Gestural Remote Control Using FPGA

Final Report

Andres Hasfura, Ricardo Jasinski, Vladimir Eremin

December 12, 2012

Abstract

Universal remote controls are usually large and unintuitive devices. While appealing to the more technology savvy users, most individuals would probably enjoy a less clunky and mind-taxing interface. We propose the development of a gestural interface to control infrared-enabled devices with simple hand movements.

In the proposed system, the user's hands are identified by wearing red and green gloves. The X and Y coordinates of each hand's center of mass are calculated, and fed into a gesture recognition state machine. This FSM classifies the hand movements into a set of predefined gestures, which trigger a corresponding infrared command and play an audio signal to provide feedback to the user.

For the scope of this project, we control a TV set and work with a predefined number of gestures. However, the proposed design could be generalized to operate with multiple devices and a larger number of hand movements.

Contents

1 Overview

Universal remotes are infrared controllers that can be programmed to operate many different appliances. Because they must work with different brands and models, these devices usually have a large number of buttons and an unintuitive user interface. The task of programming and using a universal remote is often convoluted, involving elaborate key sequences.

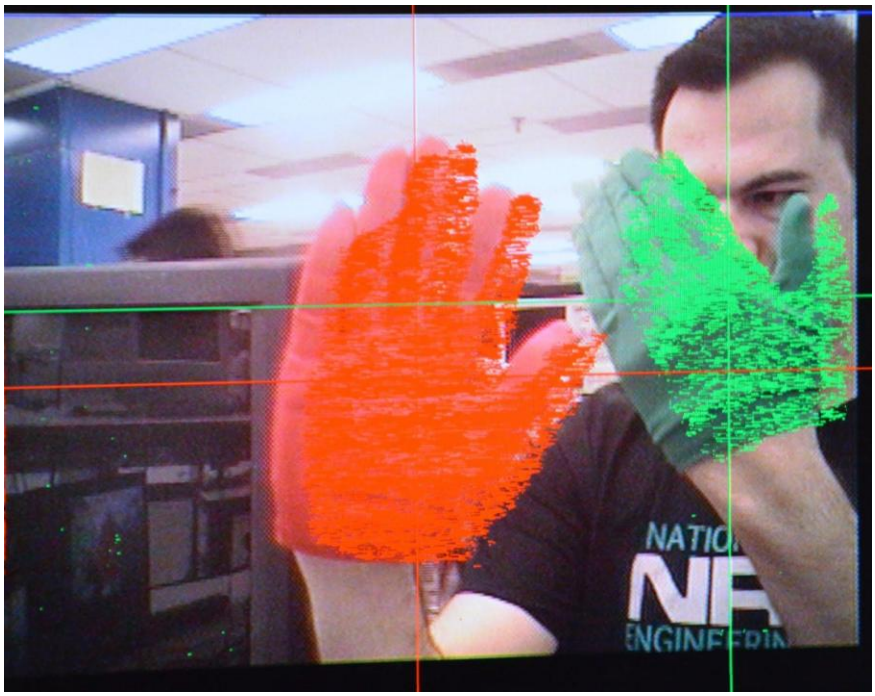

Figure 1: A screen capture of the Gestural Remote Control video output.

To provide a better user experience, we propose a gestural remote control able to operate infrared-enabled devices with simple hand movements. In the proposed system, the user's hands are identified by red and green gloves. The X and Y coordinates of each hand's center of mass are calculated and input into a gesture recognition finite state machine (FSM). This FSM classifies the hand movements into a set of predefined gestures, and sends the corresponding infrared command to the controlled device.

Major design decisions include:

- Video is captured with a standard NTSC camera. A dedicated chip in the labkit decodes the NTSC signal and provides the FPGA with digital pixel data, allowing for a hardware-only solution that would not be achievable with a USB camera.
- Infrared commands are encoded using the Sony™ Infrared Protocol. This allows the remote to be used with a wide range of existing devices.

- The gestural interface recognizes only a predefined set of hand movements. Even though it could be possible to make the controller learn new gestures, this is not an essential feature, and would add unnecessary risk to the project.
- Audio feedback is provided by playing wave files stored in a compact flash card inside the labkit. The main advantage of this solution is that it does not require any external hardware. On the other hand, it requires additional resources to control the flash memory interface, and to communicate with a host computer for uploading the audio files.

2 Implementation

The proposed design is organized into three major blocks: video processing, gestures recognition, and system output. The video processing block takes in the NTSC signal acquired from the camera and provides the x and y coordinates of the user's hands. The gesture recognition block classifies the hand movements into a set of predefined gestures. Finally, the system output block provides audio feedback and and send the infrared commands to the controlled device. Figure 2 shows a detailed block diagram of the proposed system. Each of the system's blocks is described next.

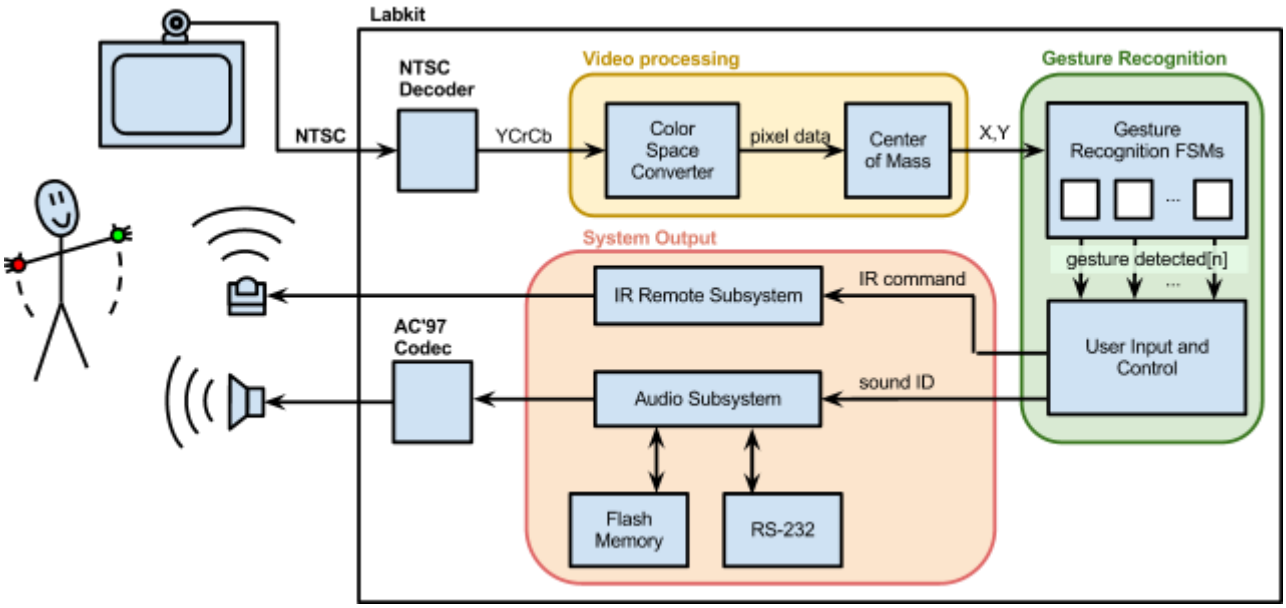

Figure 2. System block diagram.

2.1 Video Processing (Vladimir)

Figure 3. NTSC camera.

2.1.1 NTSC Decoder (Vladimir)

This module receives the input NTSC signal from the camera (see Figure 3) and outputs a 30-bit YCrCb value along with other control signals, such as horizontal and vertical synchronism pulses. The actual analog to digital conversion is performed by an external ADV7185 video decoder chip. The main task of this module is to initialize the decoder chip and to adapt the YCrCb pixel data into a format that can be readily used by the subsequent modules.

2.1.2 Color Space Converter (Vladimir)

This module receives YCrCb pixel values from the video decoder and transforms the color encoding from YCrCb to RGB and then to HSV. These color conversions are linear transformations and can be performed using hardware multipliers and instantiated dividers.

While YCrCb to RGB conversion is required only for storing the frame in the zbt memory and displaying it to the LCD screen, the RGB to HSV conversion is necessary to get an accurate color estimation in poor lighting conditions of the lab. A hue-saturation-lightness filtering performs a lot better in that non-ideal environment. That is why we use RGB to HSV conversion which makes a 24-bit HSV value from an 18-bit RGB value, filling out two least significant bits with zeros and dividing these values by predefined coefficients.

Because of the pipelined dividers' usage, this block adds a 22 clock cycle delay to the acquired video signal.

2.1.3 Center of Mass (Vladimir)

This module takes in the HSV values of each pixel from an image frame along with its corresponding hcount and vcount, and outputs the center of mass of the user's hands (see Figure 4). The outputs are two hand indicating flags and two 11 bit values for x left and right hand coordinates (in pixels) as well as two 10 bit values to be y coordinates of the same hands.

The center of mass is found by accumulating the hcount and vcount of pixels whose hsv values pass the threshold and keep a count of the number of pixels which passed the test. At the end of the whole frame processing we send the accumulated value and the count to a divider to calculate the average value, which is by definition the center of mass.

At the beginning of each frame, we clear both the count and accumulator and calculate a new center of mass at the end of the frame.

Due to the usage of 28 by 16 bit Xilinx LogiCORE Pipelined Dividers v3.0, this module has a latency of 29 cycles.

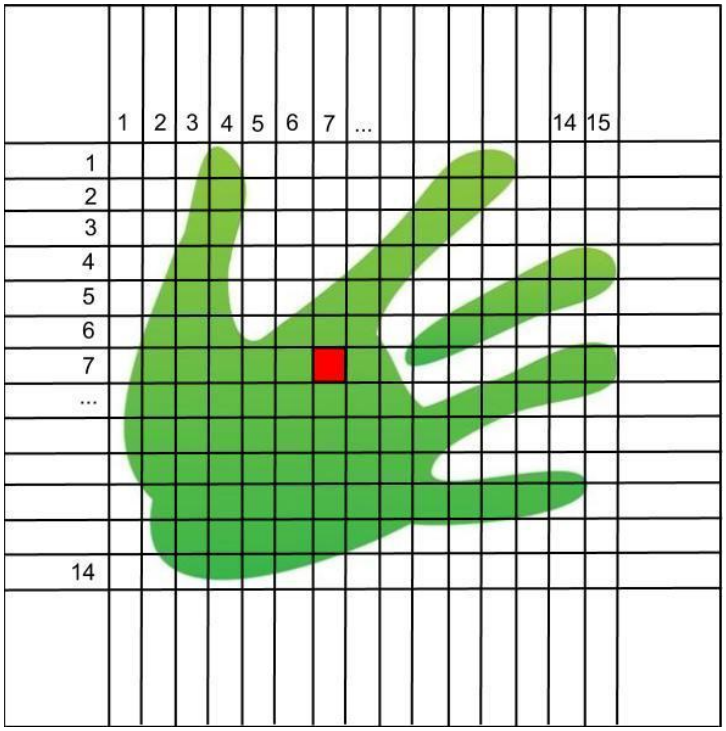

Figure 4. Center of mass calculation.

2.2 Gesture Recognition (Ricardo)

After the *x* and *y* coordinates of each hand are calculated by the center of mass module, they need to be recognized by the system and translated into specific gestures. Table I shows the gestures that can be detected by the system and the corresponding infrared commands.

Table I. Sample recognized by the GR FSMs and corresponding functions.

Gesture	Function
Waving to the camera	TV Power
Swiping left and right	TV Channel -/+
Holding right hand up	TV Volume +
Holding left hand up	TV Volume -
Drawing a "T" with both hands	TV Mute

Figure 5 shows the implementation diagram of the gestures recognition subsystem. Each of the two internal modules is explained next.

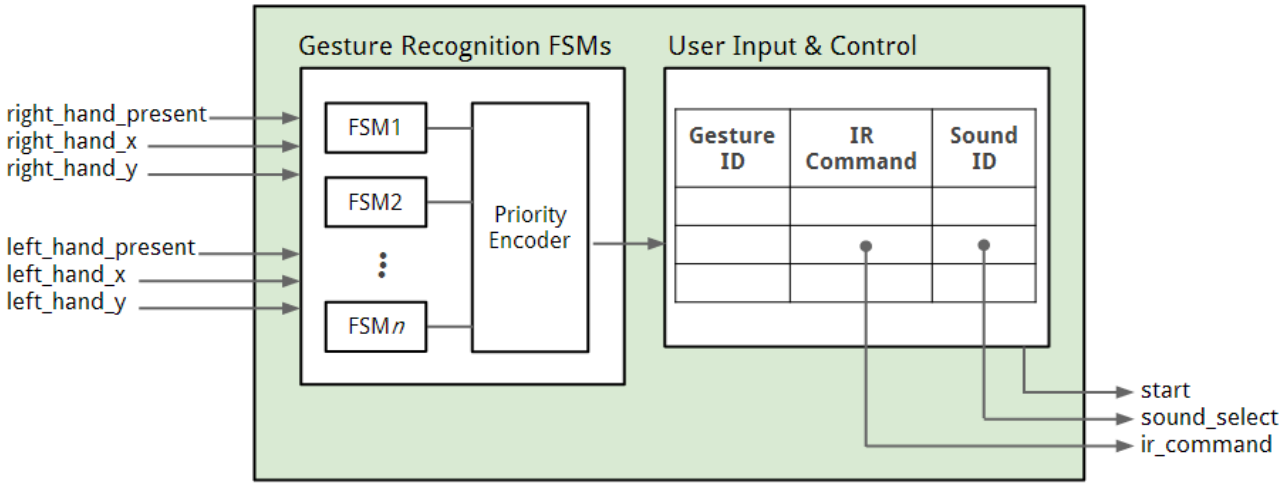

Figure 5. Implementation diagram of the gesture recognition module.

2.2.1 Gesture Recognition FSMs (Ricardo)

The Gesture Recognition (GR) FSM classifies the user's hand movements into a set of predefined gestures. The inputs to this module are the X and Y coordinates for each hand, and the outputs are a set of control signals indicating when each gesture has been recognized.

Internally, the GR module is composed of a series of minor state machines, one for each gesture. The state machines are all reset when the hands leave the field of view (FOV). Each FSM transitions through a series states as the hands are moved, asserting a status line when a gesture is completed. The states correspond to predefined regions of the screen. Figure 6 shows the four regions tracked by the "right swipe" FSM.

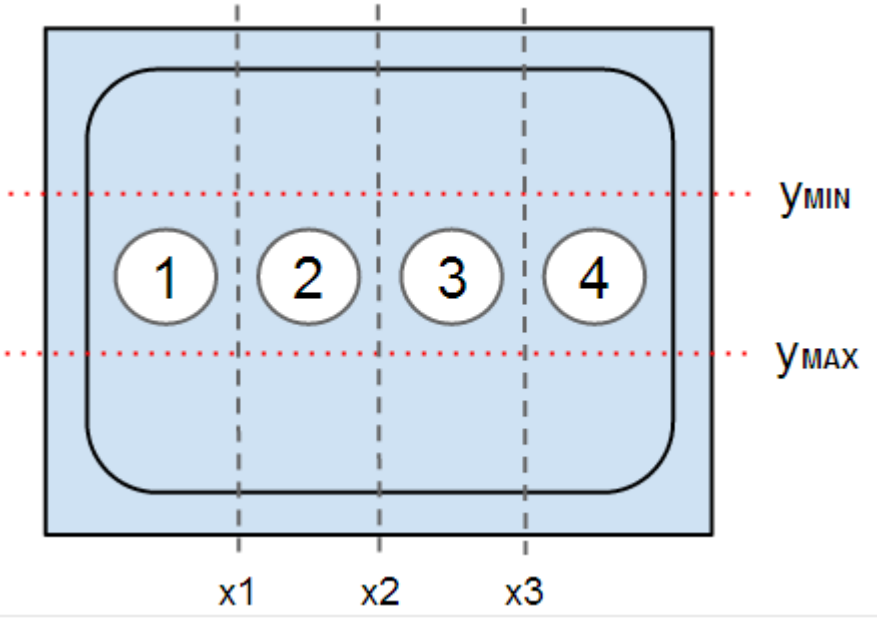

Figure 6. Screen regions tracked by the "swipe right" FSM.

As the user's hand moves from left to right, the FSM transitions through each state in the sequence. Figure 7 shows a state diagram for the "swipe right" FSM, illustrating the sequence of states for this FSM: *Idle* → *Region 1* → *Region 2* → *Region 3* → *Gesture detected*. The machine returns to idle after emitting a "gesture detected" pulse.

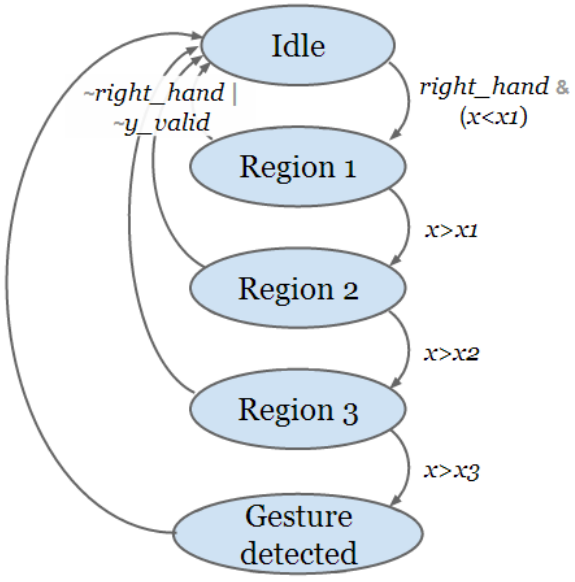

Figure 7. State diagram for the "swipe right" FSM.

