsync,pblank,pixel,em
otion,state,beat/*, count, done, done5,color_5, start*/);

   input vclock;    // 65MHz clock
   input reset;        // 1 to initialize module
   input [10:0] hcount;    // horizontal index of current pixel (0..1023)
   input [9:0] vcount; // vertical index of current pixel (0..767)
   input hsync;        // XVGA horizontal sync signal (active low)
   input vsync;        // XVGA vertical sync signal (active low)
   input blank;        // XVGA blanking (1 means output black pixel)

   output phsync;    // output horizontal sync
   output pvsync;    // output vertical sync
   output pblank;    // output blanking
   output [23:0] pixel;
   // output [2:0] count;
   // output done;
```

```
// output done5;
// output [23:0] color_5;
// output start;

 input emotion;
 input state;
 input beat;

 parameter yellow = 24'hFFFF00;
 parameter red = 24'hFF0000;
 parameter green = 24'h00FF00;
 parameter blue = 24'h0000FF;
 parameter black = 24'h000000;

 //parameters for circle_dual
 parameter radius = 11'd32;  //default width of all lights
 parameter light1_x = 11'd918;//948;
 parameter light1_y = 10'd404;
 parameter light2_x = 11'd693;//678;
 parameter light2_y = 10'd274;//248;
 parameter light3_x = 11'd693;//678;
 parameter light3_y = 10'd534;//560;
 parameter light4_x = 11'd843;//858;
 parameter light4_y = 10'd274;//248;
 parameter light5_x = 11'd618;//588;
 parameter light5_y = 10'd404;
 parameter light6_x = 11'd843;//858;
 parameter light6_y = 10'd534;//560;
 parameter light7_x = 11'd768;
 parameter light7_y = 10'd404;

 //emotion parameters
 parameter happy = 1'b1;
 parameter sad = 1'b0;

 //state parameters
```

```
    parameter state_idle = 1'b0;

    parameter state_dancing = 1'b1;


    //dual circles///////////////////////

    wire [23:0] color_1, color_2, color_3, color_4, color_5, color_6, color_7;

    wire [23:0] light1_pixel, light2_pixel, light3_pixel, light4_pixel, light5_pixel,
light6_pixel, light7_pixel;


     circle_dual light1(vclock, reset, light1_x,light1_y,hcount,vcount,radius,
color_1,light1_pixel);

      circle_dual light2(vclock, reset, light2_x,light2_y,hcount,vcount,radius,
color_2,light2_pixel);

      circle_dual light3(vclock, reset, light3_x,light3_y,hcount,vcount,radius,
color_3,light3_pixel);

      circle_dual light4(vclock, reset, light4_x,light4_y,hcount,vcount,radius,
color_4,light4_pixel);

      circle_dual light5(vclock, reset, light5_x,light5_y,hcount,vcount,radius,
color_5,light5_pixel);

      circle_dual light6(vclock, reset, light6_x,light6_y,hcount,vcount,radius,
color_6,light6_pixel);

      circle_dual light7(vclock, reset, light7_x,light7_y,hcount,vcount,radius,
color_7,light7_pixel);


    //flashers//////////////////////////


    parameter frame_rate = 6'd1;

    reg [23:0] light1_color;

    reg [23:0] light2_color;

    reg [23:0] light3_color;

    reg [23:0] light4_color;

    reg [23:0] light5_color;

    reg [23:0] light6_color;

    reg [23:0] light7_color;


    reg old_emotion;

    reg old_state;

    reg start;               //to start flashers

    reg beatreg;         //register beat input

    reg [3:0] increment;     //for every frame-rate the gradient change by every
increment unit
```

```
    reg increase;        //1:to increase (brighten) the gradient
    reg [2:0] count;     //number of flashes
    reg vsync_old;        //to store the value of vsync before the rising edge of the
clock

    wire new_frame = vsync_old & ~vsync; //falling edge of vsync
    wire done1, done2, done3, done4, done5, done6, done7;

