High Striker
6.111 Final Project Report

Jennifer Chan and Mike Stunes

December 10, 2010

Abstract

The video and audio hardware of the 6.111 lab kit, along with an external accelerometer, is used to create a digital reimplementation of the classic carnival game “High Striker”. Digital logic on the FPGA is used to calculate the motion of a weight sliding up a pole, based on force input from the accelerometer, and this information is used to draw an image of the weight and pole on a screen. In addition, audio logic produces sounds of the bell ringing and of the weight sliding along the pole.

Contents

List of Figures

1 Overview

This project is a digital reimplementation of the carnival game known as “High Striker”. In this game, the player swings a hammer at a target. Hitting the target causes a weight to slide up a pole. There is a bell at the top of the pole; if the weight hits the bell, the player “wins”. This was recreated by attaching an accelerometer to a hammer, which is used to strike a target. There is an infrared transmitter and receiver placed near the target; these are used to detect when the hammer approaches the target. The information from the accelerometer is converted to an initial velocity; a physics engine uses this to calculate the position of the weight as a function of time. The position of the weight is sent to the graphics module, which draws the weight, pole, bell, and background image to the screen. A sound module also plays the sound of the weight sliding up and down the pole, as well as the sound of the bell ringing.

See Figure 1 for an overview block diagram.

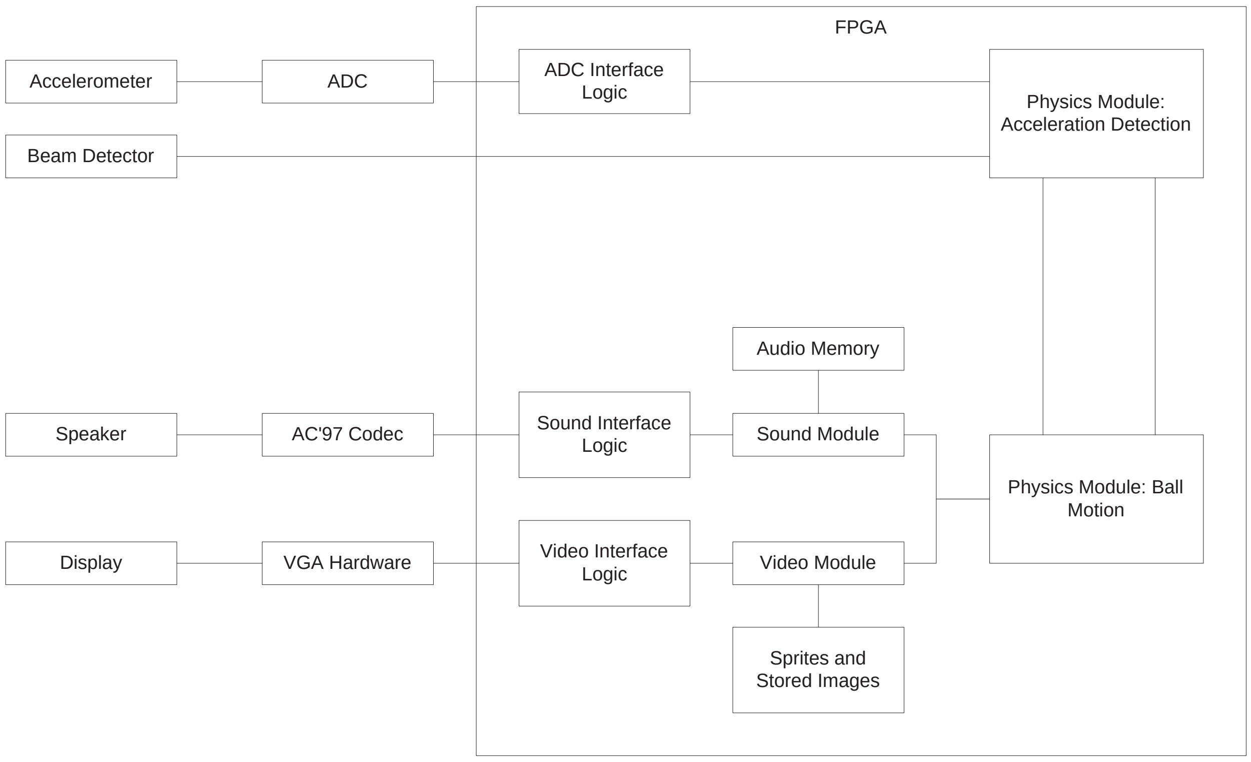

Figure 1: System block diagram.

2 Description

2.1 Beam detector (Jennifer)

We believed initially that we could simply send a 40 kHz signal from the infrared transmitter to the receiver: seeing the signal at the receiver would indicate “on”, and the absence of a signal would indicate “off”. However, we discovered that the receiver outputs a pulse (a low signal) when it receives a 40 kHz signal. As a result, the output was changed to output a signal every 600 μs. The 600 μs signal contains a 40 kHz square wave within it. This module is driven with a 27 MHz master clock, and uses two counters. One counter generates the 40 kHz square wave, and the other generates the 600 μs square wave. Each counter is associated with a register whose state is inverted whenever that counter times out; the counter is then reset. The transmitter emits an infrared pulse whenever both registers are high.

The receiver then generates a 600 μs wave whenever it detects the beam. When the beam is blocked, the receiver merely emits a continuous high signal. The receiver's logic module maintains a counter driven by a 75 μs clock; the counter is incremented whenever a high signal is detected from the receiver, and reset whenever a low signal is detected. If the count reaches 10, we know that the beam is blocked and then assert the *game_enable* signal, indicating to the force module that the game has begun.

The raw signal from the transmitter was quite strong and had a tendency to bounce quite a bit. We used straws attached to both the transmitter and receiver to focus the infrared beam; we also attenuated the signal using a resistor in series with the transmitter.

2.2 Accelerometer (Mike)

The motion of the hammer is detected using a Dimension Engineering DE-ACCM3D accelerometer module, based on the Analog Devices ADXL330 accelerometer. The module can detect acceleration along three axes; however, only one was used for this project. The accelerometer connects directly to the ADC logic module described in Section 2.3.

2.3 ADC Logic (Mike)

The analog input from the accelerometer is converted to a digital signal with an Analog Devices AD7824 four-channel ADC. Because only one axis of the accelerometer is used, only one channel of the ADC is used as well. This ADC has two reference inputs that can be used to set its effective input range; we used a resistor voltage divider with an op-amp buffer to set the input range of the ADC to the actual output range of the accelerometer (approx. 0-3 volts).

A three-state FSM is used to generate the control signals necessary to drive the ADC. This FSM repeatedly signals the ADC to begin sampling; when a sample is ready, the FSM presents the sample on its output to the rest of the project's modules and asserts a ready signal. This process then repeats indefinitely.

2.4 Force Computation Logic (Mike)

The force computation logic uses a two-state FSM to calculate the initial velocity of the weight, given the digital accelerometer signal from the ADC logic. When the *game_enable* signal from the beam detector is asserted, the FSM transitions from the "idle" state to the "counting" state. In the counting state, the FSM will count out 175,600 samples from the ADC, incorporating them into an accumulator register, and then transition back to the idle state. When transitioning back to idle, the force logic asserts a ready signal that informs the physics logic (see Section 2.6) that the velocity has been computed. For each sample, the logic adds to the accumulator the absolute value of the deviation of that sample from the value of the resting point of the accelerometer.

The accumulator is fed into a combinational logic block that scales and translates its value into an appropriate value for the physics module. In addition, this module allows the game to be "rigged" just like the real carnival game, by using the switches on the lab kit to input a value that will be subtracted from the computed initial velocity before it is passed to the physics module.

2.5 Graphics (Jennifer)

The graphics module uses a four-state FSM and takes as input the position of the weight from the physics module. It utilizes four sprites: the weight, the bell, the background image, and the high-score display.

2.5.1 Inputs

The graphics module takes, as inputs, the initial velocity and force ready signal from the force module. The force ready signal notifies the graphics module when the intial velocity has been computed; the velocity specifies the initial velocity of the weight as it leaves the ground.

2.5.2 Outputs

The graphics module outputs six parameters. Four are related to the graphics: phsync, pvsync, pblank, and pixel. Initially, there was a problem that resulted in the first row of pixels instead showing the last row of pixels, so phsync, pvsync, and pblank were delayed by one clock cycle. The graphics module's reset signal is also delayed by one clock cycle for the same reason. As long as reset is not asserted, the internal register "show2" is held high, and the pixels produced by the sprites will be drawn. The pixel output is a 24-bit value, with eight bits for each of red, green, and blue. The z-axis logic for the sprites is handled by simply testing if the sprites' values at a particular position are zero. If so, the background is drawn; otherwise, the sum of the pixel values from all of the other sprites is drawn: this technique works because all of the foreground sprites are always non-overlapping.