The gestures recognition module does not present a performance bottleneck to the system, since its inputs are simply four integer numbers (the X and Y coordinates for each hand) and no complex computations are necessary. Additionally, because hand movements occur at a relatively low speed, the conventional NTSC frame rate of 30 frames per second was enough to provide a good responsiveness.

2.2.2 User Input and Control (Ricardo)

The User Input and Control Module (UICM) initiates and controls a sequence of events when a user gesture is recognized. The inputs to this module are the hand movements identified by the gesture recognition FSM, and the outputs are the control signals to communicate with the infrared and audio subsystems. This module is implemented as a simple table lookup: the gesture ID is used as an index to access the table, and each row contains the corresponding infrared command and the sound ID of the corresponding audio file.

This module operates with simple control lines as inputs and outputs, and is not expected to present a performance bottleneck to the system. Differently from the other modules in the system, the UICM is configurable and stores its settings in internal registers, in a table-like data structure. Each row associates a gesture ID with an infrared command and an audio file, which are forwarded to the corresponding subsystems when the gesture is detected.

2.3 System Output (Andres)

System Output Introduction:
The system output module is in charge of handling everything involving the audio. From beginning to end, this meant starting with a wav file which was downloaded from the Internet and outputting from the on-board flash chip the audio file to a pair of external speakers connected to the labkit. This portion of the project was split into two totally different projects, one being writing the wav files to the flash chip, the other was actually reading the correct portion from the flash whenever the module received a sound selection corresponding to a recognized gesture from the gestural recognition FSM.

2.3.1 Wiring up the USB to FIFO module (Andres)

This section covers the external wiring of the USB to FIFO module. specs from the data sheet were read for the USB245M module which was given, and the USB245M module was wired up connecting power and ground to the prospective pins, and connecting rxf and rd to the user1 i/o ports of the labkit. Rxf was a signal that when low meant that there was something at the input buffer. If rxf was low and rd was pulled low, the first byte in the buffer would be output to the module's eight bit pins. We initially tested the USB to FIFO module by wiring rd to an external switch. I wrote a python script to send one byte at a time after each time I pressed enter on my PC's keyboard, an alternating 8'd0 and 8'd255. When a byte was sent, rxf was probed using the oscilloscope to watch as it went active low, then rd was driven low with the switch, there by sending 255 or 0 to the output pins. These pins were also probed with the channel two of the oscilloscope to check to see everything was working correctly. Once it was functioning correctly, a python script was developed to write the proper information to the USB to FIFO module and have the FPGA program take care of automating rd while reading rxf in.

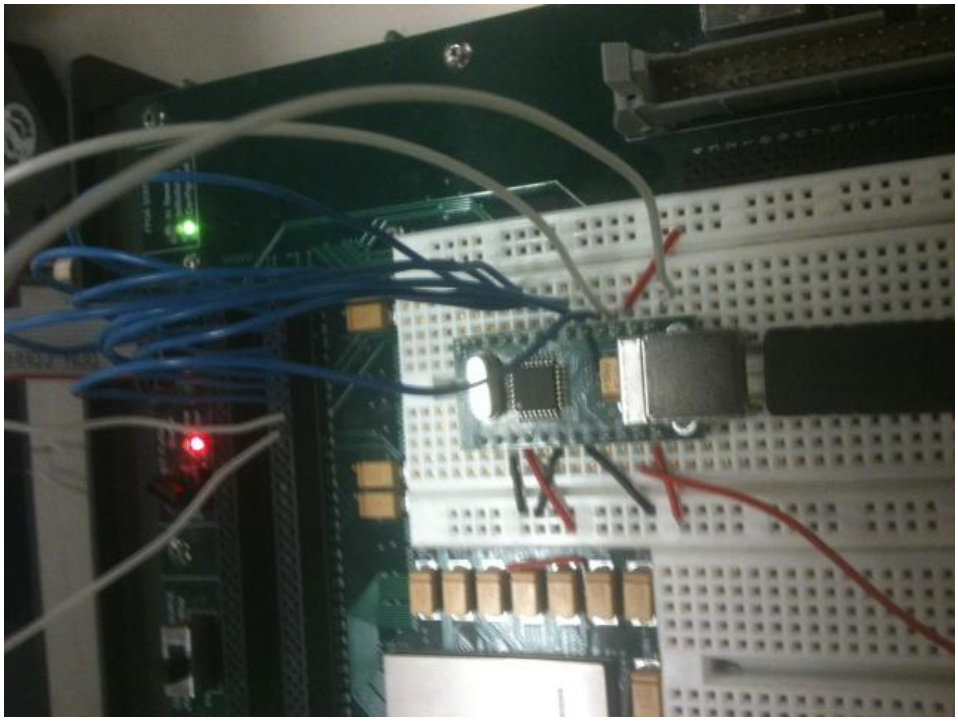

Figure 8. USB to FIFO module (USB245M) wired up to write to the on-board flash memory

2.3.2 PC to FPGA Serial Communication (Andres)

In this portion of the project, the original wav file is converted and sent to the USB to FIFO module from PC. To begin with, six different wav files were downloaded from the Internet, each of the wav files belonging to a particular gesture. These wav files were then converted into coe files using the given matlab script. These coe files had to be sent byte by byte over USB to the USB to FIFO module, so a python script was written which established a serial connection with the USB, then iterated through each coe file, converted each line (except for the first two, which are only text) from 8 integers into a byte which could be written the on-board flash chip. The script also increased the length of each file to 32,768 lines. This was done by looping through the sounds file 32,768 times, and if the length of the file was smaller than 32,768, the program would add 8 zeros instead of data found in the file. This would help simplify code later in the audio subsystem module. The index value that would be sent to the flash can be determined with ease, because instead of having different length files written sequentially into the flash, the same length file allowed for a constant sized index. Once this data has begun being sent via USB, it is ready to be loaded into the flash chip.

2.3.3 Flash Memory Interface (Andres)

This section of the project handles the interface between the USB to FIFO module and the flash chip itself. Much of the verilog code is written already through the “USB module” and the “flash manager” modules. These modules interact with each other to find when to send the flash more data from the USB to FIFO module, and when to load more data into the USB to FIFO module's buffer. Data input to the USB module would only be sent byte at a time to the flash manager

whenever the flash manager de-asserted a “busy” signal. Whenever busy went low, the data was sent to flash manager, and the flash manager would then load it into flash and increment the flashes’ memory address. The flash manager module could also erase the memory and read back memory depending on what its inputs were. Because data only needed to be written to the non-volatile flash chip once, at the beginning of the project writedata was tied high, which only allowed for data to be written. Once the flash chip was written to, writedata was permanently tied low so data would only be read from the flash memory.

2.3.4 Audio Subsystem (Andres)

This module controls when and what to play after receiving sound selections from a recognized gesture. It takes as input the sound select, data from the flash chip, and a ready pulse from the ac97, and outputs the flash address and the data from the flash chip to the ac97. The ac97 output is then connected to the external speakers, which actually produce the audio feedback to the gestural remote user. The audio subsystem module initially waits for a sound select to transition this simple FSM from the idle state to the playing state. Once a sounds select is received, the module also produces a value in a “base” register which contains the address of the beginning of where the audio for the recognized gesture is stored. The audio submodule then takes the byte from the flash memory located at the address and connects it to the ac97 on every third ac97 pulse, so as to down sample the audio file. Also at every third ac97 ready pulse another register containing the offset from the base would be incremented, and the sum of the offset and base are sent to the flash chip as the requested address. This allows the entirety of the desired audio file to be played. The module stops when the offset register is equal to 32,767, because that is the size of every audio file. This transitions the module into the “idle” state again, which resets the base and offset register and again waits for another sound select input.

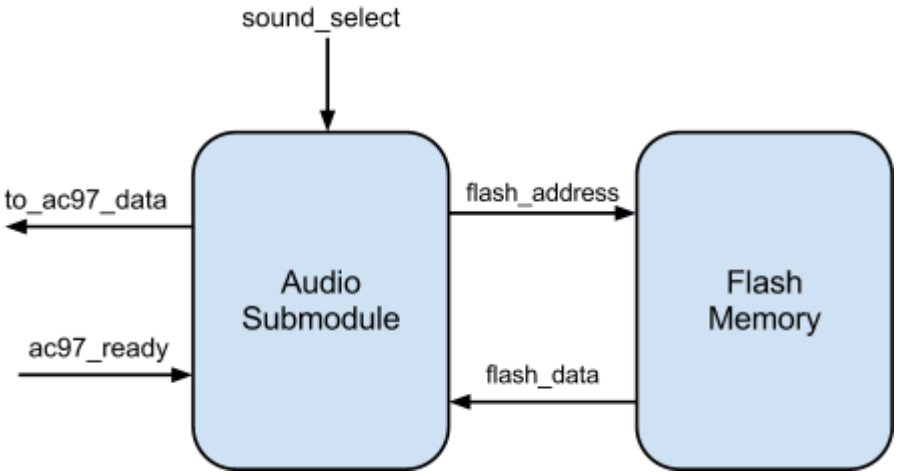

FIgure 9: A block diagram of how the Audio Submodule interfaces with the ac97 and flash memory

2.3.5 Infrared Remote Subsystem

The infrared remote subsystem is used to send commands to the controlled device. Its inputs are a command signal (7 bits wide), a device address (5 bits wide), and a start pulse. With these inputs, the subsystem will send the infrared sequence to produce the requested command. This module is very similar to lab 5b, and can reuse most of its source code.

3 External Components

The proposed design requires few additional components. The main external peripheral is a standard NTSC camera, which provides the video input for the system. Because the labkit has a composite video input and a NTSC decoder, no extra hardware is required to use the camera.

Audio and visual feedback also exploit the existing labkit resources. The LEDs and hex displays will be used to provide debug and status information. Sound will be generated with the built-in AC'97 codec and output with the standard speakers available in the 6.111 lab. A USB to FIFO module will also be required to initially write data to the on-board flash chip.

Finally, the user will be required to use a pair of colored gloves (red for the right hand, and green for the left hand). These gloves are the only external component that must be purchased for the project.

4 Testing

Our testing approach was based on three principles: extensive behavioral testbenches for data-oriented blocks, simple testbenches and early hardware testing for interface blocks, and continuous integration for the whole project.

In order to provide for better testability, the system was divided into small modules with well-defined functions and clear interfaces. The more data-oriented blocks (e.g., color converter, center of mass calculation, gesture recognition FSMs) will be thoroughly tested with behavioral testbenches to ensure their correctness. On the other hand, blocks that interface with external components (e.g., NTSC decoder, flash memory controller, AC'97 interface) will be tested with simpler testbenches that only verify their basic operation. However, these modules will be coded and tested first, in order to detect potential problems early and to exercise them for as long as possible in the course of the project.

The source code for the project will be put under version control (using SVN) and every developer will have access to the latest version of each file. The synthesis tool will be configured to use source files directly from SVN, providing for continuous integration of all system modules. This will allow us to detect problems early when a change inadvertently breaks the build.

5 Task Breakdown

The modular approach used in the system design facilitated the division of labor, because each block could be handed to an individual team member. The development of each block was considered complete only after the block had been coded and tested, using the respective testbench and physically downloading to the labkit. Table II shows the tasks assigned to each team member.

Table II. Division of labor.

System Block	Team Member
NTSC Decoder	Vladimir
YCrCb to RGB/HSV	Vladimir

Center of Mass	Vladimir
Gesture Recognition FSM	Ricardo
User Input and Control FSM	Ricardo
IR Remote Subsystem	Ricardo
Audio Subsystem	Andres
PC to FPGA Serial communication	Andres
Flash Memory Interface	Andres

6 Conclusion

During the period of approximately five weeks we were able to implement all the intended functionality for the gestural remote, including some stretch goals such as simultaneous tracking of the left and rights hands, and both-handed gestures. One of the main difficulties during the implementation was the long compile time in Xilinx's ISE tool. Even for a small change (such as changing hue values for the color detection), we faced a delay of approximately fifteen minutes between making a change to the code and seeing its effects. In retrospect, we believe that a wise strategy would have been to design the system in a more debugging-friendly manner, such as wiring the labkit switches to frequently changed variables . This would allow us to test changes on the fly, without requiring a new compilation.

Another technique that we found useful was the use of BRAM instead of flash memory while the flash interface was not available. This allowed us to test the audio subsystem before having a functional non-volatile storage module and a PC file transmission interface.

We believe that our division of labor was correct and adequate, with each team member putting in approximately the same amount of work. The simple and narrow interfaces between the modules helped us develop the system parts independently, and integration worked without significant problems. Overall, we had a great experience working on this project and learned how to use several new technologies, such as the USB module, the NTSC camera, and the flash and ZBT memories.