    flasher_w flasher1(vclock, reset, vsync, light1_color, increase, increment,
frame_rate, color_1, start, done1);
    flasher_w flasher2(vclock, reset, vsync, light2_color, increase, increment,
frame_rate, color_2, start, done2);
    flasher_w flasher3(vclock, reset, vsync, light3_color, increase, increment,
frame_rate, color_3, start, done3);
    flasher_w flasher4(vclock, reset, vsync, light4_color, increase, increment,
frame_rate, color_4, start, done4);
    flasher_w flasher5(vclock, reset, vsync, light5_color, increase, increment,
frame_rate, color_5, start, done5);
    flasher_w flasher6(vclock, reset, vsync, light6_color, increase, increment,
frame_rate, color_6, start, done6);
    flasher_w flasher7(vclock, reset, vsync, light7_color, increase, increment,
frame_rate, color_7, start, done7);

    parameter max_count = 5;
    parameter increment_dancing = 4'd13;
    parameter increment_happy = 4'd13;
    parameter increment_sad = 4'd4;

    wire done = done1 & done2 & done3 & done4 & done5 & done6 & done7;
    wire case_reset = (old_emotion!=emotion || old_state!=state); //emotion or state
changed within the past frame period

    always @(posedge vclock)
    begin
        old_emotion <= new_frame? emotion: old_emotion;
        old_state <= new_frame? state : old_state;
        vsync_old <= vsync;
        beatreg <= beat? 1: new_frame? 0: beatreg; //beat within the past frame period
        if(reset || case_reset)
            begin
```

```
            count <= 0;
            start <= 1;
            light1_color <= black;
            light2_color <= black;
            light3_color <= black;
            light4_color <= black;
            light5_color <= black;
            light6_color <= black;
            light7_color <= black;
            increase <= 0;
            increment <= 4'd1;
        end

    else if(new_frame)
        case(state)
        state_idle:
            if(done && emotion==happy)
                begin
                    start <= 1;
                    count <= (count==max_count)? 0: count+1;
                    light1_color <= (count==0 || count==1)? red : black;
                    light2_color <= (count==0 || count==1)? green : black;
                    light3_color <= (count==0 || count==1)? blue : black;
                    light4_color <= (count==2 || count==3)? blue : black;
                    light5_color <= (count==2 || count==3)? red : black;
                    light6_color <= (count==2 || count==3)? green : black;
                    light7_color <= (count==4 || count==5)? yellow : black;
                    increase <= (count==0 || count==2 || count==4);
                    increment <= increment_happy;
                end
            else if(done && emotion==sad)
                begin
                    start <= 1;
                    count <= (count==max_count)? 0: count+1;
                    light1_color <= green;
                    light2_color <= green;
```

```
                    light3_color <= green;
                    light4_color <= green;
                    light5_color <= green;
                    light6_color <= green;
                    light7_color <= green;
                    increase <= (count==0 || count==2 || count==4);
                    increment <= increment_sad;
                end
            else begin end
        state_dancing:
            if(beatreg)
                begin
                    start <= 1;
                    count <= (count==max_count)? 0: count+1;
                    light1_color <= (count==0)? red : black;          //the
lights,except light7 take turn to flash
                    light2_color <= (count==1)? green : black;
                    light3_color <= (count==2)? blue : black;
                    light4_color <= (count==3)? blue : black;
                    light5_color <= (count==4)? red : black;
                    light6_color <= (count==5)? green : black;
                    light7_color <= yellow;
                    increase <= 0;
                    increment <= increment_dancing;
                end
            else begin end
        endcase


    else
        start <= 0;
end


assign pixel = light1_pixel + light2_pixel + light3_pixel + light4_pixel +
light5_pixel + light6_pixel + light7_pixel;
assign phsync = hsync;
assign pvsync = vsync;
assign pblank = blank;
```

```
endmodule
```

circle_dual.v

```
module circle_dual(clk, reset,

x,y,hcount,vcount,radius,color,pixel
/*xa,ya,hcounta,vcounta,radiusa,colora,pa,
xb,yb,hcountb,vcountb,radiusb,colorb,pb,              //,
wsa, wra, dxa, dya, sqxa, sqya,
wsb, wrb, dxb, dyb, sqxb, sqyb,
enable_a, old*/);

	input clk,reset;
	input [10:0] x,hcount;
	input [9:0] y,vcount;
	input [10:0] radius;  //maximum radius is the width of the screen
	input [23:0] color;
	output [23:0] pixel;

	wire [23:0] pixela, pixelb;
/*	output [23:0] pa, pb;
	output [10:0] xa, xb, hcounta, hcountb;
	output [9:0] ya, yb, vcounta, vcountb;
	output [10:0] radiusa, radiusb;
	output [23:0] colora, colorb;

	output enable_a;
	output [1:0] old;   */

	reg [10:0] xa, xb, hcounta, hcountb;
	reg [9:0] ya, yb, vcounta, vcountb;

	reg [10:0] radiusa, radiusb;
	reg [23:0] colora, colorb;
	reg [23:0] pixel;
```

```
	reg [1:0] old;

	always @(posedge clk) begin
	if(reset)
	old <= 0;
	else
	old <= (old==1)? 0: old+1 ;
	end

	wire ea = old==0;