Two additional outputs convey information computed by the graphics module: bell_ringing and relative_ball_height. bell_ringing notifies the rest of the system if the weight collided with the bell. This signal is asserted when the weight collides with the bell, and deasserted when the weight reaches the ground again. The relative_ball_height ranges from 0 to 425 (in arbitrary units), and is simply the weight's absolute position on the screen minus its starting position at the bottom of the pole.

2.5.3 Graphics FSM

The graphics module FSM uses four states: idle, calculating, start-game, and moving. When in the idle state, the position of the weight is the bottom of the pole. The force_ready signal from the force module is asserted while the game is inactive and becomes deasserted when the player strikes the target. When this signal becomes deasserted, the graphics FSM transitions to the calculating state, and the position of the weight remains at the bottom of the pole. When force_ready becomes asserted again, the FSM knows that the initial velocity is ready and transitions to the start-game state. In the start-game state, the FSM initializes its velocity to the initial velocity, and transitions to the moving state. In the moving state, it activates the physics logic that calculates the actual movement of the weight (see Section 2.6). Once the weight reaches the ground, the FSM transitions back to the idle state.

2.5.4 Background Image and Screen Resolution

The background image is 800x600, with four-bit color. There are four COE files associated with the background image. The main image COE file consists of 480,000 four-bit values (ranging from 0 to 16). These values refer to color-maped red, green, and blue values. This color map is kept in the three other COE files, each of which contains sixteen 8-bit values. The picture of the carnival character is taken from `http://media.wii.ign.com/media/908/908180/imgs_1.html` and is originally from Wii Carnival Games.

In addition, because the screen resolution is 800x600, some changes needed to be made from the video code from Lab 5. The graphics modules are clocked at 40 MHz, in order to produce a 60 Hz refresh rate. In order to properly display each pixel, a pipeline delay had to be added. After some experimentation, we determined that we needed to delay the pixel output by eight clock cycles.

2.5.5 Weight and Bell

The image of the weight is 50x50 pixels and the image of the bell is 80x80 pixels. Each image is encoded with four-bit color. Each image has its own color maps. Because the generated images are circular and not square, the corresponding pixels are shown if

$$(x - x_c)^2 + (y - y_c)^2 < r$$

where (x_c, y_c) is the coordinates of the center of the image, and r is its radius. Other ideas for producing the cicrular shapes would have included pixel color testing and alpha blending, but the circle test proved effective. Because the pole is part of the background image, the weight and bell had to be aligned with the background image appropriately. The bell oscillates side-to-side if the weight collides with it. While the bell_ringing signal is asserted, the bell's x position alternates moving two pixels to the right and two pixels to the left of its center position. When bell_ringing becomes deasserted, the bell returnes to its original position.

2.5.6 Score Display

The score is displayed in the upper-left corner of the screen. It uses the VGA character module to display the characters of the word "SCORE" and the numeric score. The score is a discretization of the relative_ball_height signal. Thirteen if statements test the range of relative_ball_height and output a four-character string ranging from 0 to 1200, incrementing in steps of 100.

2.6 Physics (Jennifer)

The physics logic is triggered when the graphics module enters the "moving" state (see Section 2.5). It takes as input the initial velocity of the weight, from the force module. Based on calculations and experimentation, the effective range of the input (in arbitrary units) is from 0 (no movement) to 70 (the weight hitting the bell fairly quickly). The physics logic makes the weight act as if it were obeying the laws of physics, decelerating by gravity as it travels up the pole and accelerating as it goes back down. The initial design used a timestep module and a collision module, with the motion of the weight modeled as

$$x = v_0 t + \frac{1}{2} a t^2$$

However, initial testing and implementation revealed that a different approach would be needed, as it was difficult to estimate what the timestep should be without making the system unstable.

The current design uses an Euler integration to determine the position of the weight. Each vclock (from the graphics module) is considered a change in $t = 1$. At each vclock, $v = v + \frac{1}{2}a$ and $x = x + v$, where $a = 10$ or $a = -10$ depending on the direction in which the weight is traveling. As the time continues to increment, the velocity decreases until it reaches zero. In our implmentation, a higher position on the screen corresponds to a smaller x, therefore $x = x - v$. To detect a collision with the bell, the logic merely checks if x is greater than the position of the bell. If so, the logic produces an adjusted x which is equal to the position of the bell, and asserts a *bell_ringing* signal. If the weight does not collide with the bell, it keeps rising until the velocity becomes zero. Once the velocity is zero or the weight collides with the bell, the weight starts traveling down the pole. If the weight is falling, then $v = v + 5$ (i.e., it accelerates as it travels down the pole) and $x = x + v$.

This implementation produced some aliasing because the weight was moving too fast. Therefore, we needed to slow down the timestep, such that the weight changed position less frequently. A counter is used to count for n timesteps before recalculating the position of the weight. Experimentation revealed that $n = 4$ was optimal. Therefore, the position of the weight is calculated every four vclock signals.

2.7 Audio (Mike)

2.7.1 Bell

The sound of the bell ringing is implemented with a COE-formatted reencoding of a WAV audio clip, sampled at 12 kHz, stored in the FPGA's BRAM. Playback of the audio clip is handled by a two-state FSM. The FSM is initially in the “idle” state, which simply outputs nothing to the AC’97 codec. When a rising edge is detected on the *bell_ringing* signal from the physics module (see Section 2.6), the FSM transitions to the “playing” state. When in the playing state, a memory address register is initialized to zero. Every four AC’97 ready cycles (the AC’97 codec operates at 48 kHz, so every fourth cycle would produce a 12 kHz signal), the memory address is incremented, and the sample at that address is sent to the AC’97 codec.

The sound of the bell was acquired from `http://soundbible.com/56-Boxing-Bell-Start-Round.html`.

2.7.2 Sound of sliding weight

The sound of the weight sliding up and down the pole is implemented by generating a square wave at a frequency determined by the height of the weight, and passing the generated waveform through the AC’97 codec. Twenty static frequencies are defined in the Verilog code, and an FSM chooses which frequency is to be played at a particular time based on the height of the weight. The same FSM merely switches the state of the output to the AC’97 codec from 0 to 255 at the determined frequency.

3 Testing

3.1 Beam Detector

Testing of the beam detector was performed by simply connecting its output to an indicator on the hex display, showing whether the beam was blocked. Initial testing required the use of the oscilloscope to verify that the correct signals were being produced. Also, initial testing with the oscilloscope revealed that the receiver only emits a pulse when receiving a 40 kHz signal.

3.2 Force and Accelerometer

Characterizing the output of the accelerometer was done by observing its output on the oscilloscope while striking it against a target. Testing the output of the force module was performed using the logic analyzer, triggered on the signal indicating that the velocity calculation is complete. The accelerometer was struck on the target with different amounts of force and the final output of the force module was observed.

3.3 Graphics and Physics

The physics module was initially implemented with two simple solid color sprites, built on top of the pong game video logic from Lab 5. Initial testing quickly revealed that the method for calculating the motion of the weight needed to be changed, as guessing a timestamp module seemed to be tricky. After changing the physics module to use an Euler integration, the modules were set up to make the weight bounce up and down repeatedly with the same initial velocity, and the buttons on the lab kit could be used to change the initial velocity. However, it was quickly realized that using the buttons was not an accurate way to set the velocity, so the modules were changed to use the lab kit’s switches instead.

Once the physics modules were working, the ROMs were then loaded with the graphics sprites. A separate ISE project was created to test the drawing of the weight and bell from the ROM, to make sure that the ROMs were being loaded properly. Because of the size of the background image and the resulting synthesis time, we often tested the project by removing the background image. The ROMs were then integrated with the physics module, and a button on the lab kit was temporarily rigged to trigger the start of the game.

3.4 Audio

Testing of the sound of the bell was done by using a button on the lab kit as a trigger to begin playing the audio clip.

4 Challenges

One challenge for our design was ensuring that everything would fit within the constraints of our FPGA. In its final form, our design uses 143 of the 144 available BRAMs. This high utilization is caused mainly by the image sprites and background image for the graphics module: the weight and bell sprites are each 50x50 with 4-bit color, and the background is 800x600 with 4-bit color. This high BRAM use also led to long synthesis times (30-40 minutes), which impacted the process of debugging and modifying our design in the final stages.