Appendix I - Verilog Source Code

```
// start of file rtl/audio/audio_subsystem.v
////////////////////////////////////////////////////////////////////////////////
// Audio subsystem.
////////////////////////////////////////////////////////////////////////////////

// File name                 # samples (@8kHz)     # samples (@16kHz)
// sound_00_tv_power              7440                  14880
// sound_01_tv_mute               9040                  18080
// sound_02_tv_volume_up         10720                  21440
// sound_03_tv_volume_down       12480                  24960
// sound_04_tv_channel_up        11720                  23520
// sound_05_tv_channel_down      12240                  24480

module audio_subsystem(
    // 64.8 MHz system clock
    input wire clock,
    input wire reset,
    // 1 when AC97 data is available
    input wire ready,
    // 1 to start playing the selected file
    input wire start,
    // sound ID of the audio file to be played
    input wire [7:0] sound_select,
    // 8-bit PCM data to headphone
    output reg [7:0] to_ac97_data,
    // 1 to play from flash; 0 to play from bram
    input use_flash_memory,
    // data read from flash
    input [15:0] flash_read_data,
    // address to read data from
    output [23:0] flash_read_address,
    // read pulse for flash manager
    output flash_read_request
);
    parameter AUDIO_FILES_COUNT = 6;

    // number of samples in each audio file (i.e., length of each BRAM)
    wire [15:0] samples_count [AUDIO_FILES_COUNT-1:0];
    assign samples_count[0] = 16'd14880;
    assign samples_count[1] = 16'd18080;
    assign samples_count[2] = 16'd21440;
    assign samples_count[3] = 16'd24960;
    assign samples_count[4] = 16'd23520;
    assign samples_count[5] = 16'd24480;

    // registered value of identifier specifying the audio file to be played
    reg [7:0] selected_sound_id;
    // number of samples in the sound file associated with the current sound ID
    reg [15:0] selected_sound_samples_count;
    // counter to reduce the sampling rate; a new sample is presented at the
    // output only when the counter overflows
    reg [2:0] oversampling_counter = 0;
    // index of current sample in the audio roms
    reg [23:0] audio_rom_offset = 0;

    reg [23:0] selected_sound_base_address = 0;

    // copy output of the currently used rom to the module output
    wire [7:0] rom_outputs [AUDIO_FILES_COUNT-1:0];

    // selected audio sample to be sent to AC97, output of one of the roms
    wire [7:0] rom_audio_sample =
        (selected_sound_id < AUDIO_FILES_COUNT) ? rom_outputs[selected_sound_id] : 0;

    wire [7:0] flash_audio_sample;

    wire [7:0] audio_sample =
        use_flash_memory ? flash_audio_sample : rom_audio_sample;
```

```
    // instantiate the audio roms with the sound samples
    sound_00_tv_power_audio_rom
audio_rom_00(.clka(clock),.addra(audio_rom_offset[13:0]),.douta(rom_outputs[0]));
    sound_01_tv_mute_audio_rom
audio_rom_01(.clka(clock),.addra(audio_rom_offset[14:0]),.douta(rom_outputs[1]));
    sound_02_tv_volume_up_audio_rom
audio_rom_02(.clka(clock),.addra(audio_rom_offset[14:0]),.douta(rom_outputs[2]));
    sound_03_tv_volume_down_audio_rom
audio_rom_03(.clka(clock),.addra(audio_rom_offset[14:0]),.douta(rom_outputs[3]));
    sound_04_tv_channel_up_audio_rom
audio_rom_04(.clka(clock),.addra(audio_rom_offset[14:0]),.douta(rom_outputs[4]));
    sound_05_tv_channel_down_audio_rom
audio_rom_05(.clka(clock),.addra(audio_rom_offset[14:0]),.douta(rom_outputs[5]));


    assign flash_read_request = ready;
    assign flash_read_address = selected_sound_base_address + audio_rom_offset;
    assign flash_audio_sample = flash_read_data[7:0];

    parameter ST_IDLE = 0;
    parameter ST_PLAYING = 1;
    parameter ST_REPLAYING = 2;
    parameter ST_DONE = 3;

    reg [1:0] current_state, next_state;

    // next state logic
    always @(*) begin
        next_state = current_state;
        case (current_state)
            ST_IDLE:
                if (start) next_state = ST_PLAYING;
            ST_PLAYING:
                // this asserts done if the offset exceeds the sample count of the file
                if (audio_rom_offset >= selected_sound_samples_count)
                    next_state = ST_DONE;
                else if (start)
                    next_state = ST_REPLAYING;
            ST_REPLAYING:
                next_state = ST_PLAYING;
            ST_DONE:
                if (!start) next_state = ST_IDLE;
            default:
                next_state = ST_IDLE;
        endcase
   end

    // assume next state
    always @(posedge clock) begin
        if (reset)
            current_state <= ST_IDLE;
        else
            current_state <= next_state;
    end

    wire state_change_due = (current_state != next_state);

    always @(posedge clock) begin

        if (reset) begin
            selected_sound_id <= 8'hFF;
            selected_sound_base_address <= 24'b0;
            end

        // when entering the PLAYING state, register value of selected
        // sound id and the total number of samples in the selected file
        if (state_change_due && next_state == ST_PLAYING) begin
            selected_sound_id <= sound_select;
            selected_sound_samples_count <= samples_count[sound_select];
            selected_sound_base_address <= { 1'b0, sound_select, 15'b0 };
        end
```

```
        if (current_state == ST_PLAYING && ready) begin
            if (oversampling_counter < 2) begin
                oversampling_counter <= oversampling_counter + 1;
            end else begin
                oversampling_counter <= 0;
            end

            to_ac97_data <= audio_sample;

        end else begin
            if (current_state != ST_PLAYING)
                audio_rom_offset <= 0;
        end
    end

endmodule
// end of file rtl/audio/audio_subsystem.v


// start of file rtl/audio/ac97_interface.v
////////////////////////////////////////////////////////////////////////////////
// Bi-directional monaural interface to AC97
////////////////////////////////////////////////////////////////////////////////

module ac97_monaural_interface (
  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,
  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_serial_to_parallel ac97_serial_to_parallel(
        .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
    ac97_init 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_serial_to_parallel (
  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 ac97_init (
  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 // ac97_init
// end of file rtl/audio/ac97_interface.v


// start of file rtl/audio/test_fsm.v
`define STATUS_RESET             4'h0
`define STATUS_READ_ID           4'h1
`define STATUS_CLEAR_LOCKS       4'h2
`define STATUS_ERASING           4'h3
`define STATUS_WRITING           4'h4
`define STATUS_READING           4'h5
`define STATUS_SUCCESS           4'h6
`define STATUS_BAD_MANUFACTURER  4'h7
`define STATUS_BAD_SIZE          4'h8
`define STATUS_LOCK_BIT_ERROR    4'h9
`define STATUS_ERASE_BLOCK_ERROR 4'hA
`define STATUS_WRITE_ERROR       4'hB
`define STATUS_READ_WRONG_DATA   4'hC

`define NUM_BLOCKS 128
`define BLOCK_SIZE 64*1024
`define LAST_BLOCK_ADDRESS ((`NUM_BLOCKS-1)*`BLOCK_SIZE)
`define LAST_ADDRESS (`NUM_BLOCKS*`BLOCK_SIZE-1)

`define FLASHOP_IDLE  2'b00
`define FLASHOP_READ  2'b01
`define FLASHOP_WRITE 2'b10
```

```
module test_fsm (reset, clock, fop, faddress, fwdata, frdata, fbusy, dots, mode, busy, datain, addrin,
state);
   input reset, clock;
   output [1:0] fop;
   output [22:0] faddress;
   output [15:0] fwdata;
   input [15:0]  frdata;
   input fbusy;
   output [639:0] dots;
   input [1:0] mode;
   output busy;
   input [15:0] datain;
   input [22:0] addrin;
   output state;

   reg [1:0] fop;
   reg [22:0] faddress;
   reg [15:0] fwdata;
   reg [639:0] dots;
   reg busy;
   reg [15:0] data_to_store;

   ////////////////////////////////////////////////////////////////////////////
   // State Machine
   ////////////////////////////////////////////////////////////////////////////

   reg [7:0] state;
   reg [3:0] status;

   parameter MODE_IDLE  = 0;
   parameter MODE_INIT  = 1;
   parameter MODE_WRITE = 2;
   parameter MODE_READ  = 3;

   parameter MAX_ADDRESS = 23'h030000;

   parameter HOME = 8'h12;


   always @(posedge clock)
     if (reset)
       begin
                        state <= HOME;
                        status <= `STATUS_RESET;
                        faddress <= 0;
                        fop <= `FLASHOP_IDLE;
                        busy <= 1;
       end
     else if (!fbusy && (fop == `FLASHOP_IDLE))
       case (state)

         HOME://12
                case(mode)
                        MODE_INIT: begin
                                state <= 8'h00;
                                busy <= 1;
                        end

                        MODE_WRITE: begin
                                state <= 8'h0C;
                                busy <= 1;
                        end

                        MODE_READ: begin
                                busy <= 1;
                                if(status == `STATUS_READING)
                                        state <= 8'h11;
                                else
                                        state <= 8'h10;
                        end

                        default: begin
                                state <= HOME;
```

```
                        busy <= 0;
                end
        endcase

////////////////////////////////////////////////////////////////////////
// Wipe It
////////////////////////////////////////////////////////////////////////
8'h00:
        begin
                // Issue "read id codes" command
                status <= `STATUS_READ_ID;
                faddress <= 0;
                fwdata <= 16'h0090;
                fop <= `FLASHOP_WRITE;
                state <= state+1;
        end

8'h01:
  begin
     // Read manufacturer code
     faddress <= 0;
     fop <= `FLASHOP_READ;
     state <= state+1;
  end

8'h02:
  if (frdata != 16'h0089) // 16'h0089 = Intel
    status <= `STATUS_BAD_MANUFACTURER;
  else
    begin
       // Read the device size code
       faddress <= 1;
       fop <= `FLASHOP_READ;
       state <= state+1;
    end

8'h03:
  if (frdata != 16'h0018) // 16'h0018 = 128Mbit
    status <= `STATUS_BAD_SIZE;
  else
    begin
       faddress <= 0;
       fwdata <= 16'hFF;
       fop <= `FLASHOP_WRITE;
       state <= state+1;
    end

8'h04:
  begin
     // Issue "clear lock bits" command
     status <= `STATUS_CLEAR_LOCKS;
     faddress <= 0;
     fwdata <= 16'h60;
     fop <= `FLASHOP_WRITE;
     state <= state+1;
  end

8'h05:
  begin
     // Issue "confirm clear lock bits" command
     faddress <= 0;
     fwdata <= 16'hD0;
     fop <= `FLASHOP_WRITE;
     state <= state+1;
  end

8'h06:
  begin
     // Read status
     faddress <= 0;
     fop <= `FLASHOP_READ;
     state <= state+1;
  end
```

```
8'h07:
  if (frdata[7] == 1) // Done clearing lock bits
    if (frdata[6:1] == 0) // No errors
      begin
         faddress <= 0;
         fop <= `FLASHOP_IDLE;
         state <= state+1;
      end
    else
      status <= `STATUS_LOCK_BIT_ERROR;
  else // Still busy, reread status register
    begin
       faddress <= 0;
       fop <= `FLASHOP_READ;
    end

////////////////////////////////////////////////////////////////////////
// Block Erase Sequence
////////////////////////////////////////////////////////////////////////
8'h08:
  begin
     status <= `STATUS_ERASING;
     fwdata <= 16'h20; // Issue "erase block" command
     fop <= `FLASHOP_WRITE;
     state <= state+1;
  end

8'h09:
  begin
     fwdata <= 16'hD0; // Issue "confirm erase" command
     fop <= `FLASHOP_WRITE;
     state <= state+1;
  end
8'h0A:
  begin
     fop <= `FLASHOP_READ;
     state <= state+1;
  end
8'h0B:
  if (frdata[7] == 1) // Done erasing block
    if (frdata[6:1] == 0) // No errors
      if (faddress != MAX_ADDRESS) // `LAST_BLOCK_ADDRESS)
        begin
           faddress <= faddress+`BLOCK_SIZE;
           fop <= `FLASHOP_IDLE;
           state <= state-3;
        end
      else
        begin
           faddress <= 0;
           fop <= `FLASHOP_IDLE;
           state <= HOME;              //done erasing, go home
        end
    else // Erase error detected
      status <= `STATUS_ERASE_BLOCK_ERROR;
  else // Still busy
    fop <= `FLASHOP_READ;

////////////////////////////////////////////////////////////////////////
// Write Mode
////////////////////////////////////////////////////////////////////////
8'h0C:
       begin
               data_to_store <= datain;
               status <= `STATUS_WRITING;
               fwdata <= 16'h40; // Issue "setup write" command
               fop <= `FLASHOP_WRITE;
               state <= state+1;
       end

8'h0D:
  begin
```

```
            fwdata <= data_to_store; // Finish write
            fop <= `FLASHOP_WRITE;
            state <= state+1;
          end
        8'h0E:
          begin
            // Read status register
            fop <= `FLASHOP_READ;
            state <= state+1;
          end
        8'h0F:
          if (frdata[7] == 1) // Done writing
            if (frdata[6:1] == 0) // No errors
              if (faddress != 23'h7FFFFF) // `LAST_ADDRESS)
                            begin
                                    faddress <= faddress+1;
                                    fop <= `FLASHOP_IDLE;
                                    state <= HOME;
                            end
                        else
                                    status <= `STATUS_WRITE_ERROR;
            else // Write error detected
              status <= `STATUS_WRITE_ERROR;
          else // Still busy
            fop <= `FLASHOP_READ;

        ////////////////////////////////////////////////////////////////////
        // Read Mode INIT
        ////////////////////////////////////////////////////////////////////
        8'h10:
          begin
            status <= `STATUS_READING;
            fwdata <= 16'hFF; // Issue "read array" command
            fop <= `FLASHOP_WRITE;
            faddress <= 0;
            state <= state+1;
          end

        ////////////////////////////////////////////////////////////////////
        // Read Mode
        ////////////////////////////////////////////////////////////////////
        8'h11:
          begin
                faddress <= addrin;
            fop <= `FLASHOP_READ;
            state <= HOME;
          end

       default:
                begin
                        status <= `STATUS_BAD_MANUFACTURER;
                        faddress <= 0;
                        state <= HOME;
                end

       endcase
     else
       fop <= `FLASHOP_IDLE;

function [39:0] nib2char;
   input [3:0] nib;
   begin
      case (nib)
        4'h0: nib2char = 40'b00111110_01010001_01001001_01000101_00111110;
        4'h1: nib2char = 40'b00000000_01000010_01111111_01000000_00000000;
        4'h2: nib2char = 40'b01100010_01010001_01001001_01001001_01000110;
        4'h3: nib2char = 40'b00100010_01000001_01001001_01001001_00110110;
        4'h4: nib2char = 40'b00011000_00010100_00010010_01111111_00010000;
        4'h5: nib2char = 40'b00100111_01000101_01000101_01000101_00111001;
        4'h6: nib2char = 40'b00111100_01001010_01001001_01001001_00110000;
        4'h7: nib2char = 40'b00000001_01110001_00001001_00000101_00000011;
        4'h8: nib2char = 40'b00110110_01001001_01001001_01001001_00110110;
        4'h9: nib2char = 40'b00000110_01001001_01001001_00101001_00011110;
```

```
        4'hA: nib2char = 40'b01111110_00001001_00001001_00001001_01111110;
        4'hB: nib2char = 40'b01111111_01001001_01001001_01001001_00110110;
        4'hC: nib2char = 40'b00111110_01000001_01000001_01000001_00100010;
        4'hD: nib2char = 40'b01111111_01000001_01000001_01000001_00111110;
        4'hE: nib2char = 40'b01111111_01001001_01001001_01001001_01000001;
        4'hF: nib2char = 40'b01111111_00001001_00001001_00001001_00000001;
      endcase
   end
endfunction

wire [159:0] data_dots;
assign data_dots = {nib2char(frdata[15:12]), nib2char(frdata[11:8]),
                    nib2char(frdata[7:4]), nib2char(frdata[3:0])};

wire [239:0] address_dots;
assign address_dots = {nib2char({ 1'b0, faddress[22:20]}),
                       nib2char(faddress[19:16]),
                       nib2char(faddress[15:12]),
                       nib2char(faddress[11:8]),
                       nib2char(faddress[7:4]),
                       nib2char(faddress[3:0])};

always @(status or address_dots or data_dots)
  case (status)
    `STATUS_RESET:
      dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
               40'b01111111_01001001_01001001_01001001_01000001, // E
               40'b00100110_01001001_01001001_01001001_00110010, // S
               40'b01111111_01001001_01001001_01001001_01000001, // E
               40'b00000001_00000001_01111111_00000001_00000001, // T
               40'b00000000_00000000_00000000_00000000_00000000, //
               40'b00000000_00000000_00000000_00000000_00000000, //
               40'b00000000_00000000_00000000_00000000_00000000, //
               40'b00000000_00000000_00000000_00000000_00000000, //
               40'b00000000_00000000_00000000_00000000_00000000, //
               40'b00001000_00001000_00001000_00001000_00001000, // -
               40'b00001000_00001000_00001000_00001000_00001000, // -
               40'b00001000_00001000_00001000_00001000_00001000, // -
               40'b00001000_00001000_00001000_00001000_00001000, // -
               40'b00001000_00001000_00001000_00001000_00001000, // -
               40'b00001000_00001000_00001000_00001000_00001000};// -
    `STATUS_READ_ID:
      dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
               40'b01111111_01001001_01001001_01001001_01000001, // E
               40'b01111110_00001001_00001001_00001001_01111110, // A
               40'b01111111_01000001_01000001_01000001_00111110, // D
               40'b00000000_00000000_00000000_00000000_00000000, //
               40'b00000000_01000001_01111111_01000001_00000000, // I
               40'b01111111_01000001_01000001_01000001_00111110, // D
               40'b00000000_00000000_00000000_00000000_00000000, //
               40'b00000000_00000000_00000000_00000000_00000000, //
               40'b00000000_00000000_00000000_00000000_00000000, //
               address_dots};
    `STATUS_CLEAR_LOCKS:
      dots <= {40'b00111110_01000001_01000001_01000001_00100010, // C
               40'b01111111_01000000_01000000_01000000_01000000, // L
               40'b01111111_00001001_00011001_00101001_01000110, // R
               40'b00000000_00000000_00000000_00000000_00000000, //
               40'b01111111_01000000_01000000_01000000_01000000, // L
               40'b00111110_01000001_01000001_01000001_00111110, // O
               40'b00111110_01000001_01000001_01000001_00100010, // C
               40'b01111111_00001000_00010100_00100010_01000001, // K
               40'b00100110_01001001_01001001_01001001_00110010, // S
               40'b00000000_00000000_00000000_00000000_00000000, //
               address_dots};
    `STATUS_ERASING:
      dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
               40'b01111111_00001001_00011001_00101001_01000110, // R
               40'b01111110_00001001_00001001_00001001_01111110, // A
               40'b00100110_01001001_01001001_01001001_00110010, // S
               40'b00000000_01000001_01111111_01000001_00000000, // I
               40'b01111111_00000010_00000100_00001000_01111111, // N
               40'b00111110_01000001_01001001_01001001_00111010, // G
```

```
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          address_dots};
`STATUS_WRITING:
  dots <= {40'b01111111_00100000_00011000_00100000_01111111, // W
          40'b01111111_00001001_00011001_00101001_01000110, // R
          40'b00000000_01000001_01111111_01000001_00000000, // I
          40'b00000001_00000001_01111111_00000001_00000001, // T
          40'b00000000_01000001_01111111_01000001_00000000, // I
          40'b01111111_00000010_00000100_00001000_01111111, // N
          40'b00111110_01000001_01001001_01001001_00111010, // G
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          address_dots};
`STATUS_READING:
  dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
          40'b01111111_01001001_01001001_01001001_01000001, // E
          40'b01111110_00001001_00001001_00001001_01111110, // A
          40'b01111111_01000001_01000001_01000001_00111110, // D
          40'b00000000_01000001_01111111_01000001_00000000, // I
          40'b01111111_00000010_00000100_00001000_01111111, // N
          40'b00111110_01000001_01001001_01001001_00111010, // G
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          address_dots};
`STATUS_SUCCESS:
  dots <= {40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00101010_00011100_01111111_00011100_00101010, // *
          40'b00101010_00011100_01111111_00011100_00101010, // *
          40'b00101010_00011100_01111111_00011100_00101010, // *
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b01111111_00001001_00001001_00001001_00000110, // P
          40'b01111110_00001001_00001001_00001001_01111110, // A
          40'b00100110_01001001_01001001_01001001_00110010, // S
          40'b00100110_01001001_01001001_01001001_00110010, // S
          40'b01111111_01001001_01001001_01001001_01000001, // E
          40'b01111111_01000001_01000001_01000001_00111110, // D
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00101010_00011100_01111111_00011100_00101010, // *
          40'b00101010_00011100_01111111_00011100_00101010, // *
          40'b00101010_00011100_01111111_00011100_00101010, // *
          40'b00000000_00000000_00000000_00000000_00000000};//
`STATUS_BAD_MANUFACTURER:
  dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
          40'b01111111_00001001_00011001_00101001_01000110, // R
          40'b01111111_00001001_00011001_00101001_01000110, // R
          40'b00000000_00110110_00110110_00000000_00000000, // :
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b01111111_00000010_00001100_00000010_01111111, // M
          40'b01111110_00001001_00001001_00001001_01111110, // A
          40'b01111111_00000010_00000100_00001000_01111111, // N
          40'b01111111_00001001_00001001_00001001_00000001, // U
          40'b01111111_00001001_00001001_00001001_00000001, // F
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          data_dots};
`STATUS_BAD_SIZE:
  dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
          40'b01111111_00001001_00011001_00101001_01000110, // R
          40'b01111111_00001001_00011001_00101001_01000110, // R
          40'b00000000_00110110_00110110_00000000_00000000, // :
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00100110_01001001_01001001_01001001_00110010, // S
          40'b00000000_01000001_01111111_01000001_00000000, // I
          40'b01100001_01010001_01001001_01000101_01000011, // Z
          40'b01111111_01001001_01001001_01001001_01000001, // E
          40'b00000000_00000000_00000000_00000000_00000000,
          40'b00000000_00000000_00000000_00000000_00000000,
          40'b00000000_00000000_00000000_00000000_00000000,
          data_dots};
```

```
      `STATUS_LOCK_BIT_ERROR:
        dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
                 40'b01111111_00001001_00011001_00101001_01000110, // R
                 40'b01111111_00001001_00011001_00101001_01000110, // R
                 40'b00000000_00110110_00110110_00000000_00000000, // :
                 40'b00000000_00000000_00000000_00000000_00000000, //
                 40'b01111111_01000000_01000000_01000000_01000000, // L
                 40'b00111110_01000001_01000001_01000001_00111110, // O
                 40'b00111110_01000001_01000001_01000001_00100010, // C
                 40'b01111111_00001000_00010100_00100010_01000001, // K
                 40'b00100110_01001001_01001001_01001001_00110010, // S
                 40'b00000000_00000000_00000000_00000000_00000000,
                 40'b00000000_00000000_00000000_00000000_00000000,
                 data_dots};
      `STATUS_ERASE_BLOCK_ERROR:
        dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
                 40'b01111111_00001001_00011001_00101001_01000110, // R
                 40'b01111111_00001001_00011001_00101001_01000110, // R
                 40'b00000000_00110110_00110110_00000000_00000000, // :
                 40'b00000000_00000000_00000000_00000000_00000000, //
                 40'b01111111_01001001_01001001_01001001_01000001, // E
                 40'b01111111_00001001_00011001_00101001_01000110, // R
                 40'b01111110_00001001_00001001_00001001_01111110, // A
                 40'b00100110_01001001_01001001_01001001_00110010, // S
                 40'b01111111_01001001_01001001_01001001_01000001, // E
                 40'b00000000_00000000_00000000_00000000_00000000,
                 40'b00000000_00000000_00000000_00000000_00000000,
                 data_dots};
      `STATUS_WRITE_ERROR:
        dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
                 40'b01111111_00001001_00011001_00101001_01000110, // R
                 40'b01111111_00001001_00011001_00101001_01000110, // R
                 40'b00000000_00110110_00110110_00000000_00000000, // :
                 40'b00000000_00000000_00000000_00000000_00000000, //
                 40'b01111111_00100000_00011000_00100000_01111111, // W
                 40'b01111111_00001001_00011001_00101001_01000110, // R
                 40'b00000000_01000001_01111111_01000001_00000000, // I
                 40'b00000001_00000001_01111111_00000001_00000001, // T
                 40'b01111111_01001001_01001001_01001001_01000001, // E
                 40'b00000000_00000000_00000000_00000000_00000000,
                 40'b00000000_00000000_00000000_00000000_00000000,
                 data_dots};
      `STATUS_READ_WRONG_DATA:
        dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
                 40'b01111111_00001001_00011001_00101001_01000110, // R
                 40'b01111111_00001001_00011001_00101001_01000110, // R
                 40'b00000000_00110110_00110110_00000000_00000000, // :
                 40'b00000000_00000000_00000000_00000000_00000000,
                 address_dots,
                 40'b00000000_00000000_00000000_00000000_00000000,
                 data_dots};
        default:
        dots <= {16{40'b01010101_00101010_01010101_00101010_01010101}};
    endcase

endmodule
// end of file rtl/audio/test_fsm.v


// start of file rtl/audio/flash_int.v
module flash_int(reset, clock, op, address, wdata, rdata, busy, flash_data,
                 flash_address, flash_ce_b, flash_oe_b, flash_we_b,
                 flash_reset_b, flash_sts, flash_byte_b);

   parameter access_cycles = 5;
   parameter reset_assert_cycles = 1000;
   parameter reset_recovery_cycles = 30;

   input reset, clock; // Reset and clock for the flash interface
   input [1:0] op; // Flash operation select (read, write, idle)
   input [22:0] address;
   input [15:0] wdata;
   output [15:0] rdata;
```