/*	output wsa, wra;
	output [10:0] dxa, dya;
	output [21:0] sqxa, sqya;

	output wsb, wrb;
	output [10:0] dxb, dyb;
	output [21:0] sqxb, sqyb;
  */
	circle a(xa,ya,hcounta,vcounta,radiusa, colora, pixela/*, wsa, wra, dxa, dya,
sqxa, sqya*/);
	circle b(xb,yb,hcountb,vcountb,radiusb, colorb, pixelb/*, wsb, wrb, dxb, dyb,
sqxb, sqyb*/);

	always @(posedge clk) begin
	if(reset)
		begin
			xa     <= x;
			ya     <= y;
			hcounta <= hcount;
			vcounta <= vcount;
			radiusa <= radius;
			colora  <= color;
			pixel <= pixela;
		end
	else if(ea)
```

```
        begin
              xb     <= x;
              yb     <= y;
              hcountb <= hcount;
              vcountb <= vcount;
              radiusb <= radius;
              colorb  <= color;
              pixel <= pixelb;
        end
    else
        begin
              xa     <= x;
              ya     <= y;
              hcounta <= hcount;
              vcounta <= vcount;
              radiusa <= radius;
              colora  <= color;
              pixel <= pixela;
        end
    end


//      assign  pa = pixela;
//      assign pb = pixelb;
//      assign enable_a = ea;




        endmodule
```

circle.v

```
module circle(x,y,hcount,vcount,radius, color, pixel/*, ws, wr, dx, dy, a, b*/);
```

```
	input [10:0] x,hcount;
	input [9:0] y,vcount;

	input [10:0] radius;  //maximum radius is the width of the screen

	input [23:0] color;
	output [23:0] pixel;
	reg [23:0] pixel;

/*	output ws, wr;
	output [10:0] dx, dy;
	output [21:0] a, b;	*/

	wire [9:0] delta_y = (vcount>y)? vcount - y: y-vcount;
	wire [10:0] delta_x = (hcount>x)? hcount - x: x-hcount;

	//wire [1:0] short_delta_x = delta_x[1:0];
	//wire [1:0] short_delta_y = delta_y[1:0];

	wire within_square_x = delta_x <=radius;
	wire within_square_y = delta_y <= radius;
	wire within_square = within_square_x && within_square_y;
	wire within_radius = within_square? (delta_x*delta_x + delta_y*delta_y) <=
radius*radius : 0;
	//wire within_radius = within_square? (short_delta_x*short_delta_x +
short_delta_y*short_delta_y) <= radius*radius : 0;

	always @(x or y or hcount or vcount or radius or color)
	begin
	if(within_radius)
		pixel = color;
	else
		pixel =0;
	end

/*	assign ws = within_square;
	assign wr = within_radius;
```

```
	assign a = delta_x*delta_x;
	assign b= delta_y*delta_y;
	assign dx = delta_x;
	assign dy = delta_y;*/

endmodule
```

flasher.v

```
module
flasher_w(vclock,reset,vsync,rgbin,increase,increment,frame_rate,rgbout,start,done);

	input vclock, reset, vsync;
	input [23:0] rgbin;
	input increase;
	input [3:0] increment;
	input [5:0] frame_rate;
	input start;
	output done;
	output [23:0] rgbout;

	wire [7:0] rin = rgbin[23:16];
	wire [7:0] gin = rgbin[15:8];
	wire [7:0] bin = rgbin[7:0];

	wire [7:0] rout = rgbout[23:16];
	wire [7:0] gout = rgbout[15:8];
	wire [7:0] bout = rgbout[7:0];

	reg [5:0] count_frame;	//count till 59
	reg [23:0] rgbout;

	wire signed [8:0] next_decr = rout - increment;
	wire signed [8:0] next_decg = gout - increment;
	wire signed [8:0] next_decb = bout - increment;
```