An additional challenge arose from our use of multiple clock domains: the graphics modules ran on a 40 MHz clock, but most of the other logic (due to the needs of modules we used from previous labs) ran on a 27 MHz clock. This required us to make sure that none of our inter-module signals would get lost in the boundaries between clock domains.

Finally, our graphics modules were somewhat finicky, and did not always compile correctly. We often needed to experiment with different delays as well.

5 Conclusion

This project uses a series of interconnected FSMs, together with the graphics and audio hardware of the 6.111 lab kit and an external accelerometer, to produce a digital reimplementation of the classic carnival game known as High Striker. An infrared beam detector is used to determine when a hammer strikes a target; a series of digital logic modules then converts the force of the hammer, measured by the accelerometer, to the motion of the weight sliding up the pole. The image of the weight and pole is displayed on the screen; sound effects logic also generates the sound of the bell ringing and the weight sliding up and down.

A Verilog Code for AC97 Interface

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:32:36 11/29/2010
// Design Name:
// Module Name:    ac97
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////



module lab4audio (
    input wire clock_27mhz,
    input wire reset,
    input wire [4:0] volume,
    output wire [7:0] audio_in_data,
    input wire [7:0] audio_out_data,
    output wire ready,
    output reg audio_reset_b,       // ac97 interface signals
    output wire ac97_sdata_out,
    input wire ac97_sdata_in,
    output wire ac97_synch,
    input wire ac97_bit_clock
);

    wire [7:0] command_address;
    wire [15:0] command_data;
    wire command_valid;
    wire [19:0] left_in_data, right_in_data;
    wire [19:0] left_out_data, right_out_data;

    // wait a little before enabling the AC97 codec
    reg [9:0] reset_count;
    always @(posedge clock_27mhz) begin
        if (reset) begin
            audio_reset_b = 1'b0;
            reset_count = 0;
        end
        else if (reset_count == 1023)
            audio_reset_b = 1'b1;
        else
            reset_count = reset_count+1;
    end

    wire ac97_ready;
    ac97 ac97(.ready(ac97_ready),
        .command_address(command_address),
        .command_data(command_data),
        .command_valid(command_valid),
        .left_data(left_out_data), .left_valid(1'b1),
        .right_data(right_out_data), .right_valid(1'b1),
        .left_in_data(left_in_data), .right_in_data(right_in_data),
        .ac97_sdata_out(ac97_sdata_out),
        .ac97_sdata_in(ac97_sdata_in),
        .ac97_synch(ac97_synch),
        .ac97_bit_clock(ac97_bit_clock));

    // ready: one cycle pulse synchronous with clock_27mhz
    reg [2:0] ready_sync;
    always @ (posedge clock_27mhz) ready_sync <= {ready_sync[1:0], ac97_ready};
    assign ready = ready_sync[1] & ~ready_sync[2];

    reg [7:0] out_data;
    always @ (posedge clock_27mhz)
        if (ready) out_data <= audio_out_data;
    assign audio_in_data = left_in_data[19:12];
    assign left_out_data = {out_data, 12'b000000000000};
    assign right_out_data = left_out_data;

    // generate repeating sequence of read/writes to AC97 registers
    ac97commands cmds(.clock(clock_27mhz), .ready(ready),
        .command_address(command_address),
        .command_data(command_data),
        .command_valid(command_valid),
        .volume(volume),
        .source(3'b000));     // mic
endmodule

// assemble/disassemble AC97 serial frames
module ac97 (
    output reg ready,
    input wire [7:0] command_address,
    input wire [15:0] command_data,
    input wire command_valid,
    input wire [19:0] left_data,
    input wire left_valid,
    input wire [19:0] right_data,
    input wire right_valid,
    output reg [19:0] left_in_data, right_in_data,
    output reg ac97_sdata_out,
    input wire ac97_sdata_in,
    output reg ac97_synch,
    input wire ac97_bit_clock
);
    reg [7:0] bit_count;

    reg [19:0] l_cmd_addr;
    reg [19:0] l_cmd_data;
    reg [19:0] l_left_data, l_right_data;
    reg l_cmd_v, l_left_v, l_right_v;

    initial begin
        ready <= 1'b0;
        // synthesis attribute init of ready is "0";
        ac97_sdata_out <= 1'b0;
        // synthesis attribute init of ac97_sdata_out is "0";
        ac97_synch <= 1'b0;
        // synthesis attribute init of ac97_synch is "0";

        bit_count <= 8'h00;
        // synthesis attribute init of bit_count is "0000";
        l_cmd_v <= 1'b0;
        // synthesis attribute init of l_cmd_v is "0";
        l_left_v <= 1'b0;
        // synthesis attribute init of l_left_v is "0";
        l_right_v <= 1'b0;
        // synthesis attribute init of l_right_v is "0";

        left_in_data <= 20'h00000;
        // synthesis attribute init of left_in_data is "00000";
        right_in_data <= 20'h00000;
        // synthesis attribute init of right_in_data is "00000";
    end

    always @(posedge ac97_bit_clock) begin
        // Generate the sync signal
        if (bit_count == 255)
            ac97_synch <= 1'b1;
        if (bit_count == 15)
            ac97_synch <= 1'b0;

        // Generate the ready signal
        if (bit_count == 128)
            ready <= 1'b1;
        if (bit_count == 2)
            ready <= 1'b0;

        // Latch user data at the end of each frame. This ensures that the
        // first frame after reset will be empty.
        if (bit_count == 255) begin
            l_cmd_addr <= {command_address, 12'h000};
            l_cmd_data <= {command_data, 4'h0};
            l_cmd_v <= command_valid;
            l_left_data <= left_data;
            l_left_v <= left_valid;
            l_right_data <= right_data;
            l_right_v <= right_valid;
        end

        if ((bit_count >= 0) && (bit_count <= 15))
            // Slot 0: Tags
            case (bit_count[3:0])
                4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
                4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
                4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
                4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
                4'h4: ac97_sdata_out <= l_right_v; // Right data valid
                default: ac97_sdata_out <= 1'b0;
            endcase
        else if ((bit_count >= 16) && (bit_count <= 35))
            // Slot 1: Command address (8-bits, left justified)
            ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;
        else if ((bit_count >= 36) && (bit_count <= 55))
            // Slot 2: Command data (16-bits, left justified)
            ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;
        else if ((bit_count >= 56) && (bit_count <= 75)) begin
            // Slot 3: Left channel
            ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
            l_left_data <= { l_left_data[18:0], l_left_data[19] };
        end
        else if ((bit_count >= 76) && (bit_count <= 95))
            // Slot 4: Right channel
            ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
        else
            ac97_sdata_out <= 1'b0;

        bit_count <= bit_count+1;
    end // always @ (posedge ac97_bit_clock)

    always @(negedge ac97_bit_clock) begin
        if ((bit_count >= 57) && (bit_count <= 76))
```

```
            // Slot 3: Left channel
            left_in_data <= { left_in_data[18:0], ac97_sdata_in };
        else if ((bit_count >= 77) && (bit_count <= 96))
            // Slot 4: Right channel
            right_in_data <= { right_in_data[18:0], ac97_sdata_in };
    end
endmodule

// issue initialization commands to AC97
module ac97commands (
    input wire clock,
    input wire ready,
    output wire [7:0] command_address,
    output wire [15:0] command_data,
    output reg command_valid,
    input wire [4:0] volume,
    input wire [2:0] source
);
    reg [23:0] command;

    reg [3:0] state;
    initial begin
        command <= 4'h0;
        // synthesis attribute init of command is "0";
        command_valid <= 1'b0;
        // synthesis attribute init of command_valid is "0";
        state <= 16'h0000;
        // synthesis attribute init of state is "0000";
    end

    assign command_address = command[23:16];
    assign command_data = command[15:0];

    wire [4:0] vol;
    assign vol = 31-volume;  // convert to attenuation

    always @(posedge clock) begin
        if (ready) state <= state+1;

            case (state)
                4'h0: // Read ID
                    begin
                        command <= 24'h80_0000;
                        command_valid <= 1'b1;
                    end
                4'h1: // Read ID
                    command <= 24'h80_0000;
                4'h3: // headphone volume
                    command <= { 8'h04, 3'b000, vol, 3'b000, vol };
                4'h5: // PCM volume
                    command <= 24'h18_0808;
                4'h6: // Record source select
                    command <= { 8'h1A, 5'b00000, source, 5'b00000, source};
                4'h7: // Record gain = max
                    command <= 24'h1C_0F0F;
                4'h9: // set +20db mic gain
                    command <= 24'h0E_8048;
                4'hA: // Set beep volume
                    command <= 24'h0A_0000;
                4'hB: // PCM out bypass mix1
                    command <= 24'h20_8000;
                default:
                    command <= 24'h80_0000;
            endcase // case(state)
    end // always @ (posedge clock)
```