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

reg [1:0] lop;
reg [15:0] rdata;
reg busy;
reg [15:0] flash_wdata;
reg flash_ddata;
reg [23:0] flash_address;
reg flash_oe_b, flash_we_b, flash_reset_b;

assign flash_ce_b = flash_oe_b && flash_we_b;
assign flash_byte_b = 1; // 1 = 16-bit mode (A0 ignored)

assign flash_data = flash_ddata ? flash_wdata : 16'hZ;

initial
  flash_reset_b <= 1'b1;

reg [9:0] state;

always @(posedge clock)
  if (reset)
    begin
                    state <= 0;
                    flash_reset_b <= 0;
                    flash_we_b <= 1;
                    flash_oe_b <= 1;
                    flash_ddata <= 0;
                    busy <= 1;
    end
  else if (flash_reset_b == 0)
    if (state == reset_assert_cycles)
                    begin
                            flash_reset_b <= 1;
                            state <= 1023-reset_recovery_cycles;
                    end
    else
                    state <= state+1;
      else if ((state == 0) && !busy)
    // The flash chip and this state machine are both idle. Latch the user's
    // address and write data inputs. Deassert OE and WE, and stop driving
    // the data buss ourselves. If a flash operation (read or write) is
    // requested, move to the next state.
    begin
                    flash_address <= {address, 1'b0};
                    flash_we_b <= 1;
                    flash_oe_b <= 1;
                    flash_ddata <= 0;
                    flash_wdata <= wdata;
                    lop <= op;
                    if (op != `FLASHOP_IDLE)
                            begin
                                    busy <= 1;
                                    state <= state+1;
                            end
                    else
                            busy <= 0;
                    end
            else if ((state==0) && flash_sts)
                    busy <= 0;
            else if (state == 1)
    // The first stage of a flash operation. The address bus is already set,
    // so, if this is a read, we assert OE. For a write, we start driving
    // the user's data onto the flash databus (the value was latched in the
    // previous state.
                    begin
                            if (lop == `FLASHOP_WRITE)
                                    flash_ddata <= 1;
```

```
                                else if (lop == `FLASHOP_READ)
                                        flash_oe_b <= 0;
                                state <= state+1;
        end
      else if (state == 2)
        // The second stage of a flash operation. Nothing to do for a read. For
        // a write, we assert WE.
        begin
                        if (lop == `FLASHOP_WRITE)
                                flash_we_b <= 0;
                        state <= state+1;
        end
      else if (state == access_cycles+1)
        // The third stage of a flash operation. For a read, we latch the data
        // from the flash chip. For a write, we deassert WE.
        begin
                        if (lop == `FLASHOP_WRITE)
                                flash_we_b <= 1;
                        if (lop == `FLASHOP_READ)
                                rdata <= flash_data;
                        state <= 0;
        end
      else
        begin
                        if (!flash_sts)
                                busy <= 1;
                        state <= state+1;
        end

endmodule
// end of file rtl/audio/flash_int.v


// start of file rtl/audio/flash_manager.v
//manages all the stuff needed to read and write to the flash ROM
module flash_manager(clock, reset, dots, writemode, wdata, dowrite, raddr, frdata, doread, busy, flash_data,
flash_address, flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_sts, flash_byte_b, fsmstate);
        input reset, clock;             //clock and reset
        output [639:0] dots;            //outputs to dot-matrix to help debug flash, not necessary
        input writemode;                //if true then we're in write mode, else we're in read mode
        input [15:0] wdata;             //data to be written
        input dowrite;                  //putting this high tells the manager the data it has is new, write
it
        input [22:0] raddr;             //address to read from
        output[15:0] frdata;            //data being read
        reg[15:0]    rdata;
        input doread;                    //putting this high tells the manager to perform a read on the
current address
        output busy;                     //and an output to tell folks we're still working on the last thing
        reg busy;

        inout [15:0] flash_data;                                        //direct passthrough from labkit to
low-level modules (flash_int and test_fsm)
        output [23:0] flash_address;
        output flash_ce_b, flash_oe_b, flash_we_b;
        output flash_reset_b, flash_byte_b;
        input  flash_sts;

        wire flash_busy;                //except these, which are internal to the interface
        wire[15:0] fwdata;
        wire[15:0] frdata;
        wire[22:0] address;
        wire [1:0] op;

        reg [1:0] mode;
        wire fsm_busy;

        reg[2:0] state;                                 //210

        output[11:0] fsmstate;
        wire [7:0] fsmstateinv;
        assign fsmstate = {state,flash_busy,fsm_busy,fsmstateinv[4:0],mode};    //for debugging only
```

```
                                                                        //this guy takes care of
/some/ of flash's tantrums
        flash_int flash(reset, clock, op, address, fwdata, frdata, flash_busy, flash_data, flash_address,
flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_sts, flash_byte_b);
                                                                        //and this guy takes care of
the rest of its tantrums
        test_fsm  fsm  (reset, clock, op, address, fwdata, frdata, flash_busy, dots, mode, fsm_busy, wdata,
raddr, fsmstateinv);

        parameter MODE_IDLE     = 0;
        parameter MODE_INIT     = 1;
        parameter MODE_WRITE = 2;
        parameter MODE_READ     = 3;

        parameter HOME          = 3'd0;
        parameter MEM_INIT      = 3'd1;
        parameter MEM_WAIT      = 3'd2;
        parameter WRITE_READY= 3'd3;
        parameter WRITE_WAIT    = 3'd4;
        parameter READ_READY    = 3'd5;
        parameter READ_WAIT     = 3'd6;

        always @ (posedge clock)
                if(reset)
                        begin
                                busy <= 1;
                                state <= HOME;
                                mode <= MODE_IDLE;
                        end
                else begin
                        case(state)
                                HOME://0                                //we always start here
                                        if(!fsm_busy)
                                                begin
                                                        busy <= 0;
                                                        if(writemode)
                                                                begin
                                                                        busy <= 1;
                                                                        state <= MEM_INIT;
                                                                end
                                                        else
                                                                begin
                                                                        busy <= 1;
                                                                        state <= READ_READY;
                                                                end
                                                end
                                        else
                                                mode <= MODE_IDLE;

                                MEM_INIT://1                                                    //begin
wiping the memory
                                        begin
                                                busy <= 1;
                                                mode <= MODE_INIT;
                                                if(fsm_busy)                                    //to give the
fsm a chance to raise its busy signal
                                                        state <= MEM_WAIT;
                                        end

                                MEM_WAIT://2                                            //finished wiping
                                        if(!fsm_busy)
                                                begin
                                                        busy <= 0;
                                                        state<= WRITE_READY;
                                                end
                                        else
                                                mode <= MODE_IDLE;

                                WRITE_READY://3                                 //waiting for data to write
to flash
                                        if(dowrite)
                                                begin
                                                        busy <= 1;
```

```
                                        mode <= MODE_WRITE;
                                end
                        else if(busy)
                                state <= WRITE_WAIT;
                        else if(!writemode)
                                state <= READ_READY;

                WRITE_WAIT://4                          //waiting for flash to finish writing
                        if(!fsm_busy)
                                begin
                                        busy <= 0;
                                        state <= WRITE_READY;
                                end
                        else
                                mode <= MODE_IDLE;

                READ_READY://5                          //ready to read data
                        if(doread)
                                begin
                                        busy <= 1;
                                        mode <= MODE_READ;
                                        if(busy)                        //lets the fsm raise
its busy level
                                                state <= READ_WAIT;
                                end
                        else
                                busy <= 0;

                READ_WAIT://6                   //waiting for flash to give the data up
                        if(!fsm_busy)
                                begin
                                        busy <= 0;
                                        state <= READ_READY;
                                end
                        else
                                mode <= MODE_IDLE;

                default: begin          //should never happen...
                        state <= 3'd7;
                end
        endcase
    end
endmodule
// end of file rtl/audio/flash_manager.v


// start of file rtl/video_processing/rgb_to_hsv.v
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Engineer: Kevin Zheng Class of 2012
//////////////////////////////////////////////////////////////////////////////////
module rgb_to_hsv(clock, reset, r, g, b, h, s, v
//, lightness_correction_enabled, x, y    // [ricardo]
);
    input wire clock;
    input wire reset;
    input wire [7:0] r;
    input wire [7:0] g;
    input wire [7:0] b;
    output reg [7:0] h;
    output reg [7:0] s;
    output reg [7:0] v;

    reg [7:0] my_r_delay1, my_g_delay1, my_b_delay1;
    reg [7:0] my_r_delay2, my_g_delay2, my_b_delay2;
    reg [7:0] my_r, my_g, my_b;
    reg [7:0] min, max, delta;
    reg [15:0] s_top;
    reg [15:0] s_bottom;
    reg [15:0] h_top;
    reg [15:0] h_bottom;
    wire [15:0] s_quotient;
    wire [15:0] s_remainder;
```

```
    wire s_rfd;
    wire [15:0] h_quotient;
    wire [15:0] h_remainder;
    wire h_rfd;
    reg [7:0] v_delay [19:0];
    reg [18:0] h_negative;
    reg [15:0] h_add [18:0];
    reg [4:0] i;

    // Clocks 4-18: perform all the divisions
    //the s_divider (16/16) has delay 18
    //the hue_div (16/16) has delay 18

    divider_16by16 hue_div1(
        .clk(clock),
        .dividend(s_top),
        .divisor(s_bottom),
        .quot(s_quotient),
        .remd(s_remainder),
        .rfd(s_rfd)
    );

    divider_16by16 hue_div2(
        .clk(clock),
        .dividend(h_top),
        .divisor(h_bottom),
        .quot(h_quotient),
        .remd(h_remainder),
        .rfd(h_rfd)
    );

    `include "pixel_functions.v"

    always @ (posedge clock) begin

        // Clock 1: latch the inputs (always positive)
        {my_r, my_g, my_b} <= {r, g, b};

        // Clock 2: compute min, max
        {my_r_delay1, my_g_delay1, my_b_delay1} <= {my_r, my_g, my_b};

        if((my_r >= my_g) && (my_r >= my_b)) //(B,S,S)
            max <= my_r;
        else if((my_g >= my_r) && (my_g >= my_b)) //(S,B,S)
            max <= my_g;
        else    max <= my_b;

        if((my_r <= my_g) && (my_r <= my_b)) //(S,B,B)
            min <= my_r;
        else if((my_g <= my_r) && (my_g <= my_b)) //(B,S,B)
            min <= my_g;
        else
            min <= my_b;

        // Clock 3: compute the delta
        {my_r_delay2, my_g_delay2, my_b_delay2} <= {my_r_delay1, my_g_delay1, my_b_delay1};
        v_delay[0] <= max;
        delta <= max - min;

        // Clock 4: compute the top and bottom of whatever divisions we need to do
        s_top <= 8'd255 * delta;
        s_bottom <= (v_delay[0]>0)?{8'd0, v_delay[0]}: 16'd1;


        if(my_r_delay2 == v_delay[0]) begin
            h_top <= (my_g_delay2 >= my_b_delay2)?(my_g_delay2 - my_b_delay2) * 8'd255:(my_b_delay2 -
my_g_delay2) * 8'd255;
            h_negative[0] <= (my_g_delay2 >= my_b_delay2)?0:1;
            h_add[0] <= 16'd0;
        end
        else if(my_g_delay2 == v_delay[0]) begin
            h_top <= (my_b_delay2 >= my_r_delay2)?(my_b_delay2 - my_r_delay2) * 8'd255:(my_r_delay2 -
my_b_delay2) * 8'd255;
```

```
            h_negative[0] <= (my_b_delay2 >= my_r_delay2)?0:1;
            h_add[0] <= 16'd85;
        end
        else if(my_b_delay2 == v_delay[0]) begin
            h_top <= (my_r_delay2 >= my_g_delay2)?(my_r_delay2 - my_g_delay2) * 8'd255:(my_g_delay2 -
my_r_delay2) * 8'd255;
            h_negative[0] <= (my_r_delay2 >= my_g_delay2)?0:1;
            h_add[0] <= 16'd170;
        end

        h_bottom <= (delta > 0)?delta * 8'd6:16'd6;


        //delay the v and h_negative signals 18 times
        for(i=1; i<19; i=i+1) begin
            v_delay[i] <= v_delay[i-1];
            h_negative[i] <= h_negative[i-1];
            h_add[i] <= h_add[i-1];
        end

        v_delay[19] <= v_delay[18];