```
	wire [7:0] decr = (count_frame==frame_rate)? (next_decr <=0)? 0 :
next_decr[7:0] : rout;

	wire [7:0] decg = (count_frame==frame_rate)? (next_decg <=0)? 0 :
next_decg[7:0] : gout;

	wire [7:0] decb = (count_frame==frame_rate)? (next_decb <=0)? 0 :
next_decb[7:0] : bout;


	wire signed [8:0] next_incr = rout + increment;

	wire signed [8:0] next_incg = gout + increment;

	wire signed [8:0] next_incb = bout + increment;


	wire [7:0] incr = (count_frame==frame_rate)? (rin==0)? 0: (next_incr <=0)? rin :
next_incr[7:0] : rout;

	wire [7:0] incg = (count_frame==frame_rate)? (gin==0)? 0: (next_incg <=0)? gin :
next_incg[7:0] : gout;

	wire [7:0] incb = (count_frame==frame_rate)? (bin==0)? 0: (next_incb <=0)? bin :
next_incb[7:0] : bout;


	wire [7:0] newr = increase? incr : decr;

	wire [7:0] newg = increase? incg : decg;

	wire [7:0] newb = increase? incb : decb;


	wire [23:0] newcolor = {newr, newg, newb};

	wire [5:0] next_count_frame = (count_frame==frame_rate)? 0: count_frame+1;


	reg vsync_old;		//to store the value of vsync before the rising edge
of the clock

	wire new_frame = vsync_old & ~vsync; //falling edge of vsync


	always @(posedge vclock)

	begin


		vsync_old <= vsync;


		if(reset || start)

		begin

			rgbout <= increase? 0 : rgbin ;

			count_frame <= 0;

		end
```

```
            else if(new_frame)
            begin
                  rgbout <= newcolor;
                  count_frame <= next_count_frame;
            end   else begin end
      end

      assign done = increase? (rout==rin &&  gout==gin && bout==bin) : (rout==0 &&
bout==0 && gout==0);

endmodule
```

video_graph.v

```
module
graph_video(ready,from_ac97_data,vclock,reset,hcount,vcount,hsync,vsync,blank,phsync,p
vsync,pblank,pixel
, spectrum
//,energy, busy, rfd,ffwd_inv_we,fstart,ffwd_inv,fscale_sch_we,fscale_sch, fxk_re,
fxk_im, fxn_index, fxk_index, oreal, oimag, height
);

    input ready, reset;
      input [7:0] from_ac97_data;

      input vclock;
    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)

    /*
    ///for video_sim
    output [10:0] hcount;    // horizontal index of current pixel (0..1023)
   output [9:0] vcount; // vertical index of current pixel (0..767)
   output hsync;        // XVGA horizontal sync signal (active low)
   output vsync;        // XVGA vertical sync signal (active low)
```

```
    */
    input blank;        // XVGA blanking (1 means output black pixel)

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

             wire [9:0] scale_sch = 10'b1010101011;

    reg vsync_old;        //to store the value of vsync before the rising edge of the
clock
    wire new_frame = vsync_old & ~vsync; //falling edge of vsync

    reg ready_old;
       reg rdy;
    wire dclk = ~ready_old & rdy; //rising edge of ready

    //wires for fft
    wire ce = dclk; //feed fft when there is new audio data
    //wire rfd;
    wire fwd_inv = 1'b1; //never does inverse fourier
    wire dv;
    wire done;
    //wire busy;
    wire edone;
   // wire [5:0] scale_sch = 6'b011010;
    wire [7:0] xn_im = 8'b0;
    wire signed [7:0] xn_re = from_ac97_data;
    wire signed [7:0] xk_im;
    wire signed [7:0] xk_re;
   wire [9:0] xn_index;      //fft input index
    wire [9:0] xk_index;     //fft output index

    //initialize fft
    //reset: nothing
```

```
 //1st ready: fwd_inv_we
 //2nd ready: scale_sch_we
 //3rd ready: start fft
 reg [1:0] initclk;
always @(posedge vclock)
  initclk <= reset ? 0 : (((initclk<3) & ce) ? initclk + 1 : initclk);
wire fwd_inv_we = (initclk==2'd1);
wire start = (initclk==2'd3);

// reload scale factors when changed as well as at init time
reg [7:0]   old_scale_sch;
always @(posedge vclock) old_scale_sch <= scale_sch;
assign scale_sch_we = initclk[0] | ~(scale_sch==old_scale_sch);

 always @(posedge vclock)
 begin
     vsync_old <= vsync;
     ready_old <= rdy;
            rdy <= ready;
 end

 assign phsync = hsync;
   assign pvsync = vsync;
   assign pblank = blank;