```
endmodule // ac97commands
```

B Verilog Code for ADC Interface

```
`timescale 1ns / 1ps
`default_nettype none

module adc(
    input [7:0] adc_data,
    input adc_int,
    input clk,
    //input reset,

    output reg [7:0] data,
    output reg ready,
    output reg adc_rd
);

    reg [2:0] cycle_count;

    always @(posedge clk) begin
        cycle_count <= 0;

        if (!adc_rd && !adc_int) begin
            //// ADC just finished conversion: read data and start over
            //data <= adc_data;
            //adc_rd <= 1;
            //ready <= 1;

            // ADC just finished conversion: give it a few cycles to settle
            //case (cycle_count)
            if (cycle_count < 4) begin
                data <= data;
                adc_rd <= 0;
                ready <= 0;
                cycle_count <= cycle_count + 1;
            end
            else begin
                data <= adc_data;
                adc_rd <= 1;
                ready <= 1;
                cycle_count <= 0;
            end
        end
        else if (!adc_rd && adc_int) begin
            // ADC is in the middle of conversion
            data <= data;
            adc_rd <= 0;
            ready <= 0;
        end
        else if (adc_rd) begin
            // Just finished conversion and need to start it again
            data <= data;
            adc_rd <= 0;
            ready <= 0;
        end
    end

endmodule
```

C Verilog Code for Sprites

```
//////////////////////////////////////////////////////////////////////
//
// ball sprite from generated image
//
//////////////////////////////////////////////////////////////////////
module ball_sprite
    #(parameter WIDTH = 50, HEIGHT = 50)  // default picture width // default picture height
    (
    input pixel_clk,
    input show2,
    input [10:0] x, hcount,
    input [9:0] y,vcount,
    output reg [23:0] pixel
);

    reg [18:0] deltax;
    reg [18:0] deltay;
    reg [18:0] radiussquared = 625;
    parameter RADIUS = 25; // can change it to 24/23 to remove weird edges/one column pixel delay

    wire  [11:0] image_addr;  // num of bits for 50x50 ROM
    wire [3:0] image_bits;  // only 16 colors available- each color maps to some (rgb) combination
    wire [7:0] red_mapped,green_mapped, blue_mapped;   // red, green, blue each expressed as a value from 0-255


    // note the one clock cycle delay in pixel!
    always @ (posedge pixel_clk) begin
        deltax <= (hcount > (x+RADIUS)) ? (hcount-(x+RADIUS)) : ((x+RADIUS)-hcount);
        deltay <= (vcount > (y+RADIUS)) ? (vcount-(y+RADIUS)) : ((y+RADIUS)-vcount);

        if(show2 && (deltax*deltax+deltay*deltay <= radiussquared))
            pixel <= {red_mapped, green_mapped, blue_mapped};
        else pixel <= 0;
    end

   // image_bits, for each bit, has a value from 0-16 (representing the color of the pixel)- there are 16 possible colors
    ball_rom myBallROM(.addr(image_addr),.clk(pixel_clk), .dout(image_bits));

    // use color map to create 8bits R, 8bits G, 8 bits B;
    // the roms are created by the system
    ball_rcm_rom ballredRom(.addr(image_bits), .clk(pixel_clk), .dout(red_mapped));
    ball_gcm_rom ballgreenRom(.addr(image_bits),.clk(pixel_clk),.dout(green_mapped));
    ball_bcm_rom ballblueRom(.addr(image_bits),.clk(pixel_clk), .dout(blue_mapped));

    // calculate rom address and read the location
    assign image_addr = (hcount-x) + (vcount-y) * WIDTH;

endmodule

//////////////////////////////////////////////////////////////////////
//
// bell sprite from generated image
//
//////////////////////////////////////////////////////////////////////
module bell_sprite
    #(parameter WIDTH = 80, HEIGHT = 80)  // default picture width // default picture height
    (input pixel_clk, input show2, input [10:0] x,hcount, input [9:0] y,vcount, output  reg [23:0] pixel);

    reg [18:0] deltax;
    reg [18:0] deltay;
    reg [18:0] radiussquared = 1600;
    parameter RADIUS = 40;      // can change it to 39/38 to remove weird edges/one column pixel delay
    wire  [12:0] image_addr;  // num of bits for 50x50 ROM
    wire [3:0] image_bits;  // only 16 colors available- each color maps to some (rgb) combination
    wire [7:0] red_mapped,green_mapped, blue_mapped;   // red, green, blue each expressed as a value from 0-255


    // note the one clock cycle delay in pixel!
    always @ (posedge pixel_clk) begin
        deltax <= (hcount > (x+RADIUS)) ? (hcount-(x+RADIUS)) : ((x+RADIUS)-hcount);
        deltay <= (vcount > (y+RADIUS)) ? (vcount-(y+RADIUS)) : ((y+RADIUS)-vcount);

        if(show2 && (deltax*deltax+deltay*deltay <= radiussquared))
            pixel <= {red_mapped, green_mapped, blue_mapped};
        else pixel <= 0;
    end


   // image_bits, for each bit, has a value from 0-16 (representing the color of the pixel)- there are 16 possible colors
    bell_rom myBellROM(.addr(image_addr),.clk(pixel_clk), .dout(image_bits));

    // use color map to create 8bits R, 8bits G, 8 bits B;
    // the roms are created by the system
    bell_rcm_rom bellredRom(.addr(image_bits), .clk(pixel_clk), .dout(red_mapped));
    bell_gcm_rom bellgreenRom(.addr(image_bits),.clk(pixel_clk),.dout(green_mapped));
    bell_bcm_rom bellblueRom(.addr(image_bits),.clk(pixel_clk), .dout(blue_mapped));

    // calculate rom address and read the location
    assign image_addr = (hcount-x) + (vcount-y) * WIDTH;

endmodule

//////////////////////////////////////////////////////////////////////
//
// background image sprite from generated image
//
//////////////////////////////////////////////////////////////////////
module backgroundImage
    #(parameter WIDTH = 800, HEIGHT = 600)  // default picture width // default picture height
    (input pixel_clk, input show2, input [10:0] x,hcount, input [9:0] y,vcount, output  reg [23:0] pixel);

    wire [18:0] image_addr;  // num of bits for 50x50 ROM
    wire [3:0] image_bits;  // only 16 colors available- each color maps to some (rgb) combination
    wire [7:0] red_mapped,green_mapped, blue_mapped;   // red, green, blue each expressed as a value from 0-255

    // note the one clock cycle delay in pixel!
    always @ (posedge pixel_clk) begin
        if (show2 && (hcount >= x && hcount < (x+WIDTH)) && (vcount >= y && vcount < (y+HEIGHT)))
            pixel <= {red_mapped, green_mapped, blue_mapped};
        else pixel <= 0;
    end

    // calculate rom address and read the location
    assign image_addr = (hcount-x) + (vcount-y) * WIDTH;

   // image_bits, for each bit, has a value from 0-16 (representing the color of the pixel)- there are 16 possible colors
    bg_rom myBackgroundROM(.addr(image_addr),.clk(pixel_clk), .dout(image_bits));

    // use color map to create 8bits R, 8bits G, 8 bits B;
    // the roms are created by the system
    image_rcm_rom imageredRom(.addr(image_bits), .clk(pixel_clk), .dout(red_mapped));
    image_gcm_rom imagegreenRom(.addr(image_bits),.clk(pixel_clk),.dout(green_mapped));
    image_bcm_rom imageblueRom(.addr(image_bits),.clk(pixel_clk), .dout(blue_mapped));

endmodule
```

D Verilog Code for Beam Detector

```
`timescale 1ns / 1ps

module receiver (
    input clk,
    input clk75us,
    input ir_input,

    output wire game_enable
);
    reg [9:0] count;

    reg isSignalBlocked;