        //Clock 22: compute the final value of h
        //depending on the value of h_delay[18], we need to subtract 255 from it to make it come back around
the circle

        if (h_negative[18] && (h_quotient > h_add[18])) begin
            h <= 8'd255 - h_quotient[7:0] + h_add[18];
        end
        else if(h_negative[18]) begin
            h <= h_add[18] - h_quotient[7:0];
        end
        else begin
            h <= h_quotient[7:0] + h_add[18];
        end

        //pass out s and v straight
        s <= s_quotient;
        v <= v_delay[19];
    end
endmodule
// end of file rtl/video_processing/rgb_to_hsv.v


// start of file rtl/video_processing/center_of_mass.v
module center_of_mass(
    input clock,
    input reset,
    input [10:0] x,
    input [9:0] y,
    input [23:0] hsv,
    output reg [9:0] right_hand_x,
    output reg [9:0] right_hand_y,
    output reg [9:0] left_hand_x,
    output reg [9:0] left_hand_y,
    output reg right_hand_present,
    output reg [9:0] left_hand_present,
    inout [31:0] logic_analyzer_bus,
    output reg [25:0] x_dividend,
    output reg [17:0] red_count_out,
    output reg [25:0] x_divisor,
    // true when the current pixel at (x,y) was considered red
    output reg current_pixel_is_red,
    output reg current_pixel_is_green
);

    `include "pixel_functions.v"

    reg [17:0] red_count;       // number of red pixels
    reg [17:0] green_count;     // number of green pixels
    reg [25:0] red_x_accum;     // sum of x coordinates of all red pixels
    reg [25:0] green_x_accum;   // sum of x coordinates of all green pixels
    reg [25:0] red_y_accum;     // sum of y coordinates of all red pixels
```

```
reg [25:0] green_y_accum;   // sum of y coordinates of all green pixels

reg [25:0] red_divisor;
reg [25:0] red_x_dividend;
reg [25:0] red_y_dividend;

reg [25:0] green_divisor;
reg [25:0] green_x_dividend;
reg [25:0] green_y_dividend;

wire [25:0] quotient_x_rh;
wire [17:0] remainder_x_rh;
wire rfd_x_rh;

wire [25:0] quotient_y_rh;
wire [17:0] remainder_y_rh;
wire rfd_y_rh;

wire [25:0] quotient_x_lh;
wire [17:0] remainder_x_lh;
wire rfd_x_lh;

wire [25:0] quotient_y_lh;
wire [17:0] remainder_y_lh;
wire rfd_y_lh;

divider_26by26 x_righthand(
    .clk(clock),
    .dividend(red_x_dividend),
    .divisor(red_divisor),
    .quot(quotient_x_rh),
    .remd(remainder_x_rh),
    .rfd(rfd_x_rh)
);

divider_26by26 y_righthand(
    .clk(clock),
    .dividend(red_y_dividend),
    .divisor(red_divisor),
    .quot(quotient_y_rh),
    .remd(remainder_y_rh),
    .rfd(rfd_y_rh)
);

divider_26by26 x_lefthand(
    .clk(clock),
    .dividend(green_x_dividend),
    .divisor(green_divisor),
    .quot(quotient_x_lh),
    .remd(remainder_x_lh),
    .rfd(rfd_x_lh)
);

divider_26by26 y_lefthand(
    .clk(clock),
    .dividend(green_y_dividend),
    .divisor(green_divisor),
    .quot(quotient_y_lh),
    .remd(remainder_y_lh),
    .rfd(rfd_y_lh)
);

// shift register to remove red noise from the captured frame; pixel will
// be considered red only if 4 consecutive pixels are within the threshold
reg [4:0] red_pixel_history;
//reg [3:0] green_pixel_history;
reg [4:0] green_pixel_history;
always @(posedge clock) begin

    if (x == 0)
        red_pixel_history <= 5'b00000;
    else
        red_pixel_history <= {
```

```
            red_pixel_history[3:0],
            (pixel_is_visible(x, y) && pixel_is_red(hsv))
        };
    current_pixel_is_red <= (red_pixel_history == 5'b11111);

    if (x == 0)
        green_pixel_history <= 5'b0000;
    else
        green_pixel_history <= {
            green_pixel_history[3:0],
            (pixel_is_visible(x, y) && pixel_is_green(hsv))
        };
    current_pixel_is_green <= (green_pixel_history == 5'b11111);
end


always @(posedge clock) begin
    // set initial values
    if (x == 0 && y == 0) begin
        red_count <= 0;
        red_x_accum <= 0;
        red_y_accum <= 0;

        green_count <= 0;
        green_x_accum <= 0;
        green_y_accum <= 0;

        red_count_out <= 0;
    end
    else if (current_pixel_is_red) begin
        red_count <= red_count + 1;
        red_x_accum <= red_x_accum + x;
        red_y_accum <= red_y_accum + y;
    end
    else if (current_pixel_is_green) begin
        green_count <= green_count + 1;
        green_x_accum <= green_x_accum + x;
        green_y_accum <= green_y_accum + y;
    end
    // division stage
    else if (x == 0 && y == 523 ) begin
        if (red_count >= MINIMUM_RED_PIXELS_COUNT_FOR_DETECTION) begin
            red_divisor <= red_count;
            red_x_dividend <= red_x_accum;
            red_y_dividend <= red_y_accum;
            right_hand_present <= 1;
        end else begin
            red_divisor <= 1;
            red_x_dividend <= 0;
            red_y_dividend <= 0;
            right_hand_present <= 0;
        end

        if (green_count >= MINIMUM_GREEN_PIXELS_COUNT_FOR_DETECTION) begin
            green_divisor <= green_count;
            green_x_dividend <= green_x_accum;
            green_y_dividend <= green_y_accum;
            left_hand_present <= 1;
        end else begin
            green_divisor <= 1;
            green_x_dividend <= 0;
            green_y_dividend <= 0;
            left_hand_present <= 0;
        end

        red_count_out <= red_count;
        x_dividend <= red_x_dividend;
        x_divisor <= red_divisor;
    end
end

// update outputs on rising edge of clock to prevent latches
always @(posedge clock) begin
```

```
        right_hand_x[9:0] = rfd_x_rh ? quotient_x_rh[9:0] : right_hand_x[9:0];
        right_hand_y[9:0] = rfd_y_rh ? quotient_y_rh[9:0] : right_hand_y[9:0];
        left_hand_x[9:0] = rfd_x_lh ? quotient_x_lh[9:0] : left_hand_x[9:0];
        left_hand_y[9:0] = rfd_y_lh ? quotient_y_lh[9:0] : left_hand_y[9:0];
    end

endmodule
// end of file rtl/video_processing/center_of_mass.v


// start of file rtl/video_processing/pixel_functions.v
`include "../common/global_definitions.v"

// true if pixel coordinates are within viewport limits
function pixel_is_visible(input [10:0] x, input [9:0] y);
    pixel_is_visible =
        (x >= CENTER_OF_MASS_X_MIN) && (x <= CENTER_OF_MASS_X_MAX) &&
        (y >= CENTER_OF_MASS_Y_MIN) && (y <= CENTER_OF_MASS_Y_MAX);
endfunction

// true if pixel color is within hsv limits for red
function pixel_is_red(input [23:0] hsv);
    reg [7:0] h, s, v;
    begin
        { h, s, v } = hsv;
        // (hsv[23:16] < 15 && hsv[15:8]>120 && hsv[7:0]>100)
        pixel_is_red =
            ((h <= 5) || (h >= 251)) &&
            (s >= 130) &&
            (v >= 100);
    end
endfunction

// true if pixel color is within hsv limits for green
function pixel_is_green(input [23:0] hsv);
    reg [7:0] h, s, v;
    begin
        { h, s, v } = hsv;
        pixel_is_green = (h > 48) && (h < 112) && (s > 64) && (s < 117) && (v > 30) && (v < 240);
    end
endfunction

// Calculate the euclidean distance ( sqrt[dx^2+dy^2] ) between two points,
// using an approximated function without square roots or multipliers.
// http://www.flipcode.com/archives/Fast_Approximate_Distance_Functions.shtml
function [19:0] euclidean_distance(input [10:0] x1, input [9:0] y1, input [10:0] x2, input [9:0] y2);
    reg [10:0] dx, dy;
    reg [10:0] longer_distance, shorter_distance;
    begin
        dx = (x1 > x2) ? (x1 - x2) : (x2 - x1);
        dy = (y1 > y2) ? (y1 - y2) : (y2 - y1);
        longer_distance = (dx > dy) ? dx : dy;
        shorter_distance = (dx > dy) ? dy : dx;

        euclidean_distance = (
            (
                  ( longer_distance << 8 )
                + ( longer_distance << 3 )
                - ( longer_distance << 4 )
                - ( longer_distance << 1 )
                + ( shorter_distance << 7 )
                - ( shorter_distance << 5 )
                + ( shorter_distance << 3 )
                - ( shorter_distance << 1 )
            ) >> 8
        );
    end
endfunction

// Correct the lightness of a pixel based on its distance from the center
// of the image. The NTSC camera tends to produce an image that is lighter
// around the center and darker on the edges.
function [7:0] adjust_lightness(input [10:0] x, input [9:0] y, input [7:0] lightness);
```

```
    reg [19:0] center_distance;
    reg [10:0] new_lightness;
    begin
        center_distance = euclidean_distance(x, y, 390, 300);
        new_lightness = lightness + (center_distance >> 3);
        adjust_lightness = (new_lightness > 255) ? 255 : new_lightness[7:0];
    end
endfunction


function point_is_in_crosshair(input [9:0] x1, input [9:0] y1, input [9:0] x2, input [9:0] y2);
    point_is_in_crosshair = (x1 == x2) || (y1 == y2);
endfunction
// end of file rtl/video_processing/pixel_functions.v


// start of file rtl/labkit/i2c_interface.v

// i2c module for use with the ADV7185

module i2c_interface(reset, clock4x, data, load, idle, ack, scl, sda);

   input reset;
   input clock4x;
   input [7:0] data;
   input load;
   output ack;
   output idle;
   output scl;
   output sda;

   reg [7:0] ldata;
   reg ack, idle;
   reg scl;
   reg sdai;

   reg [7:0] state;

   assign sda = sdai ? 1'bZ : 1'b0;

   always @(posedge clock4x)
     if (reset)
       begin
        state <= 0;
        ack <= 0;
       end
     else
       case (state)
     8'h00: // idle
       begin
          scl <= 1'b1;
          sdai <= 1'b1;
          ack <= 1'b0;
          idle <= 1'b1;
          if (load)
        begin
           ldata <= data;
           ack <= 1'b1;
           state <= state+1;
        end
       end
     8'h01: // Start
       begin
          ack <= 1'b0;
          idle <= 1'b0;
          sdai <= 1'b0;
          state <= state+1;
       end
     8'h02:
       begin
          scl <= 1'b0;
          state <= state+1;
       end
```

```
8'h03: // Send bit 7
  begin
     ack <= 1'b0;
     sdai <= ldata[7];
     state <= state+1;
  end
8'h04:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h05:
  begin
     state <= state+1;
  end
8'h06:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h07:
  begin
     sdai <= ldata[6];
     state <= state+1;
  end
8'h08:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h09:
  begin
     state <= state+1;
  end
8'h0A:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h0B:
  begin
     sdai <= ldata[5];
     state <= state+1;
  end
8'h0C:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h0D:
  begin
     state <= state+1;
  end
8'h0E:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h0F:
  begin
     sdai <= ldata[4];
     state <= state+1;
  end
8'h10:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h11:
  begin
     state <= state+1;
  end
8'h12:
```

```
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h13:
  begin
     sdai <= ldata[3];
     state <= state+1;
  end
8'h14:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h15:
  begin
     state <= state+1;
  end
8'h16:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h17:
  begin
     sdai <= ldata[2];
     state <= state+1;
  end
8'h18:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h19:
  begin
     state <= state+1;
  end
8'h1A:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h1B:
  begin
     sdai <= ldata[1];
     state <= state+1;
  end
8'h1C:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h1D:
  begin
     state <= state+1;
  end
8'h1E:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h1F:
  begin
     sdai <= ldata[0];
     state <= state+1;
  end
8'h20:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h21:
  begin
```

```
            state <= state+1;
          end
        8'h22:
          begin
             scl <= 1'b0;
             state <= state+1;
          end
        8'h23: // Acknowledge bit
          begin
             state <= state+1;
          end
        8'h24:
          begin
             scl <= 1'b1;
             state <= state+1;
          end
        8'h25:
          begin
             state <= state+1;
          end
        8'h26:
          begin
             scl <= 1'b0;
             if (load)
           begin
              ldata <= data;
              ack <= 1'b1;
              state <= 3;
           end
             else
           state <= state+1;
          end
        8'h27:
          begin
             sdai <= 1'b0;
             state <= state+1;
          end
        8'h28:
          begin
             scl <= 1'b1;
             state <= state+1;
          end
        8'h29:
          begin
             sdai <= 1'b1;
             state <= 0;
          end
          endcase

endmodule
// end of file rtl/labkit/i2c_interface.v


// start of file rtl/labkit/display_16hex.v
////////////////////////////////////////////////////////////////////////////////
// 6.111 FPGA Labkit -- Hex display driver
// Author: Nathan Ickes
//
// This module drives the labkit hex displays and shows the value of
// 8 bytes (16 hex digits) on the displays.
////////////////////////////////////////////////////////////////////////////////

module display_16hex (reset, clock_27mhz, data_in,
                      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_in;        // 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 [5:0] count;
reg [7:0] reset_count;
wire      dreset;
wire      clock = (count<27) ? 0 : 1;

always @(posedge clock_27mhz)
  begin
     count <= reset ? 0 : (count==53 ? 0 : count+1);
     reset_count <= reset ? 100 : ((reset_count==0) ? 0 : reset_count-1);
  end

assign dreset = (reset_count != 0);
assign disp_clock = ~clock;
wire   clock_tick = ((count==27) ? 1 : 0);

////////////////////////////////////////////////////////////////////////
// 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
reg [63:0] data;

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

always @(posedge clock_27mhz)
  if (clock_tick)
    begin
       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
               state <= state+1;
            end

          8'h04:
            begin
               // Setup the control register
               disp_rs <= 1'b1; // Select 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
               data <= data_in;
               state <= state+1;
            end

          8'h06:
            begin
               // Load the user's dot data into the dot reg, char by char
               disp_rs <= 1'b0;                     // Select the dot register
               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
                      data <= data_in;
                      dot_index <= 39;
                   end
               else
                 dot_index <= dot_index-1;  // else loop thru all dots
            end

        endcase // casex(state)
    end

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
// end of file rtl/labkit/display_16hex.v


// start of file rtl/labkit/debounce.v
// Switch Debounce Module; use your system clock for the clock input
// to produce a synchronous, debounced output
module debounce #(
    parameter DELAY=648000   // .01 sec with a 64.8 MHz clock
)(
    input reset, clock,
    input raw,
    output reg debounced
);

    reg [18:0] count;
    reg stable_value;

    always @(posedge clock) begin
        if (reset) begin
            count <= 0;
            stable_value <= raw;
            debounced <= raw;
        end else if (raw != stable_value) begin
            stable_value <= raw;
            count <= 0;
        end else if (count == DELAY)
            debounced <= stable_value;
        else
            count <= count + 1;
    end
endmodule
// end of file rtl/labkit/debounce.v


// start of file rtl/labkit/labkit_top_gestural_remote.v
//////////////////////////////////////////////////////////////////////////////////
// This file was based on labkit_top.v, 26-Nov-05, ichuang@mit.edu -
// sample code for the MIT 6.111 labkit demonstrating use of the ZBT
// memories for video display. Here are some of the original comments:
//   - Video input from the NTSC digitizer is displayed within an XGA
//     (1024x768) window
//   - One ZBT memory (ram0) is used as the video frame buffer, with 18bpp
//   - Since the ZBT is read once for every 4 pixels, this frees up time for
//     data to be stored to the ZBT during other pixel times
//   - The NTSC decoder runs at 27 MHz, whereas the XGA runs at 65 MHz, so we
//     synchronize signals between the two and let the NTSC data be stored
//     to ZBT memory whenever it is available, during cycles when pixel reads
//     are not being performed.
//   - We use a very simple ZBT interface, which does not involve any clock
//     generation or hiding of the pipelining (see zbt_pipelined_interface.v)
```

```
//
// switch[7] enables/disables writing the NTSC signal to the ZBT ram
// switch[6] is used for testing the NTSC decoder
// switch[1] selects between test bar periods; these are stored to ZBT
//           during blanking periods
// switch[0] selects vertical test bars (hardwired; not stored in ZBT)
//////////////////////////////////////////////////////////////////////////////

`default_nettype none

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

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

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

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

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

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

        clock_feedback_out, clock_feedback_in,

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

        rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

        mouse_clock, mouse_data, keyboard_clock, keyboard_data,

        clock_27mhz, clock1, clock2,

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

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

        switch,

        led,

        user1, user2, user3, user4,

        daughtercard,

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

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

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

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

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

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

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

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

inout [43:0] daughtercard;

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

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

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

// Audio Input and Output
assign beep= 1'b0;

// 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_fifo_read = 1'b1;
    assign tv_in_fifo_clock = 1'b0;
    assign tv_in_iso = 1'b1;
    assign tv_in_clock = clock_27mhz;//1'b0;