/*    output [15:0] energy;
 output busy;
 output rfd;

 output ffwd_inv_we;
 output fstart;
 output ffwd_inv;
 output fscale_sch_we;
 output [5:0] fscale_sch;

 output [7:0] fxk_re;
```

```
    output [7:0] fxk_im;

    output [4:0] fxn_index;

    output [4:0] fxk_index;


    assign ffwd_inv_we = fwd_inv_we;

    assign fstart = start;

    assign ffwd_inv = fwd_inv;

    assign fscale_sch_we = scale_sch_we;

    assign fscale_sch = scale_sch;


    assign fxk_re = xk_re;

    assign fxk_im = xk_im;

    assign fxn_index = xn_index;

    assign fxk_index = xk_index;


    output [7:0] oreal;

    output [7:0] oimag;


    output [8:0] height;

  */


    /*

    video_sim xvgasim(vclock, reset, hcount, vcount, hsync, vsync);

    defparam xvgasim.max_hcount = 40; //1343;

    defparam xvgasim.max_vcount = 40; //805;

    defparam xvgasim.max_frame_x = 35; //1024;

    defparam xvgasim.max_frame_y = 35;  //768;

    */

        wire overflow;

       fft_1024
fft(xn_re,xn_im,start,fwd_inv,fwd_inv_we,scale_sch,scale_sch_we,ce,vclock,xk_re,xk_im,
xn_index,xk_index,rfd,busy,dv,edone,done,overflow);


  /*

       fft fre1 (

       xn_re,

       xn_im,
```

```
        start,
        fwd_inv,
        fwd_inv_we,
        scale_sch,
        scale_sch_we,
        ce,
        vclock,
        xk_re,
        xk_im,
        xn_index,
        xk_index,
        rfd,
        busy,
        dv,
        edone,
        done); */
   // fft
fre1(ce,fwd_inv_we,rfd,start,fwd_inv,dv,scale_sch_we,done,vclock,busy,edone,scale_sch,
xn_re,xk_im,xn_index,xk_re,xn_im,xk_index);

        wire [15:0] xk_re_sq;
        wire [15:0] xk_im_sq;

        output [7:0] spectrum;

        multiplier re(vclock, xk_re,xk_re,xk_re_sq);
        multiplier im(vclock, xk_im,xk_im,xk_im_sq);

    wire [16:0] energy = xk_re_sq + xk_im_sq;
    wire [7:0] data1 = energy[15:0];
    wire [7:0] data2 = 8'b0;
    wire [10:0] cx = 11'd0;
    wire [9:0] cy = 10'd0;

         assign spectrum = data1;
```

```
    /*
   //first test
   graph2_buf graph(reset,vclock,hcount,vcount,pixel,data1,data2,dclk,done,cx,cy);

   defparam graph.NX = 31;    // number of data pixels (horiz size)
   defparam graph.GPIX_RGB1 = 24'hFFFFFF;    // pixel RGB for graph data1, red
   defparam graph.GPIX_RGB2 = 24'h000000;    // pixel RGB for graph data1, green

   defparam graph.screen_width = 1023;
   defparam graph.screen_height = 767;
   defparam graph.graph_height = 513; //512 pixels for data, graph(with border)
(0..513)
   defparam graph.graph_width = 513; //512 pixels for data, graph(with border)
(0..513)

  */
   wire [23:0] pixel;
   //second test
   graph2_bar graph2(reset,vclock,hcount,vcount,pixel,data1,data2,dclk,done,cx,cy);

   defparam graph2.NX = 1024;//31;    // number of data pixels (horiz size)
   defparam graph2.GPIX_RGB1 = 24'hFFFFFF;    // pixel RGB for graph data1, red
   defparam graph2.GPIX_RGB2 = 24'h000000;    // pixel RGB for graph data1, green
   defparam graph2.increment = 0;    // pixel RGB for graph data1, green



   defparam graph2.screen_width = 1023;
   defparam graph2.screen_height = 767;
   defparam graph2.graph_height = 513;//257; //512 pixels for data, graph(with border)
(0..513)
   defparam graph2.graph_width = 216;//513; //512 pixels for data, graph(with border)
(0..513)



endmodule
```