    // just listen for when the signal has been 0 for a certain amount of time
    always @(posedge clk) begin
        if(clk75us) begin
            if(!ir_input) begin
                if(count > 10) count <= count;
                else count <= count+1;
            end
            else begin
                count <= 0;
            end
        end
        else count <= count;

    end


    assign game_enable = (count > 10);

endmodule


// sends a 600us signal, each "on" time is a 40 kHz signal
module transmitter (
    input clk,
    input reset,

    output signal_out
);

    reg [25:0] count_40khz; // used to be [10:0]
    reg [25:0] count_600us;
    reg signal_40khz;
    reg signal_600us;

    always @(posedge clk) begin
        if (reset) begin
            count_40khz <= 0;
        end

        else if (count_40khz == 337) begin
            count_40khz <= 0;
            signal_40khz <= !signal_40khz;
        end

        else begin
            count_40khz <= count_40khz + 1;
        end
    end

    always @(posedge clk) begin
```

```
        if (reset) begin
            count_600us <= 0;
        end

        else if (count_600us == 16200) begin
            count_600us <= 0;
            signal_600us <= !signal_600us;
        end

        else begin
            count_600us <= count_600us + 1;
        end
    end

    assign signal_out = signal_40khz && signal_600us;

endmodule
```

E Verilog Code for Sound Effects

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:09:39 11/29/2010
// Design Name:
// Module Name:    bell
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module bell(
    input enable,
    input ac97_ready,
    input clk,
    input reset,

    input [9:0] rel_height,

    output reg [7:0] ac97_data,

    // Debugging outputs
    output reg [13:0] dbg_mem_addr,
    output reg [1:0] dbg_ready_count,
    output reg dbg_state
);

    // FSM states
    parameter S_IDLE = 0;
    parameter S_PLAY = 1;

    parameter MEM_LENGTH = 'd11070;


    reg  [13:0] mem_addr;
    wire [15:0] mem_data;
    reg         state;
    reg   [1:0] ready_count;    // Count of AC97 ready signals

    reg         prev_enable;

    reg reset_state;


    // The sound memory. The FSM below it will generate the mem_addr signal;
    // the output of the memory goes directly to the AC97 unit.
    sound_mem sm(
        .addr(mem_addr),
        .clk(clk),
        .dout(mem_data)
    );


    wire [7:0] travel_sound;
```

```
// The travel sound.
travelSound ts(
   .reset(reset),
   .clk(clk),
   .rel_height(rel_height),
   .snd(travel_sound)
);


// This section is clocked on the AC'97 codec ready pulse, and is
// responsible for incrementing the mem addr counter when necessary.
always @(posedge ac97_ready) begin

    case (state)

        S_IDLE: begin
            mem_addr <= 0;
            ready_count <= 0;
        end

        S_PLAY: begin
            if (ready_count == 0) begin
                if (mem_addr + 1 == MEM_LENGTH) begin
                    mem_addr <= mem_addr;
                    ready_count <= 0;
                end
                else begin
                    mem_addr <= mem_addr + 1;
                    ready_count <= ready_count + 1;
                end
            end

            else begin
                mem_addr <= mem_addr;
                ready_count <= ready_count + 1;
            end
        end

        default: begin
            mem_addr <= 0;
            ready_count <= 0;
        end
    endcase
end

// This section is clocked on the 27 MHz clock and is responsible for
// starting and stopping playback.
always @(posedge clk) begin
    case (state)
        S_IDLE: begin
            if (enable && !prev_enable) begin
                state <= S_PLAY;
            end
            else begin
                state <= S_IDLE;
            end
        end

        S_PLAY: begin
            if (mem_addr + 1 == MEM_LENGTH) begin
                state <= S_IDLE;
            end
            else begin
                state <= S_PLAY;
            end
            end
        endcase

        reset_state <= 0;
        prev_enable <= enable;
    end


    // The actual output--this section is also responsible for multiplexing
    // the bell and the weight travel noise. If the bell FSM is playing, it
    // outputs the sound of the bell; otherwise it outputs the sound of the
    // weight traveling.
    always @( * ) begin
        if (state == S_PLAY)
            ac97_data[7:0] = mem_data[15:8];
        else
            ac97_data[7:0] = travel_sound[7:0];

        dbg_mem_addr = mem_addr;
        dbg_ready_count = ready_count;
        dbg_state = state;
    end


endmodule
```

F Verilog Code for Force Computation Logic

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    21:40:00 11/18/2010
// Design Name:
// Module Name:    force
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module force_mod (
    input reset,
    input clk,
    input trigger,
    input [7:0] adc_data,
    input adc_ready,

    output reg [10:0] velocity,
    output reg output_ready,

    output reg [27:0] accumulator,
    output reg state

);

    parameter [7:0] NEUTRAL_POINT = 'hAB;
    parameter [17:0] CYCLES = 'h2ADF0;

    parameter S_COUNTING = 1;
    parameter S_NOT_COUNTING = 0;


    reg [17:0] cycle_count;

    reg next_state;
    reg [27:0] next_accumulator;
    reg next_ready;

    reg [17:0] next_cycle_count;

    reg prev_trigger;

    always @( * ) begin
        if (state == S_NOT_COUNTING) begin
            // Currently not counting. If trigger is active, we should
            // transition to counting, otherwise stay in not counting.
            if (trigger && !prev_trigger) begin
                next_state = S_COUNTING;
                next_accumulator = 0;
            end
            else begin
                next_state = S_NOT_COUNTING;
                next_accumulator = accumulator;
            end

            next_cycle_count = 0;
        end

        else if ((state == S_COUNTING) && adc_ready) begin
            // Currently counting, and new data is available
            if (cycle_count >= CYCLES) begin
                // Reached our stable point: stop counting and strobe output
                next_state = S_NOT_COUNTING;
                next_cycle_count = 0;
            end

            else begin
                next_cycle_count = cycle_count + 1;
                next_state = S_COUNTING;
            end

            // Do some integratin'
            if (adc_data > NEUTRAL_POINT) begin
                next_accumulator = accumulator + ((adc_data - NEUTRAL_POINT));
            end
            else begin
                next_accumulator = accumulator + ((NEUTRAL_POINT - adc_data));
            end
        end

        else begin
            next_state = state;
            next_accumulator = accumulator;
            next_ready = 1;
            next_cycle_count = cycle_count;
        end
    end


    always @(posedge clk) begin
        if (reset) begin
            accumulator <= 0;
            state <= S_NOT_COUNTING;
            cycle_count <= 0;
        end
        else begin
            accumulator <= next_accumulator;
            state <= next_state;
            cycle_count <= next_cycle_count;
        end

        prev_trigger <= trigger;
    end


    always @( * ) begin
        velocity = accumulator[26:16];

        output_ready = (state == S_NOT_COUNTING);
    end
endmodule
```

G Verilog Code for Graphics

```
`timescale 1ns / 1ps

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

module pong_game (
    input vclock,     // 65MHz clock
    input reset,      // 1 to initialize module
    input up,         // 1 when paddle should move up
    input down,       // 1 when paddle should move down
    input [3:0] pspeed,  // puck speed in pixels/tick
    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 reg phsync,   // pong game's horizontal sync
    output reg pvsync,   // pong game's vertical sync
    output reg pblank,   // pong game's blanking
    output [23:0] pixel, // pong game's pixel

    input trigger_game,
    input [6:0] initial_vel,
    output reg bell_ringing,
    output [9:0] relative_ball_height

    );



    // SCREEN COORDINATES
    parameter SCREEN_X = 799;
    parameter SCREEN_Y = 599;

    // BALL VARIABLE
    parameter BALL_SIZE = 50;

    // BELL VARIABLE
    parameter BELL_SIZE = 80;

    // pipeline to handle drawing the circle
    reg [1:0] next = 0;

    // position variables
    parameter BALL_START = 555-25;
    parameter POLE_TOP = 105;
    parameter BELL_POS_Y = 40-40;   // 40 + 80/2 + 25 = 105 = final position of ball
    parameter BELL_POS_X = 184;
    reg [10:0] posX = 197;
    reg [9:0] posY = BALL_START;

    assign relative_ball_height = BALL_START-posY;

    // bell parameters
    reg [10:0] bell_x = BELL_POS_X;
    reg bell_osc = 0;

    // FSM Parameters
    reg [1:0] state;
    reg [1:0] nextState;
    parameter IDLE = 0;
    parameter CALCULATING = 1;
    parameter START_GAME = 2;
    parameter MOVING = 3;


    // add a one line delay
    reg show1, show2, ph, pv, pb;
    always@(posedge vclock) begin
        show1 <= ~reset;
        show2 <= show1;
        {phsync, ph} <= {ph,hsync};
        {pvsync, pv} <= {pv, vsync};
        {pblank, pb} <= {pb, blank};

        state <= nextState;
    end

    // draw the sprites
    wire [23:0] bell_pixel;
    wire [23:0] ball_pixel;
    wire [23:0] background_pixel;
    ball_sprite  #(.WIDTH(50), .HEIGHT(50))
        myBall(.pixel_clk(vclock), .show2(show2), .x(posX),.hcount(hcount),
        .y(posY-BALL_SIZE/2),.vcount(vcount), .pixel(ball_pixel));
    bell_sprite  #(.WIDTH(80), .HEIGHT(80))
        myBell(.pixel_clk(vclock), .show2(show2), .x(bell_x),.hcount(hcount),
        .y(BELL_POS),.vcount(vcount), .pixel(bell_pixel));
    // UNCOMMENT, just to speed up testing
    backgroundImage #(.WIDTH(800), .HEIGHT(600))  // default picture width // default picture height
        myBackgroundImage(.pixel_clk(vclock), .show2(show2), .x(0), .hcount(hcount),
        .y(0),.vcount(vcount), .pixel(background_pixel));