/* enable RAM pins */

    assign ram0_ce_b = 1'b0;
    assign ram0_oe_b = 1'b0;
    assign ram0_adv_ld = 1'b0;
    assign ram0_bwe_b = 4'h0;

    assign ram1_data = 36'hZ;
    assign ram1_address = 19'h0;
    assign ram1_adv_ld = 1'b0;
    assign ram1_clk = 1'b0;

    //These values has to be set to 0 like ram0 if ram1 is used.
    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;

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

    // User I/Os
    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;

    ////////////////////////////////////////////////////////////////////////
    // Demonstration of ZBT RAM as video memory

    // 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));
```

```
 wire clk;
 wire locked;

 zbt_clock_generator zbt_clock_generator(
     .ref_clock(clock_65mhz),
     .fpga_clock(clk),
     .ram0_clock(ram0_clk),
     .clock_feedback_in(clock_feedback_in),
     .clock_feedback_out(clock_feedback_out),
     .locked(locked)
 );

// power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clk), .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 reset_debounce(power_on_reset, clk, ~button_enter, user_reset);
assign reset = user_reset | power_on_reset;

// display module for debugging

reg [63:0] dispdata;
display_16hex debug_hex_display(reset, clk, dispdata,
         disp_blank, disp_clock, disp_rs, disp_ce_b,
         disp_reset_b, disp_data_out);

// generate basic XVGA video signals
wire [10:0] hcount;
wire [9:0]  vcount;
wire hsync,vsync,blank_n;

 // wire up to ZBT ram
 wire ram0_clk_not_used;
 wire [35:0] zbt_write_data;
 wire [35:0] zbt_read_data;
 wire zbt_write_enable;
 wire [18:0] zbt_read_addr;
 wire [18:0] zbt_write_addr;

 zbt_pipelined_interface zbt_pipelined_interface(
   .clk(clk),
   .cen(1), // (clock enable?)
   .we(zbt_write_enable),
   .read_address(zbt_read_addr),
   .write_address(zbt_write_addr),
   .write_data(zbt_write_data),
   .read_data(zbt_read_data),
   // to get good timing, don't connect ram_clk to zbt_pipelined_interface
   .ram_clk(ram0_clk_not_used),
   .ram_we_b(ram0_we_b),
   .ram_address(ram0_address),
   .ram_data(ram0_data),
   .ram_cen_b(ram0_cen_b)
 );


 // generate pixel value from reading ZBT memory
 wire [17:0] vram_pixel;

 // ADV7185 NTSC decoder initialization module
 adv7185_init adv7185_init(.reset(reset), .clock_27mhz(clock_27mhz),
          .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
          .tv_in_i2c_clock(tv_in_i2c_clock),
          .tv_in_i2c_data(tv_in_i2c_data));

 ////////////////////////////////////////////////////////////////////////
 // Instantiate the video acquisition subsystem
 ////////////////////////////////////////////////////////////////////////
```

```
video_acquisition_subsystem video_acquisition_subsystem(
    .clock(clk),
    .reset(reset),
    .adv7185_line_in_clock(tv_in_line_clock1),
    .adv7185_pixel_port(tv_in_ycrcb),
    .debug_switch(~switch[6]),
    .vram_write_address(zbt_write_addr),
    .vram_write_data(zbt_write_data),
    .vram_write_enable(zbt_write_enable),
    .vram_pixel(vram_pixel),
    .vram_read_address(zbt_read_addr),
    .vram_read_data(zbt_read_data),
    .hcount(hcount),
    .vcount(vcount),
    .vsync(vsync),
    .hsync(hsync),
    .blank(blank_n),
    .show_background_only(~switch[7]),
    .use_pattern_as_background(~switch[4])
);

// select output pixel data
reg [17:0] output_pixel;
reg b, hs, vs;

wire bypass_vram = switch[0];
wire [17:0] hardwired_pattern_data = { hcount[8:0], vcount[8:0] };

parameter rgb_red = 18'b111111_000000_000000;
parameter rgb_green = 18'b000000_111111_000000;
parameter rgb_blue = 18'b000000_000000_111111;

wire [9:0] right_hand_x;
wire [9:0] right_hand_y;
wire [9:0] left_hand_x;
wire [9:0] left_hand_y;
wire right_hand_present;
wire left_hand_present;
wire [23:0] hsv;

`include "pixel_functions.v"

// overlay pixel data with red and green crosshairs at the center
// of mass of the red and green colors
wire current_pixel_is_red, current_pixel_is_green;
wire highlight_red_pixels = switch[5];
wire show_crosshair = switch[3];
reg [17:0] crosshaired_pixel;
reg [9:0] crosshair_x = 100;
reg [9:0] crosshair_y = 100;

always @(*) begin

    if (highlight_red_pixels)
        crosshaired_pixel =
            point_is_in_crosshair(right_hand_x, right_hand_y, hcount[9:0], vcount[9:0]) ? rgb_red :
            point_is_in_crosshair(left_hand_x, left_hand_y, hcount[9:0], vcount[9:0]) ? rgb_green :
            current_pixel_is_red ? rgb_red :
            current_pixel_is_green ? rgb_green :
                (hcount==crosshair_x || vcount==crosshair_y) ? rgb_blue :
            vram_pixel;
    else
            crosshaired_pixel =
                point_is_in_crosshair(right_hand_x, right_hand_y, hcount[9:0], vcount[9:0]) ? rgb_red :
                point_is_in_crosshair(left_hand_x, left_hand_y, hcount[9:0], vcount[9:0]) ? rgb_green :
                vram_pixel;
end

reg [23:0] hsv_blue_crosshair;

always @(posedge clk) begin
```

```
        if (!button_up && crosshair_y>CENTER_OF_MASS_Y_MIN && vcount==767 && hcount==1023) crosshair_y <=
crosshair_y - 1;
        if (!button_down && crosshair_y<CENTER_OF_MASS_Y_MAX && vcount==767 && hcount==1023) crosshair_y <=
crosshair_y + 1;
        if (!button_left && crosshair_x>CENTER_OF_MASS_X_MIN && vcount==767 && hcount==1023) crosshair_x <=
crosshair_x - 1;
        if (!button_right && crosshair_x<CENTER_OF_MASS_X_MAX && vcount==767 && hcount==1023) crosshair_x <=
crosshair_x + 1;

        if (crosshair_y==vcount && crosshair_x==hcount) hsv_blue_crosshair = hsv;
    end

    always @(posedge clk) begin
        output_pixel <= bypass_vram ? hardwired_pattern_data : crosshaired_pixel;
        b <= blank_n;
        hs <= hsync;
        vs <= vsync;
    end

    wire [17:0] red_pixels_count;
    wire [17:0] output_pixel_delayed;

    // VGA Output.
    assign vga_out_red   = { output_pixel[17:12], 2'b00 };
    assign vga_out_green = { output_pixel[11:6], 2'b00 };
    assign vga_out_blue  = { output_pixel[5:0], 2'b00 };

    // in order to meet the setup and hold times of the AD7125, we send it ~clk
    assign vga_out_pixel_clock = ~clk;
    assign vga_out_blank_b = ~b;
    assign vga_out_hsync = hs;
    assign vga_out_vsync = vs;
    assign vga_out_sync_b = 1'b1;    // not used

    wire [7:0] vram_pixel_red   = { vram_pixel[17:12], 2'b00 };
    wire [7:0] vram_pixel_green = { vram_pixel[11:6], 2'b00 };
    wire [7:0] vram_pixel_blue  = { vram_pixel[5:0], 2'b00};

    wire lightness_correction_enabled = switch[1];

    //instantiate rgb to hsv converter, TAKES 22 CLOCK CYCLES
    rgb_to_hsv rgb_to_hsv(
        .clock(clk), .reset(reset),
        .r(vram_pixel_red),
        .g(vram_pixel_green),
        .b(vram_pixel_blue),
        .h(hsv[23:16]),
        .s(hsv[15:8]),
        .v(hsv[7:0])
    );

    center_of_mass center_of_mass (
        .clock(clk), .reset(reset),
        .x(hcount[10:0]),
        .y(vcount[9:0]),
        .hsv(hsv),
        .right_hand_x(right_hand_x),
        .right_hand_y(right_hand_y),
        .left_hand_x(left_hand_x),
        .left_hand_y(left_hand_y),
        .right_hand_present(right_hand_present),
        .left_hand_present(left_hand_present),
        .logic_analyzer_bus(),
        .red_count_out(red_pixels_count),
        .x_dividend(),
        .x_divisor(),
        .current_pixel_is_red(current_pixel_is_red),
        .current_pixel_is_green(current_pixel_is_green)
    );

    ////////////////////////////////////////////////////////////////////////
    // Instantiate the gesture recognizer module
    ////////////////////////////////////////////////////////////////////////
```

```
wire gesture_detected;
wire [7:0] gesture_id;
wire [31:0] debug_data;
gestures_recognition gestures_recognition(
    .clock(clk),
    .reset(reset),
    .right_hand_x(right_hand_x),
    .right_hand_y(right_hand_y),
    .left_hand_x(left_hand_x),
    .left_hand_y(left_hand_y),
    .right_hand_present(right_hand_present),
    .left_hand_present(left_hand_present),
    .gesture_detected(gesture_detected),
    .gesture_id(gesture_id),
    .debug_data(debug_data)
);

////////////////////////////////////////////////////////////////////////
// Instantiate the user interface and control module
////////////////////////////////////////////////////////////////////////
wire audio_start;
wire [7:0] sound_select;
wire ir_start;
wire [6:0] ir_command;
wire [4:0] ir_address;
user_interface_and_control user_interface_and_control(
    .clock(clk),
    .reset(reset),
    .gesture_detected(gesture_detected),
    .gesture_id(gesture_id),
    .audio_start(audio_start),
    .sound_select(sound_select),
    .ir_start(ir_start),
    .ir_command(ir_command),
    .ir_address(ir_address)
);

////////////////////////////////////////////////////////////////////////
// Instantiate the infrared subsystem
////////////////////////////////////////////////////////////////////////
wire ir_led;
infrared_subsystem infrared_subsystem(
    .clock(clk),
    .reset(reset),
    .start(ir_start),
    .ir_command(ir_command),
    .ir_address(ir_address),
    .ir_led(ir_led)
);
assign user1[31:1] = 31'hZ;
assign user1[0] = ir_led;

////////////////////////////////////////////////////////////////////////
// Flash stuff
////////////////////////////////////////////////////////////////////////

 wire [1:0] fop;
wire audio_subsystem_read_request;
wire [22:0] audio_subsystem_read_address;
wire [15:0] fwdata, flash_manager_read_data;
wire fbusy;
wire [639:0] dots;

 wire busy;
 wire [7:0] data;
 wire ac97_ready;
 wire [7:0] fifo_data = user4[7:0];
 wire rxf_n = user4[9];
 wire rd_n;
 assign user4[8] =  rd_n;

flash_manager flash_manager (
    .clock(clock_27mhz),
```

```
    .reset(~button0),
    .dots(dots),
    .writemode(0),
    .wdata(0),
    .dowrite(0),
    .raddr(audio_subsystem_read_address),
    .frdata(flash_manager_read_data),
    .doread(audio_subsystem_read_request),
    .busy(busy),
    .flash_data(flash_data),
    .flash_address(flash_address),
    .flash_ce_b(flash_ce_b),
        .flash_oe_b(flash_oe_b),
        .flash_we_b(flash_we_b),
        .flash_reset_b(flash_reset_b),
        .flash_sts(flash_sts),
        .flash_byte_b(flash_byte_b),
    .fsmstate());


////////////////////////////////////////////////////////////////////////
// Instantiate the audio subsystem
////////////////////////////////////////////////////////////////////////

// 8-bit PCM data to AC97 codec
wire [7:0] to_ac97_data;
wire use_flash_memory = switch[2];
audio_subsystem audio_subsystem(
    .clock(clk),
    .reset(reset),
    .ready(ac97_ready),
    .start(audio_start),
    .sound_select(sound_select),
    .to_ac97_data(to_ac97_data),
    .use_flash_memory(use_flash_memory),
    .flash_read_data(flash_manager_read_data),
    .flash_read_address(audio_subsystem_read_address),
    .flash_read_request(audio_subsystem_read_request)
);

// AC97 interface
ac97_monaural_interface ac97_monaural_interface(
    .clock_27mhz(clk),
    .reset(reset),
    .volume(5'd31),
    .audio_in_data(),
    .audio_out_data(to_ac97_data),
    .ready(ac97_ready),
    .audio_reset_b(audio_reset_b),
    .ac97_sdata_out(ac97_sdata_out),
    .ac97_sdata_in(ac97_sdata_in),
    .ac97_synch(ac97_synch),
    .ac97_bit_clock(ac97_bit_clock)
);

// output detected gesture ID to leds
reg [7:0] debug_leds;
always @(posedge clk) begin
    if (switch[0])
        debug_leds <= 8'b0000_0000;
    else if (gesture_detected)
        debug_leds <=
            (gesture_id == 0) ? 8'b0000_0001 :
            (gesture_id == 1) ? 8'b0000_0010 :
            (gesture_id == 2) ? 8'b0000_0100 :
            (gesture_id == 3) ? 8'b0000_1000 :
            (gesture_id == 4) ? 8'b0001_0000 :
            (gesture_id == 5) ? 8'b0010_0000 :
            8'b1111_1111;
end
assign led = ~debug_leds;

// output debug data to hex displays
```

```
    wire [9:0] crosshair_x_wire;
    wire [9:0] crosshair_y_wire;
    wire [23:0] hsv_blue_crosshair_wire;
    assign crosshair_x_wire = crosshair_x;
    assign crosshair_y_wire = crosshair_y;
    assign hsv_blue_crosshair_wire = hsv_blue_crosshair;

    always @(posedge clk) begin

        if (!switch[3])
            dispdata <= {
                    { 2'b0, right_hand_x },          // 12 bits (3 digits)
                    { 2'b0, right_hand_y },          // 12 bits (3 digits)
                    { 3'b0, right_hand_present },    //  4 bits (1 digit)
                    { 3'b0, left_hand_present },      //  4 bits (1 digit)

                    debug_data                          //  32 bits (8 digits)
            };
            else

                dispdata <= {
                    { 2'b0, crosshair_x_wire },     // 12 bits (3 digits)
                    { 2'b0, crosshair_y_wire },     // 12 bits (3 digits)
                    8'b0,                                 // zeros (2 digits)
                    hsv_blue_crosshair_wire,
                    8'b0                               //
            };
    end
endmodule


// end of file rtl/labkit/labkit_top_gestural_remote.v


// start of file rtl/gesture_recognition/gesture_03_left_hand_hold_hand_up_states.v
//////////////////////////////////////////////////////////////////////////////////
// States definition for the hold left hand up gesture recognition FSM.
//////////////////////////////////////////////////////////////////////////////////

// initial idle state
parameter G03ST_IDLE = 0;
parameter G03ST_REGION_1 = 1;
parameter G03ST_REGION_2 = 2;
parameter G03ST_REGION_3 = 3;
parameter G03ST_HOLD = 4;
parameter G03ST_GESTURE_DETECTED = 5;
// end of file rtl/gesture_recognition/gesture_03_left_hand_hold_hand_up_states.v


// start of file rtl/gesture_recognition/gesture_05_right_hand_swipe_left_states.v
//////////////////////////////////////////////////////////////////////////////////
// States definition for the right swipe gesture recognition state machine.
//////////////////////////////////////////////////////////////////////////////////

// initial idle state
parameter G05ST_IDLE = 0;
parameter G05ST_REGION_1 = 1;
parameter G05ST_REGION_2 = 2;
parameter G05ST_REGION_3 = 3;
parameter G05ST_REGION_4 = 4;
parameter G05ST_GESTURE_DETECTED = 5;
// end of file rtl/gesture_recognition/gesture_05_right_hand_swipe_left_states.v


// start of file rtl/gesture_recognition/gesture_05_right_hand_swipe_left.v
//////////////////////////////////////////////////////////////////////////////////
// FSM for recognition of the left swipe gesture (gesture #05).
//////////////////////////////////////////////////////////////////////////////////

module gesture_05_right_hand_swipe_left(
    input clock,
    input reset,
    input [9:0] right_hand_x,
```

```
    input [9:0] right_hand_y,
    input right_hand_present,
    input [9:0] left_hand_x,
    input [9:0] left_hand_y,
    input left_hand_present,
    output gesture_detected,
    output [3:0] state,
    output [7:0] debug_data
);

    `include "gesture_05_right_hand_swipe_left_states.v"
    `include "../common/global_definitions.v"


    parameter X1 = 250;
    parameter X2 = 300;
    parameter X3 = 400;
    parameter X4 = 500;
    parameter X5 = 550;
    parameter VERTICAL_DISPLACEMENT_MAX = 80;

    reg [3:0] current_state, next_state;
    reg [9:0] y_min, y_max;


    always @(posedge clock) begin
        if (reset) begin
            y_min <= 0;
            y_max <= NTSC_V_RES;
        end else begin
            if (current_state == G05ST_IDLE) begin
                y_min <= right_hand_y - VERTICAL_DISPLACEMENT_MAX;
                y_max <= right_hand_y + VERTICAL_DISPLACEMENT_MAX;
                //$strobe("y_min: %0d, y_max: %0d", y_min, y_max);
            end
        end
    end

    wire y_valid = (y_min <= right_hand_y) && (right_hand_y <= y_max);

    // next state logic
    always @(*) begin
        next_state = current_state;

        if (!right_hand_present || !y_valid)
            next_state = G05ST_IDLE;
        else
            case (current_state)
                G05ST_IDLE:
                    if (right_hand_present && (right_hand_x < X1))
                        next_state = G05ST_REGION_1;
                G05ST_REGION_1:
                    if (right_hand_x > X2)
                         next_state = G05ST_REGION_2;
                G05ST_REGION_2:
                    if (right_hand_x > X3)
                         next_state = G05ST_REGION_3;
                G05ST_REGION_3:
                    if (right_hand_x > X4)
                         next_state = G05ST_REGION_4;
                G05ST_REGION_4:
                    if (right_hand_x > X5)
                         next_state = G05ST_GESTURE_DETECTED;
                default:
                    next_state = G05ST_IDLE;
            endcase
    end

    always @(posedge clock) begin
        current_state <= reset ? G05ST_IDLE : next_state;
        if (current_state != next_state)
            $display("Changing state: %0d --> %0d", current_state, next_state);
    end
```

```
    assign gesture_detected = (current_state == G05ST_GESTURE_DETECTED);

    // debug outputs
    assign state = current_state;
    assign debug_data = {
        next_state,
        current_state
    };

endmodule
// end of file rtl/gesture_recognition/gesture_05_right_hand_swipe_left.v


// start of file rtl/gesture_recognition/gesture_03_left_hand_hold_hand_up.v
////////////////////////////////////////////////////////////////////////////////
// FSM for recognition of the hold right hand up gesture (gesture #02).
////////////////////////////////////////////////////////////////////////////////

module gesture_03_left_hand_hold_hand_up(
    input clock,
    input reset,
    input [9:0] right_hand_x,
    input [9:0] right_hand_y,
    input right_hand_present,
    input [9:0] left_hand_x,
    input [9:0] left_hand_y,
    input left_hand_present,
    input time_base_100_ms,
    output gesture_detected,
    output [3:0] state
);

    `include "gesture_03_left_hand_hold_hand_up_states.v"
    `include "../common/global_definitions.v"

    parameter Y1 = 350;
    parameter Y2 = 300;
    parameter Y3 = 250;
    parameter Y4 = 200;
    parameter HORIZONTAL_DISPLACEMENT_MAX = 160;

    reg [3:0] current_state, next_state;
    reg [7:0] state_counter, state_delay;
    wire state_timer_expired;

    reg [9:0] x_min, x_max;
    always @(posedge clock) begin
        if (reset) begin
            x_min <= 0;
            x_max <= NTSC_V_RES;
        end else begin
            if (current_state == G03ST_IDLE) begin
                x_min <= left_hand_x - HORIZONTAL_DISPLACEMENT_MAX;
                x_max <= left_hand_x + HORIZONTAL_DISPLACEMENT_MAX;
                //$strobe("x_min: %0d, x_max: %0d", x_min, x_max);
            end
        end
    end

    wire x_valid = (x_min <= left_hand_x) && (left_hand_x <= x_max);

    // next state logic
    always @(*) begin
        next_state = current_state;

        if (!left_hand_present || !x_valid) begin
            next_state = G03ST_IDLE;
        end else
            case (current_state)
                G03ST_IDLE:
                    if (left_hand_present && left_hand_y > Y1)
                        next_state = G03ST_REGION_1;
```

```
                G03ST_REGION_1:
                    if (left_hand_y < Y2)
                         next_state = G03ST_REGION_2;
                G03ST_REGION_2:
                    if (left_hand_y < Y3)
                         next_state = G03ST_REGION_3;
                G03ST_REGION_3:
                    if (left_hand_y < Y4)
                         next_state = G03ST_HOLD;
                G03ST_HOLD:
                    if (left_hand_y > Y3)
                         next_state = G03ST_IDLE;
                    else if (state_timer_expired)
                        next_state = G03ST_GESTURE_DETECTED;
                G03ST_GESTURE_DETECTED:
                    next_state = G03ST_HOLD;
                default:
                    next_state = G03ST_IDLE;
            endcase
    end

    always @(posedge clock) begin
        current_state <= reset ? G03ST_IDLE : next_state;
        if (current_state != next_state)
            $display("Changing state: %0d --> %0d", current_state, next_state);
    end

    assign gesture_detected = (current_state == G03ST_GESTURE_DETECTED);

    // define delay value for specific states
    always @(*) case (next_state)
        G03ST_HOLD: state_delay = 18;
        default: state_delay = 0;
    endcase

    always @(posedge clock) begin
        if (current_state == next_state && time_base_100_ms)
            if (state_counter > 0)
                state_counter <= state_counter - 1;
        if (current_state != next_state)
            state_counter <= state_delay - 1;

        //$strobe("state_counter: %0d, state_delay: %0d", state_counter, state_delay);
        //$strobe("state_timer_expired: %b", state_timer_expired);
    end

    assign state_timer_expired = (state_counter == 0);

    // debug outputs
    assign state = current_state;

endmodule

// end of file rtl/gesture_recognition/gesture_03_left_hand_hold_hand_up.v


// start of file rtl/gesture_recognition/gesture_02_right_hand_hold_hand_up.v

////////////////////////////////////////////////////////////////////////////////
// FSM for recognition of the hold right hand up gesture (gesture #02).
////////////////////////////////////////////////////////////////////////////////

module gesture_02_right_hand_hold_hand_up(
    input clock,
    input reset,
    input [9:0] right_hand_x,
    input [9:0] right_hand_y,
    input right_hand_present,
    input [9:0] left_hand_x,
    input [9:0] left_hand_y,
    input left_hand_present,
    input time_base_100_ms,
    output gesture_detected,
```

```
    output [3:0] state
);

    `include "gesture_02_right_hand_hold_hand_up_states.v"
    `include "../common/global_definitions.v"

    parameter Y1 = 350;
    parameter Y2 = 300;
    parameter Y3 = 250;
    parameter Y4 = 200;
    parameter HORIZONTAL_DISPLACEMENT_MAX = 160;

    reg [3:0] current_state, next_state;
    reg [7:0] state_counter, state_delay;
    wire state_timer_expired;

    reg [9:0] x_min, x_max;
    always @(posedge clock) begin
        if (reset) begin
            x_min <= 0;
            x_max <= NTSC_V_RES;
        end else begin
            //if (current_state == G02ST_IDLE && next_state == G02ST_REGION_1) begin
            if (current_state == G02ST_IDLE) begin
                x_min <= right_hand_x - HORIZONTAL_DISPLACEMENT_MAX;
                x_max <= right_hand_x + HORIZONTAL_DISPLACEMENT_MAX;
                //$strobe("x_min: %0d, x_max: %0d", x_min, x_max);
            end
        end
    end

    wire x_valid = (x_min <= right_hand_x) && (right_hand_x <= x_max);

    // next state logic
    always @(*) begin
        next_state = current_state;

        if (!right_hand_present || !x_valid) begin
            next_state = G02ST_IDLE;
        end else
            case (current_state)
                G02ST_IDLE:
                    if (right_hand_present && right_hand_y > Y1)
                        next_state = G02ST_REGION_1;
                G02ST_REGION_1:
                    if (right_hand_y < Y2)
                         next_state = G02ST_REGION_2;
                G02ST_REGION_2:
                    if (right_hand_y < Y3)
                         next_state = G02ST_REGION_3;
                G02ST_REGION_3:
                    if (right_hand_y < Y4)
                         next_state = G02ST_HOLD;
                G02ST_HOLD:
                    if (right_hand_y > Y3)
                         next_state = G02ST_IDLE;
                    else if (state_timer_expired)
                        next_state = G02ST_GESTURE_DETECTED;
                G02ST_GESTURE_DETECTED:
                    next_state = G02ST_HOLD;
                default:
                    next_state = G02ST_IDLE;
            endcase
    end

    always @(posedge clock) begin
        current_state <= reset ? G02ST_IDLE : next_state;
        if (current_state != next_state)
            $display("Changing state: %0d --> %0d", current_state, next_state);
    end

    assign gesture_detected = (current_state == G02ST_GESTURE_DETECTED);
```

```
    // define delay value for specific states
    always @(*) case (next_state)
        G02ST_HOLD: state_delay = 18;
        default: state_delay = 0;
    endcase

    always @(posedge clock) begin
        if (current_state == next_state && time_base_100_ms)
            if (state_counter > 0)
                state_counter <= state_counter - 1;
        if (current_state != next_state)
            state_counter <= state_delay - 1;

        //$strobe("state_counter: %0d, state_delay: %0d", state_counter, state_delay);
        //$strobe("state_timer_expired: %b", state_timer_expired);
    end

    assign state_timer_expired = (state_counter == 0);

    // debug outputs
    assign state = current_state;

endmodule

// end of file rtl/gesture_recognition/gesture_02_right_hand_hold_hand_up.v


// start of file rtl/gesture_recognition/gesture_04_right_hand_swipe_right.v

////////////////////////////////////////////////////////////////////////////////
// FSM for recognition of the right swipe gesture (gesture #00).
////////////////////////////////////////////////////////////////////////////////

module gesture_04_right_hand_swipe_right(
    input clock,
    input reset,
    input [9:0] right_hand_x,
    input [9:0] right_hand_y,
    input right_hand_present,
    input [9:0] left_hand_x,
    input [9:0] left_hand_y,
    input left_hand_present,
    output gesture_detected,
    output [3:0] state,
    output [7:0] debug_data
);

    `include "gesture_04_right_hand_swipe_right_states.v"
    `include "../common/global_definitions.v"

    parameter X1 = 550;
    parameter X2 = 500;
    parameter X3 = 400;
    parameter X4 = 300;
    parameter X5 = 250;
    parameter VERTICAL_DISPLACEMENT_MAX = 80;

    reg [3:0] current_state, next_state;
    reg [9:0] y_min, y_max;


    always @(posedge clock) begin
        if (reset) begin
            y_min <= 0;
            y_max <= NTSC_V_RES;
        end else begin
            //if (current_state == G04ST_IDLE && next_state == G04ST_REGION_1) begin
            if (current_state == G04ST_IDLE) begin
                y_min <= right_hand_y - VERTICAL_DISPLACEMENT_MAX;
                y_max <= right_hand_y + VERTICAL_DISPLACEMENT_MAX;
                //$strobe("y_min: %0d, y_max: %0d", y_min, y_max);
            end
        end
```

```
    end

    wire y_valid = (y_min <= right_hand_y) && (right_hand_y <= y_max);

    // next state logic
    always @(*) begin
        next_state = current_state;

        if (!right_hand_present || !y_valid)
            next_state = G04ST_IDLE;
        else
            case (current_state)
                G04ST_IDLE:
                    if (right_hand_present && (right_hand_x > X1))
                        next_state = G04ST_REGION_1;
                G04ST_REGION_1:
                    if (right_hand_x < X2)
                         next_state = G04ST_REGION_2;
                G04ST_REGION_2:
                    if (right_hand_x < X3)
                         next_state = G04ST_REGION_3;
                G04ST_REGION_3:
                    if (right_hand_x < X4)
                         next_state = G04ST_REGION_4;
                G04ST_REGION_4:
                    if (right_hand_x < X5)
                         next_state = G04ST_GESTURE_DETECTED;
                default:
                    next_state = G04ST_IDLE;
            endcase
    end

    always @(posedge clock) begin
        current_state <= reset ? G04ST_IDLE : next_state;
        if (current_state != next_state)
            $display("Changing state: %0d --> %0d", current_state, next_state);
    end

    assign gesture_detected = (current_state == G04ST_GESTURE_DETECTED);

    // debug outputs
    assign state = current_state;
    assign debug_data = {
        next_state,
        current_state
    };

endmodule
// end of file rtl/gesture_recognition/gesture_04_right_hand_swipe_right.v


// start of file rtl/gesture_recognition/gestures_recognition.v
module gestures_recognition(
    input clock,
    input reset,
    input [9:0] right_hand_x,
    input [9:0] right_hand_y,
    input [9:0] left_hand_x,
    input [9:0] left_hand_y,
    input right_hand_present,
    input left_hand_present,
    output gesture_detected,
    output [7:0] gesture_id,
    output [31:0] debug_data
);
    `include "../common/global_definitions.v"

    // debug
    wire [3:0] state_fsm_00, state_fsm_01, state_fsm_02, state_fsm_03, state_fsm_04, state_fsm_05;

    // bit vector for state machine outputs; gestures_detected[N] == 1
    // means that gesture N was detected on the current clock cycle
    wire gestures_detected[15:0];
```

```
// 10 Hz timebase used by the FSMs to generate state delays
wire time_base_100_ms;

time_base_generator #(
    .CLOCK_FREQUENCY_IN_HZ   (CLOCK_FREQUENCY ),
    .TIME_BASE_PERIOD_IN_US  (100_000         )
) clock_divider_100_ms (
    .clock                   (clock           ),
    .reset                   (reset           ),
    .time_base               (time_base_100_ms)
);

gesture_00_left_hand_wave gesture_00_left_hand_wave(
    .clock(clock),
    .reset(reset),
    .right_hand_x(right_hand_x),
    .right_hand_y(right_hand_y),
    .right_hand_present(right_hand_present),
    .left_hand_x(left_hand_x),
    .left_hand_y(left_hand_y),
    .left_hand_present(left_hand_present),
    .gesture_detected(gestures_detected[0]),
    .state(state_fsm_00),
    .debug_data()
);

gesture_01_both_hands_draw_T gesture_01_both_hands_draw_T(
    .clock(clock),
    .reset(reset),
    .right_hand_x(right_hand_x),
    .right_hand_y(right_hand_y),
    .right_hand_present(right_hand_present),
    .left_hand_x(left_hand_x),
    .left_hand_y(left_hand_y),
    .left_hand_present(left_hand_present),
    .gesture_detected(gestures_detected[1]),
    .state(state_fsm_01)
);

gesture_02_right_hand_hold_hand_up gesture_02_right_hand_hold_hand_up(
    .clock(clock),
    .reset(reset),
    .right_hand_x(right_hand_x),
    .right_hand_y(right_hand_y),
    .right_hand_present(right_hand_present),
    .left_hand_x(left_hand_x),
    .left_hand_y(left_hand_y),
    .left_hand_present(left_hand_present),
    .time_base_100_ms(time_base_100_ms),
    .gesture_detected(gestures_detected[2]),
    .state(state_fsm_02)
);

gesture_03_left_hand_hold_hand_up gesture_03_left_hand_hold_hand_up(
    .clock(clock),
    .reset(reset),
    .right_hand_x(right_hand_x),
    .right_hand_y(right_hand_y),
    .right_hand_present(right_hand_present),
    .left_hand_x(left_hand_x),
    .left_hand_y(left_hand_y),
    .left_hand_present(left_hand_present),
    .time_base_100_ms(time_base_100_ms),
    .gesture_detected(gestures_detected[3]),
    .state(state_fsm_03)
);

gesture_04_right_hand_swipe_right gesture_04_right_hand_swipe_right(
    .clock(clock),
    .reset(reset),
    .right_hand_x(right_hand_x),
    .right_hand_y(right_hand_y),
```

```
        .right_hand_present(right_hand_present),
        .left_hand_x(left_hand_x),
        .left_hand_y(left_hand_y),
        .left_hand_present(left_hand_present),
        .gesture_detected(gestures_detected[4]),
        .state(state_fsm_04),
        .debug_data()
    );

    gesture_05_right_hand_swipe_left gesture_05_right_hand_swipe_left(
        .clock(clock),
        .reset(reset),
        .right_hand_x(right_hand_x),
        .right_hand_y(right_hand_y),
        .right_hand_present(right_hand_present),
        .left_hand_x(left_hand_x),
        .left_hand_y(left_hand_y),
        .left_hand_present(left_hand_present),
        .gesture_detected(gestures_detected[5]),
        .state(state_fsm_05),
        .debug_data()
    );

    wire [7:0] selected_gesture =
        gestures_detected[0] ? 0 :
        gestures_detected[1] ? 1 :
        gestures_detected[2] ? 2 :
        gestures_detected[3] ? 3 :
        gestures_detected[4] ? 4 :
        gestures_detected[5] ? 5 :
        0;

    assign gesture_detected =
        gestures_detected[0] ||
        gestures_detected[1] ||
        gestures_detected[2] ||
        gestures_detected[3] ||
        gestures_detected[4] ||
        gestures_detected[5];

    assign gesture_id = gesture_detected ? selected_gesture : 0;
    assign debug_data = {8'b0, state_fsm_05, state_fsm_04, state_fsm_03, state_fsm_02, state_fsm_01,
state_fsm_00 };

endmodule
// end of file rtl/gesture_recognition/gestures_recognition.v


// start of file rtl/gesture_recognition/gesture_00_left_hand_wave.v
////////////////////////////////////////////////////////////////////////////////
// FSM for recognition of the wave to the camera (gesture #00).
////////////////////////////////////////////////////////////////////////////////

module gesture_00_left_hand_wave(
    input clock,
    input reset,
    input [9:0] right_hand_x,
    input [9:0] right_hand_y,
    input right_hand_present,
    input [9:0] left_hand_x,
    input [9:0] left_hand_y,
    input left_hand_present,
    output gesture_detected,
    output [3:0] state,
    output [7:0] debug_data
);