    /*** HEX VGA DISPLAY ****/
    wire [2:0] hex_pixel_small;
    wire [23:0] hex_pixel;
    wire [2:0] label_pixel_small;
    wire [23:0] label_pixel;
    wire [23:0] char_pixels;
    wire [31:0] score;
    wire [5*8-1:0] scoreString = "SCORE";
    wire [23:0] stringInitials;

    scoreToString stringGen(.vclock(vclock), .relative_height(relative_ball_height), .string(score));

    char_string_display highScoreLabel(.vclock(vclock),.hcount(hcount),.vcount(vcount),
                .pixel(label_pixel_small),.cstring(scoreString),.cx(10),.cy(10));
    defparam highScoreLabel.NCHAR = 5;
    char_string_display highScoreNumDisplay(.vclock(vclock),.hcount(hcount),.vcount(vcount),.pixel(hex_pixel_small),
                .cstring(score),.cx(10),.cy(40));
    defparam highScoreNumDisplay.NCHAR = 4;

    assign hex_pixel = { {8{hex_pixel_small[2]}}, {8{hex_pixel_small[1]}}, {8{hex_pixel_small[0]}} };
    assign label_pixel = { {8{label_pixel_small[2]}}, {8{label_pixel_small[1]}}, {8{label_pixel_small[0]}} };
    assign char_pixels = hex_pixel + label_pixel;




    // Z-Logic for the background and balls
    //  add the pixel values for bell and ball, they should never intersect, so this shouldn't be a problem
    assign pixel = (bell_pixel == 0 && ball_pixel == 0 && char_pixels == 0) ? background_pixel: bell_pixel + ball_pixel + char_pixel

    reg [6:0] vel;  // velocity of the ball
    reg goingUp = 1;

    always@(posedge vsync) begin

        case(state)

            IDLE: begin     // waiting for game to start
                if(trigger_game == 0) nextState <= CALCULATING;
                else begin
                    nextState <= IDLE;
                    posY <= BALL_START;
                end
            end // end IDLE
            CALCULATING: begin  // wait while calculating forces
                if(trigger_game == 1) nextState <= START_GAME;
                else begin
                    nextState <= CALCULATING;
                    posY <= BALL_START;
                end
            end //end CALCULATING
            START_GAME: begin       // start game, initialize velocity
                vel <= initial_vel;
                nextState <= MOVING;
            end // end START_GAME
            MOVING: begin                   // logic for acceleration and deacceleration of the ball
                if(next==0) begin           // timestep every 4th vsync
                    if(goingUp) begin       // going up the pole
                        nextState <= MOVING;
                        if(posY - vel*1 < POLE_TOP) begin   // hit the bell!
                            goingUp <=0;
                            posY <= POLE_TOP;
                            bell_ringing <= 1;
                        end
                        else if(vel < 5) begin  // didn't hit the bell
                            goingUp <= 0;
                            posY <= posY;
                        end
                        else begin      // still going up the pole
                            goingUp <= 1;
                            posY <= posY - vel*1;   // euler steps
                            vel <= vel - 5;         // v' = v + 1/2*a where a=g=-10
                        end
                    end
                    else begin  // going down
                        if(posY + vel*1 > BALL_START) begin // hit the ground
                            goingUp <= 1;
                            posY <= BALL_START;
                            vel <= 0;
                            bell_ringing <= 0;
                            nextState <= IDLE;
                        end
                        else begin  // still going down
                            goingUp <= 0;               // stay 0
                            posY <= posY + vel*1;   // euler steps
                            vel <= vel + 5;         // v' = v + 1/2*a where a=g=-10
                            nextState <= MOVING;
                        end
                    end

                    // when the bell rings, oscillate bell position
                    if(bell_ringing && bell_osc) begin
                        bell_x <= BELL_POS_X +2;
                        bell_osc <= 0;
                    end
                    else if (bell_ringing && !bell_osc) begin
```

```
                        bell_x <= BELL_POS_X-2;
                        bell_osc <= 1;
                    end
                    else
                        bell_x <= BELL_POS_X;

                next <= 1;
                // next adds a time step delay so doesn't accelerate as quickly
                end  //end if(next == 0)
                else if(next==1) next <= 2;
                else if(next==2) next <= 3;
                else next <=0;
            end // end MOVING


        endcase
    end // end always block

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (800 x 600 @ 60Hz refresh)
//
////////////////////////////////////////////////////////////////////////////////

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

   // horizontal: display 800 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount == 799);
   assign hsyncon = (hcount == 839);    // add front porch
   assign hsyncoff = (hcount == 887);   // add back porch
   assign hreset = (hcount == 1055);    // add front porch, sync, and back porch

   // vertical: display 600 pixels per line
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount == 599);
   assign vsyncon = hreset & (vcount == 604 );  // add front porch and back porch
   assign vsyncoff = hreset & (vcount == 608);  // add back porch to vsyncon
   assign vreset = hreset & (vcount == 626); // add front porch, sync, and back porch

   // 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
```

H Verilog Code for Infrared Receiver FSM

```
`timescale 1ns / 1ps

module receiverFSM(input clk, input clk75us, ir_input, output wire game_enable);
    reg [9:0] count;

    reg isSignalBlocked;

    // just listen for when the signal has been 0 for a certain amount of time
    always@(posedge clk) begin
        if (clk75us) begin
            if(!ir_input) begin
                if(count > 10) count <= count;
                else count <= count+1;
            end
            else begin
                count <= 0;
            end
        end
        else count <= count;
    end

    assign game_enable = (count > 10);

endmodule


// sends a 600us signal, each "on" time is a 40 kHz signal
module transmitter (input wire clk, reset, output wire signal_out);
    reg [25:0] count_40khz; // used to be [10:0]
    reg [25:0] count_600us;
    reg signal_40khz;
    reg signal_600us;

    always@(posedge clk) begin
        if (reset)
            count_40khz <= 0;
        else if (count_40khz == 337) begin
            count_40khz <= 0;
            signal_40khz <= !signal_40khz;
        end
        else
            count_40khz <= count_40khz + 1;
    end

    always@(posedge clk) begin
        if (reset)
            count_600us <= 0;
        else if (count_600us == 16200) begin
            count_600us <= 0;
            signal_600us <= !signal_600us;
        end
        else
            count_600us <= count_600us + 1;
    end

    assign signal_out = signal_40khz && signal_600us;

endmodule
```

I Verilog Code for Weight Travel Sound Effect

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    21:26:33 12/06/2010
// Design Name:
// Module Name:    travelSound
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module travelSound(
input reset,
input clk,
input [9:0] rel_height,

output reg [7:0] snd
);

// # of 27 MHz clock cycles for each range
parameter BIN1_CLOCK = 'd180000;
parameter BIN15_CLOCK = 'd135000;
parameter BIN2_CLOCK = 'd108000;
parameter BIN25_CLOCK = 'd90000;
parameter BIN3_CLOCK = 'd77140;
parameter BIN35_CLOCK = 'd67500;
parameter BIN4_CLOCK = 'd60000;
parameter BIN45_CLOCK = 'd54000;
parameter BIN5_CLOCK = 'd49000;
parameter BIN55_CLOCK = 'd45000;
parameter BIN6_CLOCK = 'd41500;
parameter BIN65_CLOCK = 'd38570;
parameter BIN7_CLOCK = 'd36000;
parameter BIN75_CLOCK = 'd33750;
parameter BIN8_CLOCK = 'd31800;
parameter BIN85_CLOCK = 'd30000;
parameter BIN9_CLOCK = 'd28400;
parameter BIN95_CLOCK = 'd27000;
parameter BIN10_CLOCK = 'd25700;

// Current # of clock cycles between transitions
reg [17:0] current_clock_freq;

// Current count of clock cycles
reg [17:0] current_count;


always @(posedge clk) begin

if (rel_height == 0) begin
snd <= 0;
current_count <= 0;
current_clock_freq <= BIN1_CLOCK;
end
```

```
else begin
// Set current clock
if (rel_height < 20)
current_clock_freq <= BIN1_CLOCK;
else if (rel_height < 40)
current_clock_freq <= BIN15_CLOCK;
else if (rel_height < 60)
current_clock_freq <= BIN2_CLOCK;
else if (rel_height < 80)
current_clock_freq <= BIN25_CLOCK;
else if (rel_height < 100)
current_clock_freq <= BIN3_CLOCK;
else if (rel_height < 120)
current_clock_freq <= BIN35_CLOCK;
else if (rel_height < 140)
current_clock_freq <= BIN4_CLOCK;
else if (rel_height < 160)
current_clock_freq <= BIN45_CLOCK;
else if (rel_height < 180)
current_clock_freq <= BIN5_CLOCK;
else if (rel_height < 200)
current_clock_freq <= BIN55_CLOCK;
else if (rel_height < 220)
current_clock_freq <= BIN6_CLOCK;
else if (rel_height < 240)
current_clock_freq <= BIN65_CLOCK;
else if (rel_height < 260)
current_clock_freq <= BIN7_CLOCK;
else if (rel_height < 280)
current_clock_freq <= BIN75_CLOCK;
else if (rel_height < 300)
current_clock_freq <= BIN8_CLOCK;
else if (rel_height < 320)
current_clock_freq <= BIN85_CLOCK;
else if (rel_height < 340)
current_clock_freq <= BIN9_CLOCK;
else if (rel_height < 360)
current_clock_freq <= BIN95_CLOCK;
else
current_clock_freq <= BIN10_CLOCK;


// Figure out output state
if (current_count == 0) begin
current_count <= current_clock_freq;
if (snd == 0)
snd <= 0'hF0;
else
snd <= 0;
end

else begin
current_count <= current_count - 1;
snd <= snd;
end
end
end
endmodule
```

J Verilog Code for Utilities

```
// pulse synchronizer
// NSYNC = number of sync flops. must be >= 2
module synchronize #(parameter NSYNC = 2) (
    input clk,in,
    output reg out);

    reg [NSYNC-2:0] sync;

    always @ (posedge clk) begin
        {out,sync} <= {sync[NSYNC-2:0],in};
    end
endmodule

// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
// .01 sec with a 27 MHz clock
module debounce #(parameter DELAY=270000) (
    input reset, clock, noisy,
    output reg clean);

    reg [18:0] count;
    reg new;

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



///////////////////////////////////////////////////////////////////////////////
// enable goes high every 75us, providing 8x oversampling for
// 600us width signal (with 27mhz clock)
///////////////////////////////////////////////////////////////////////////////
module divider_600us (
    input wire clk,
    input wire reset,
    output wire enable);

    reg [10:0] count;

    always@(posedge clk)
    begin
        if (reset)
            count <= 0;
        else if (count == 2024)
            count <= 0;
        else
            count <= count + 1;
    end
    assign enable = (count == 2024);
```

```
endmodule

///////////////////////////////////////////////////////////////////////////////
// A programmable timer with 75us increments. When start_timer is asserted,
// the timer latches length, and asserts expired for one clock cycle
// after 'length' 75us intervals have passed. e.g. if length is 10, timer will
// assert expired after 750us.
///////////////////////////////////////////////////////////////////////////////
module timer (
    input wire clk,
    input wire reset,
    input wire start_timer,
    input wire [9:0] length,
    output wire expired);

    wire enable;
    divider_600us sc(.clk(clk),.reset(start_timer),.enable(enable));
    reg [9:0] count_length;
    reg [9:0] count;
    reg counting;

    always@(posedge clk) begin
        if (reset)
            counting <= 0;
        else if (start_timer)
        begin
            count_length <= length;
            count <= 0;
            counting <= 1;
        end
        else if (counting && enable)
            count <= count + 1;
        else if (expired)
            counting <= 0;
    end

    assign expired = (counting && (count == count_length));
endmodule



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

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

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

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

    reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

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

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

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

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

    assign dreset = (reset_count != 0);

    assign disp_clock = ~clock;

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

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

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

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

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

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

            8'h03: begin
                // Latch dot data
                disp_ce_b <= 1'b1;
                dot_index <= 31;      // re-purpose to init ctrl reg
                disp_rs <= 1'b1; // Select the control register
                state <= state+1;
            end

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

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

            8'h06:
              begin
                 // Load the user's dot data into the dot reg, char by char
```

```
                     disp_ce_b <= 1'b0;
                     disp_data_out <= dots[dot_index]; // dot data from msb
                     if (dot_index == 0)
                       if (char_index == 0)
                         state <= 5;           // all done, latch data
                   else
                   begin
                     char_index <= char_index - 1; // goto next char
                     dot_index <= 39;
                   end
                     else
                   dot_index <= dot_index-1;   // else loop thru all dots
                  end

        endcase

    always @ (data or char_index)
        case (char_index)
            4'h0:       nibble <= data[3:0];
            4'h1:       nibble <= data[7:4];
            4'h2:       nibble <= data[11:8];
            4'h3:       nibble <= data[15:12];
            4'h4:       nibble <= data[19:16];
            4'h5:       nibble <= data[23:20];
            4'h6:       nibble <= data[27:24];
            4'h7:       nibble <= data[31:28];
            4'h8:       nibble <= data[35:32];
            4'h9:       nibble <= data[39:36];
            4'hA:       nibble <= data[43:40];
            4'hB:       nibble <= data[47:44];
            4'hC:       nibble <= data[51:48];
            4'hD:       nibble <= data[55:52];
            4'hE:       nibble <= data[59:56];
            4'hF:       nibble <= data[63:60];
        endcase

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

endmodule
```

K Verilog Code for Score Character Display

```
//
// File:   cstringdisp.v
// Date:   24-Oct-05
// Author: I. Chuang, C. Terman
//
// Display an ASCII encoded character string in a video window at some
// specified x,y pixel location.
//
// INPUTS:
//
//   vclock       - video pixel clock
//   hcount       - horizontal (x) location of current pixel
//   vcount       - vertical (y) location of current pixel
//   cstring      - character string to display (8 bit ASCII for each char)
//   cx,cy        - pixel location (upper left corner) to display string at
//
// OUTPUT:
//
//   pixel        - video pixel value to display at current location
//
// PARAMETERS:
//
//   NCHAR        - number of characters in string to display
//   NCHAR_BITS   - number of bits to specify NCHAR
//
// pixel should be OR'ed (or XOR'ed) to your video data for display.
//
// Each character is 8x12, but pixels are doubled horizontally and vertically
// so fonts are magnified 2x.  On an XGA screen (1024x768) you can fit
// 64 x 32 such characters.
//
// Needs font_rom.v and font_rom.ngo
//
// For different fonts, you can change font_rom.  For different string
// display colors, change the assignment to cpixel.


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

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

    parameter NCHAR = 4; // number of 8-bit characters in cstring
    parameter NCHAR_BITS = 3; // number of bits in NCHAR

    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 [2:0] 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 @* 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 [2:0] cpixel = (font_byte[7 - h] ^ reverse) ? 7 : 0;
    wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <= cx+NCHAR*16)
             & (vcount < cy + 24));
    wire [2:0] pixel = dispflag ? cpixel : 0;

endmodule


module scoreToString(input vclock, input [9:0] relative_height,
                            output reg [31:0] string);
    always@(posedge vclock) begin
        if(relative_height <= 36) string = "0000";
        else if(relative_height > 36 && relative_height <= 72) string = "0100";
        else if(relative_height > 72 && relative_height <= 108) string = "0200";
        else if(relative_height > 108 && relative_height <= 142) string = "0300";
        else if(relative_height > 142 && relative_height <= 170) string = "0400";
        else if(relative_height > 170 && relative_height <= 206) string = "0500";
        else if(relative_height > 206 && relative_height <= 242) string = "0600";
        else if(relative_height > 242 && relative_height <= 278) string = "0700";
        else if(relative_height > 278 && relative_height <= 304) string = "0800";
        else if(relative_height > 304 && relative_height <= 340) string = "0900";
        else if(relative_height > 340 && relative_height <= 376) string = "1000";
        else if(relative_height > 376 && relative_height <= 402) string = "1100";
        else if(relative_height > 402 && relative_height <= 438) string = "1100";
        else if(relative_height > 438) string = "1200";
        else string = "0000";
    end


endmodule
```

L Verilog Code for High Striker Game

```
`default_nettype none

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

module labkit(
    // AC97
    output wire /*beep,*/ audio_reset_b, ac97_synch, ac97_sdata_out,
    input wire ac97_bit_clock, ac97_sdata_in,

    // VGA
    output wire [7:0] vga_out_red, vga_out_green, vga_out_blue,
    output wire vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync, vga_out_vsync,

    // FLUORESCENT DISPLAY
    output wire disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b,
    input wire disp_data_in,
    output wire disp_data_out,

    // BUTTONS, SWITCHES, LEDS
    input wire button0,
    input wire button1,
    input wire button2,
    input wire button3,
    input wire button_enter,
    input wire button_down,
    input wire button_up,
    input wire [7:0] switch,
    output wire [7:0] led,

    // USER CONNECTORS, DAUGHTER CARD, LOGIC ANALYZER
    inout wire [31:0] user1,
    inout wire [31:0] user2,
    inout wire [31:0] user3,
    output wire [15:0] analyzer1_data, output wire analyzer1_clock,
    output wire [15:0] analyzer2_data, output wire analyzer2_clock,

    input wire clock_27mhz
);

    ///////////////////////////////////////////////////////////////////////////
    //
    // Reset Generation
    //
    // A shift register primitive is used to generate an active-high reset
    // signal that remains high for 16 clock cycles after configuration finishes
    // and the FPGA's internal clocks begin toggling.
    //
    ///////////////////////////////////////////////////////////////////////////
    // use FPGA's digital clock manager to produce a
    // 40MHz clock (actually 40.5MHz)
    wire clock_40mhz_unbuf,clock_40mhz;
    DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_40mhz_unbuf));
    // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
    // synthesis attribute CLKFX_MULTIPLY of vclk1 is 15
    // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
    // synthesis attribute CLKIN_PERIOD of vclk1 is 37
    BUFG vclk2(.O(clock_40mhz),.I(clock_40mhz_unbuf));


    wire power_on_reset;    // remain high for first 16 clocks
    SRL16 reset_sr (.D(1'b0), .CLK(clock_40mhz), .Q(power_on_reset),
           .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
    defparam reset_sr.INIT = 16'hFFFF;


    // ENTER button is user reset
    wire reset,user_reset;
    debounce db1(.reset(power_on_reset),.clock(clock_40mhz),.noisy(~button_enter),.clean(user_reset));
    assign reset = user_reset | power_on_reset;


    wire bell_ringing;       // 1 if bell was hit, 0 if bell was not
    wire [6:0] initial_vel;  // Initial velocity of the ball, ranges from 0-70
    wire [9:0] ball_height;

    wire [7:0] adc_data;    // Data from ADC interface module
    wire adc_ready;         // Ready signal from ADC interface module

    wire beam_enable;       // Enable signal from beam detector


    /****** BEAM DETECTOR *****/
    // Beam transmitter
    transmitter tx(.clk(clock_27mhz), .reset(reset),.signal_out(user2[1]));

    // Beam detector trigger
    wire enable75;
    divider_600us enable75us(.clk(clock_27mhz),.reset(reset), .enable(enable75));
    receiver rx(.clk(clock_27mhz), .clk75us(enable75), .ir_input(~user2[0]),.game_enable(beam_enable));


    /****** ACCELEROMETER ******/
    // Force calculator
    wire [6:0] velocity;
    wire [27:0] accumulator;
    wire force_ready;
    wire force_state;
    //wire [3:0] stable_count;

    force_mod f(
        .reset(reset),
        .clk(clock_27mhz),
        .trigger(beam_enable),
        .adc_data(adc_data),
        .adc_ready(adc_ready),
        //.STABLE_CYCLES(switch[7:0]),

        .velocity(velocity),
        .output_ready(force_ready),
        .accumulator(accumulator),
        .state(force_state)
        //.debug_stable_count(stable_count)
    );


     /***** AUDIO MODULES ******/
    wire [7:0] ac97_data;
    wire ac97_ready;

     // ADC interface module
```

```
adc a(
    .adc_data(user1[7:0]),
    .adc_int(user1[9]),
    .clk(clock_27mhz),

    .data(adc_data),
    .ready(adc_ready),
    .adc_rd(user1[8])
);


// The audio interface
lab4audio aud(
    .clock_27mhz(clock_27mhz),
    .reset(reset),
    .volume('d15),
    .audio_out_data(ac97_data),
    .ready(ac97_ready),
    .ac97_sdata_out(ac97_sdata_out),
    .ac97_sdata_in(ac97_sdata_in),
    .ac97_synch(ac97_synch),
    .ac97_bit_clock(ac97_bit_clock),
    .audio_reset_b(audio_reset_b)
);


// The bell
wire [13:0] bell_mem_addr;
wire [1:0] ready_count;
wire bell_state;
bell myBell(
    .enable(bell_ringing),          // enable bell
    .ac97_ready(ac97_ready),
    .clk(clock_27mhz),
    .reset(reset),
    .ac97_data(ac97_data),
    .dbg_mem_addr(bell_mem_addr),
    .dbg_ready_count(ready_count),
    .dbg_state(bell_state),
    .rel_height(ball_height)
);


/******* GRAPHICS MODULE *************/

// UP and DOWN buttons for pong paddle
wire up,down, button3_db;
debounce db2(.reset(reset),.clock(clock_40mhz),.noisy(~button_up),.clean(up));
debounce db3(.reset(reset),.clock(clock_40mhz),.noisy(~button_down),.clean(down));
debounce db4(.reset(reset),.clock(clock_40mhz),.noisy(~button3),.clean(button3_db));


assign initial_vel = ((velocity>switch[5:0])) ? (velocity-switch[5:0]) : 0;

// generate basic XVGA video signals
wire [10:0] hcount;
wire [9:0]  vcount;
wire hsync,vsync,blank;
xvga xvga1(
    .vclock(clock_40mhz),
    .hcount(hcount),
    .vcount(vcount),
    .hsync(hsync),
    .vsync(vsync),
    .blank(blank));
```

```
 // feed XVGA signals to user's pong game
 wire [23:0] pixel;
 wire phsync,pvsync,pblank;
 pong_game myGraphics(
     .vclock(clock_40mhz),
     .reset(reset),
     .up(up),
     .down(down),
     .pspeed(switch[7:4]),
     .hcount(hcount),
     .vcount(vcount),
     .hsync(hsync),
     .vsync(vsync),
     .blank(blank),
     .phsync(phsync),
     .pvsync(pvsync),
     .pblank(pblank),
     .pixel(pixel),
     .trigger_game(force_ready),
     .initial_vel(initial_vel),
     .bell_ringing(bell_ringing),
     .relative_ball_height(ball_height));


reg [23:0] rgb;
reg b,hs,vs;
 reg ph;

 // delay the pixel output by 8
 parameter DELAY = 8;
 reg [DELAY:0] hsyncDelayed;
 reg [DELAY:0] vsyncDelayed;

 always @(posedge clock_40mhz) begin
     hsyncDelayed <= {hsyncDelayed[DELAY-1:0], phsync};
     vsyncDelayed <= {vsyncDelayed[DELAY-1:0], pvsync};
     hs <= hsyncDelayed[DELAY];
     vs <= vsyncDelayed[DELAY];
     b <= pblank;
     rgb <= pixel;
 end

 // VGA Output.  In order to meet the setup and hold times of the
 // AD7125, we send it ~clock_40mhz.
 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_40mhz;
 assign vga_out_hsync = hs;
 assign vga_out_vsync = vs;

 assign led = ~{0, force_ready, ~beam_enable, up,down,reset,switch[1:0]};


 /***** LOGIC ANALYZER SIGNALS ******/
 assign analyzer1_data[10:0] = velocity;
 assign analyzer1_data[11] = force_ready;
 assign analyzer1_data[12] = beam_enable;
 assign analyzer1_data[13] = force_state;
 assign analyzer1_clock = adc_ready;
 assign analyzer1_data[15:14] = 0;
 assign analyzer2_clock = clock_27mhz;
```

```
    assign analyzer2_data[7:0] = ac97_data;
    assign analyzer2_data[8] = bell_ringing;
    assign analyzer2_data[15:9] = 0;
    //assign analyzer2_data = 0;

    assign user1[31:10] = 0;
    assign user2[31:2] = 0;

    /****** Hex Display *****/
    // use this to display the decoded commands and for debug
    wire [63:0] my_hex_data = {3'b0, beam_enable, 3'b0, force_ready, 1'b0, initial_vel, 48'b0};
    display_16hex disp(
        .reset(reset),
        .clock_27mhz(clock_27mhz),
        .data(my_hex_data),
        .disp_blank(disp_blank),
        .disp_clock(disp_clock),
        .disp_rs(disp_rs),
        .disp_ce_b(disp_ce_b),
        .disp_reset_b(disp_reset_b),
        .disp_data_out(disp_data_out));


endmodule
```