    `include "gesture_00_left_hand_wave_states.v"
    `include "../common/global_definitions.v"

    parameter X1 = 300;
    parameter X2 = 500;
    parameter VERTICAL_DISPLACEMENT_MAX = 80;
```

```
    reg [3:0] current_state, next_state;
    reg [9:0] y_min, y_max;


    always @(posedge clock) begin
        if (reset) begin
            y_min <= 0;
            y_max <= NTSC_V_RES;
        end else begin
            if (current_state == G00ST_IDLE) begin
                y_min <= left_hand_y - VERTICAL_DISPLACEMENT_MAX;
                y_max <= left_hand_y + VERTICAL_DISPLACEMENT_MAX;
                //$strobe("y_min: %0d, y_max: %0d", y_min, y_max);
            end
        end
    end

    wire y_valid = (y_min <= left_hand_y) && (left_hand_y <= y_max);

    // next state logic
    always @(*) begin
        next_state = current_state;

        if (!left_hand_present || !y_valid)
            next_state = G00ST_IDLE;
        else
            case (current_state)
                G00ST_IDLE:
                    if (left_hand_present && (left_hand_x > X2))
                        next_state = G00ST_LEFT_WAVE_RIGHT_EDGE;
                G00ST_LEFT_WAVE_RIGHT_EDGE:
                    if (left_hand_x < X2)
                         next_state = G00ST_LEFT_WAVE_CENTER;
                G00ST_LEFT_WAVE_CENTER:
                    if (left_hand_x < X1)
                         next_state = G00ST_LEFT_WAVE_LEFT_EDGE;
                G00ST_LEFT_WAVE_LEFT_EDGE:
                    if (left_hand_x > X1)
                         next_state = G00ST_RIGHT_WAVE_CENTER;
                G00ST_RIGHT_WAVE_CENTER:
                    if (left_hand_x > X2)
                         next_state = G00ST_RIGHT_WAVE_RIGHT_EDGE;
                G00ST_RIGHT_WAVE_RIGHT_EDGE:
                    if (left_hand_x < X2)
                         next_state = G00ST_FINAL_WAVE_CENTER;
                G00ST_FINAL_WAVE_CENTER:
                    if (left_hand_x < X1)
                         next_state = G00ST_GESTURE_DETECTED;
                G00ST_GESTURE_DETECTED:
                    next_state = G00ST_IDLE;
                default:
                    next_state = G00ST_IDLE;
            endcase
    end

    always @(posedge clock) begin
        current_state <= reset ? G00ST_IDLE : next_state;
        if (current_state != next_state)
            $display("Changing state: %0d --> %0d", current_state, next_state);
    end

    assign gesture_detected = (current_state == G00ST_GESTURE_DETECTED);

    // debug outputs
    assign state = current_state;
    assign debug_data = {
        next_state,
        current_state
    };

endmodule
// end of file rtl/gesture_recognition/gesture_00_left_hand_wave.v
```

```
// start of file rtl/gesture_recognition/gesture_04_right_hand_swipe_right_states.v
////////////////////////////////////////////////////////////////////////////////
// States definition for the right swipe gesture recognition state machine.
////////////////////////////////////////////////////////////////////////////////

// initial idle state
parameter G04ST_IDLE = 0;
parameter G04ST_REGION_1 = 1;
parameter G04ST_REGION_2 = 2;
parameter G04ST_REGION_3 = 3;
parameter G04ST_REGION_4 = 4;
parameter G04ST_GESTURE_DETECTED = 5;
// end of file rtl/gesture_recognition/gesture_04_right_hand_swipe_right_states.v


// start of file rtl/gesture_recognition/gesture_01_both_hands_draw_T_states.v
////////////////////////////////////////////////////////////////////////////////
// States definition for the hold left hand up gesture recognition FSM.
////////////////////////////////////////////////////////////////////////////////

// initial idle state
parameter G01ST_IDLE = 0;
parameter G01ST_UP_SWIPE = 1;
parameter G01ST_SPREAD_HANDS = 2;
parameter G01ST_GESTURE_DETECTED = 3;
// end of file rtl/gesture_recognition/gesture_01_both_hands_draw_T_states.v


// start of file rtl/gesture_recognition/gesture_01_both_hands_draw_T.v
////////////////////////////////////////////////////////////////////////////////
// FSM for recognition of "draw a T with both hands" gesture (gesture #01).
////////////////////////////////////////////////////////////////////////////////

module gesture_01_both_hands_draw_T(
    input clock,
    input reset,
    input [9:0] right_hand_x,
    input [9:0] right_hand_y,
    input right_hand_present,
    input [9:0] left_hand_x,
    input [9:0] left_hand_y,
    input left_hand_present,
    output gesture_detected,
    output [3:0] state
);

    `include "gesture_01_both_hands_draw_T_states.v"
    `include "../common/global_definitions.v"

    parameter X1 = 450;
    parameter X2 = 250;
    parameter Y1 = 350;
    parameter Y2 = 200;
    parameter HANDS_TOGETHER_DISTANCE_MAX = 200;
    parameter HANDS_APART_DISTANCE_MIN = 300;

    wire [9:0] x_distance = (left_hand_x - right_hand_x);
    wire hands_together = (x_distance <= HANDS_TOGETHER_DISTANCE_MAX);
    wire hands_apart = (x_distance >= HANDS_APART_DISTANCE_MIN);

    reg [3:0] current_state, next_state;
    reg [7:0] state_counter, state_delay;

    // next state logic
    always @(*) begin
        next_state = current_state;

        if (!left_hand_present || !right_hand_present) begin
            next_state = G01ST_IDLE;
        end else
            case (current_state)
```

```
                G01ST_IDLE:
                    if (
                        left_hand_present &&
                        right_hand_present &&
                        hands_together &&
                        (left_hand_y > Y1)  && (right_hand_y > Y1)
                    )
                        next_state = G01ST_UP_SWIPE;
                G01ST_UP_SWIPE:
                    if (
                        (left_hand_y < Y2)  && (right_hand_y < Y2)
                    )
                         next_state = G01ST_SPREAD_HANDS;
                G01ST_SPREAD_HANDS:
                    if (hands_apart)
                         next_state = G01ST_GESTURE_DETECTED;
                G01ST_GESTURE_DETECTED:
                    next_state = G01ST_IDLE;
                default:
                    next_state = G01ST_IDLE;
            endcase
    end

    always @(posedge clock) begin
        current_state <= reset ? G01ST_IDLE : next_state;
        if (current_state != next_state)
            $display("Changing state: %0d --> %0d", current_state, next_state);
    end

    assign gesture_detected = (current_state == G01ST_GESTURE_DETECTED);

    // debug outputs
    assign state = current_state;

endmodule
// end of file rtl/gesture_recognition/gesture_01_both_hands_draw_T.v


// start of file rtl/gesture_recognition/gesture_02_right_hand_hold_hand_up_states.v
////////////////////////////////////////////////////////////////////////////////
// States definition for the hold right hand up gesture recognition FSM.
////////////////////////////////////////////////////////////////////////////////

// initial idle state
parameter G02ST_IDLE = 0;
parameter G02ST_REGION_1 = 1;
parameter G02ST_REGION_2 = 2;
parameter G02ST_REGION_3 = 3;
parameter G02ST_HOLD = 4;
parameter G02ST_GESTURE_DETECTED = 5;
// end of file rtl/gesture_recognition/gesture_02_right_hand_hold_hand_up_states.v


// start of file rtl/gesture_recognition/gesture_00_left_hand_wave_states.v
////////////////////////////////////////////////////////////////////////////////
// States definition for the wave gesture recognition state machine.
////////////////////////////////////////////////////////////////////////////////

// initial idle state
parameter G00ST_IDLE = 0;
parameter G00ST_LEFT_WAVE_RIGHT_EDGE = 1;
parameter G00ST_LEFT_WAVE_CENTER = 2;
parameter G00ST_LEFT_WAVE_LEFT_EDGE = 3;
parameter G00ST_RIGHT_WAVE_CENTER = 4;
parameter G00ST_RIGHT_WAVE_RIGHT_EDGE = 5;
parameter G00ST_FINAL_WAVE_CENTER = 6;
parameter G00ST_GESTURE_DETECTED = 7;
// end of file rtl/gesture_recognition/gesture_00_left_hand_wave_states.v


// start of file rtl/video_acquisition/ycbcr_to_rgb.v

// converts YCrCb to RGB
```

```
module ycbcr_to_rgb(
    input clock,
    input reset,
    input [7:0] y,
    input [7:0] cb,
    input [7:0] cr,
    output reg [7:0] r,
    output reg [7:0] g,
    output reg [7:0] b
);

    reg signed [8:0] y_fixed, cb_fixed, cr_fixed;

    // fix y, cb, cr values, to ensure that they're in valid ranges
    always @(*) begin
        if (y<8'd16) y_fixed[8:0] = 9'sd16;
        else if (y>8'd235) y_fixed[8:0] = 9'sd235;
        else y_fixed[8:0] = {1'b0, y};

        if (cb<8'd16) cb_fixed[8:0] = 9'sd16;
        else if (cb>8'd235) cb_fixed[8:0] = 9'sd235;
        else cb_fixed[8:0] = {1'b0,cb};

        if (cr<8'd16) cr_fixed[8:0] = 9'sd16;
        else if (cr>8'd235) cr_fixed[8:0] = 9'sd235;
        else cr_fixed[8:0] = {1'b0,cr};
    end

    // constants used in multiplication (*2^11)
    parameter RGB_Y = 14'sd2383;//1.164
    parameter R_CR = 14'sd3269;//1.596
    parameter G_CR = 14'sd1665;//0.813
    parameter G_CB = 14'sd803;//0.392
    parameter B_CB = 14'sd4131;//2.017

    //outputs of multiplications bit width = 14+9+1 due to possible overflow
    reg signed [23:0] R_scaled;
    reg signed [23:0] G_scaled;
    reg signed [23:0] B_scaled;

    reg signed[9:0]R_signed;
    reg signed[9:0]G_signed;
    reg signed[9:0]B_signed;

    always @(*) begin
        //transformation
        R_scaled = RGB_Y*(y_fixed-9'sd16) + R_CR*(cr_fixed-9'sd128);
        G_scaled = RGB_Y*(y_fixed-9'sd16) - G_CR*(cr_fixed-9'sd128) - G_CB*(cb_fixed-9'sd128);
        B_scaled = RGB_Y*(y_fixed-9'sd16) + B_CB*(cb_fixed-9'sd128);

        //scaling down
        R_signed = R_scaled>>>11;
        G_signed = G_scaled>>>11;
        B_signed = B_scaled>>>11;
    end

    always @(posedge clock) begin
        //saturation and assignment
        if (reset||R_signed<0) r <= 8'd0;
        else if (R_signed>255) r <= 8'd255;
        else r <= R_signed[7:0];

        if (reset||G_signed<0) g <= 8'd0;
        else if (G_signed>255) g <= 8'd255;
        else g <= G_signed[7:0];

        if (reset||B_signed<0)b <= 8'd0;
        else if (B_signed>255)b <= 8'd255;
        else b <= B_signed[7:0];
    end
endmodule
// end of file rtl/video_acquisition/ycbcr_to_rgb.v
```

```
// start of file rtl/video_acquisition/video_acquisition_subsystem.v
////////////////////////////////////////////////////////////////////////////////
// Top-level module wrapping all video acquisition submodules. The basic input
// is the multiplexed data stream from the ADV7185; the wires are the RGB
// pixel values and the corresponding hcount, vcount, and sync pulses.
// The steps involved in the operation are roughly:
//    1. Process the ADV7185 data stream into YCrCb values
//    2. Convert the YCrCb values to RGB
//    3. Write RGB pixel values to the video ram (2 pixels per location)
// This is for the input stuff. For the wire, we read values from the video
// ram, and generate the corresponding XVGA signals:
//    4. Generate hcount, vcount and sync pulses compatible with XVGA (1024x768)
//    5. Calculate vram address from hcount & vcount, and read out pixel values
//    6. wire the pixel value and XVGA sync pulses
////////////////////////////////////////////////////////////////////////////////

`default_nettype none

module video_acquisition_subsystem(
    input clock,
    input reset,
    input adv7185_line_in_clock,
    // video pixel wire port; multiplexed data stream with captured pixels
    input [19:0] adv7185_pixel_port,
    input debug_switch,
    output reg [18:0] vram_write_address,
    output reg [35:0] vram_write_data,
    output reg vram_write_enable,

    output [17:0] vram_pixel,
    output [18:0] vram_read_address,
    input [35:0] vram_read_data,

    output [10:0] hcount,
    output [9:0] vcount,
    output vsync,
    output hsync,
    output blank,

    input show_background_only,
    input use_pattern_as_background
);

    // The Video Pixel wire Port serves different functions depending on the
    // selected wire mode:
    //   * 8-bit multiplexed YCrCb pixel port (P19-P12);
    //   * 16-bit YCrCb pixel port (P19-P12 = Y and P9-P2 = Cb,Cr);
    //   * 10-bit multiplexed extended YCrCb pixel port (P19-P10);
    //   * 20-bit YCrCb pixel port (P19-P0)
    // We use the 3rd mode, so we need to process only bits 19..10
    wire [9:0] multiplexed_extended_ycrcb_pixel_port =
        adv7185_pixel_port[19:10];

    // YCrCb pixel data demultiplexed from the ADV7185 chip
    wire [29:0] ntsc_ycrcb;
    // 1 when the current NTSC field is odd
    wire ntsc_even_field;
    // NTSC synchronism pulses
    wire ntsc_vsync, ntsc_hsync;
    wire ntsc_data_valid;

    // instantiate the interface with the ADV7185 NTSC decoder
    adv7185_interface adv7185_interface (
        .clk(adv7185_line_in_clock),
        .reset(reset),
        .multiplexed_pixel_data(multiplexed_extended_ycrcb_pixel_port),
        .ycrcb(ntsc_ycrcb),
        .even_field(ntsc_even_field),
        .vsync(ntsc_vsync),
        .hsync(ntsc_hsync),
        .data_valid(ntsc_data_valid)
```

```
);

wire [23:0] captured_pixel;

ycbcr_to_rgb ycbcr_to_rgb(
    .clock(clock),
    .reset(reset),
    .y(ntsc_ycrcb[29:22]),
    .cb(ntsc_ycrcb[9:2]),
    .cr(ntsc_ycrcb[19:12]),
    .r(captured_pixel[23:16]),
    .g(captured_pixel[15:8]),
    .b(captured_pixel[7:0])
);

wire [18:0] ntsc_write_address;
wire [35:0] ntsc_write_data;
wire ntsc_write_enable;

vram_writer vram_writer(
  .clk(clock),
  .vclk(adv7185_line_in_clock),
  .fvh( { ntsc_even_field, ntsc_vsync, ntsc_hsync } ),
  .data_valid(ntsc_data_valid),
  .pixel_data(captured_pixel),
  .zbt_write_address(ntsc_write_address),
  .zbt_write_data(ntsc_write_data),
  .zbt_write_enable(ntsc_write_enable),
  .debug_switch(debug_switch)
);

vram_reader vram_reader(
    .reset(reset),
    .clk(clock),
    .hcount(hcount),
    .vcount(vcount),
    .vr_pixel(vram_pixel),
    .vram_read_addr(vram_read_address),
    .vram_read_data(vram_read_data)
);

xvga_timing_generator xvga_timing_generator(
    .vclock(clock),
    .hcount(hcount),
    .vcount(vcount),
    .vsync(vsync),
    .hsync(hsync),
    .blank(blank)
);

// generate a pattern of color bars that can be written to the ZBT memory
reg [31:0] count;
always @(posedge clock) count <= reset ? 0 : count + 1;
wire [18:0] vram_background_addr = count[18:0];
wire [35:0] vram_pattern_data = {
    {6{count[7]}}, {6{count[6]}}, {6{count[5]}},
    {6{count[7]}}, {6{count[6]}}, {6{count[5]}}
};

wire [35:0] vram_background_data =
    use_pattern_as_background ? vram_pattern_data : 0;

always @(*) begin
    if (show_background_only) begin
        vram_write_enable = blank;
        vram_write_address = vram_background_addr;
        vram_write_data = vram_background_data;
    end else begin
        vram_write_enable = hcount[0];
        vram_write_address = ntsc_write_address;
        vram_write_data = ntsc_write_data;
    end
end
```

```
endmodule
// end of file rtl/video_acquisition/video_acquisition_subsystem.v


// start of file rtl/video_acquisition/vram_reader.v
////////////////////////////////////////////////////////////////////////////
// generate display pixels from reading the ZBT ram

module vram_reader(
    reset,
    clk,
    hcount,
    vcount,
    vr_pixel,
    vram_read_addr,
    vram_read_data
);

    input reset, clk;
    input [10:0] hcount;
    input [9:0] vcount;
    output [17:0] vr_pixel;
    output [18:0] vram_read_addr;
    input [35:0] vram_read_data;

    parameter HCOUNT_MAX = 1048;
    parameter VCOUNT_MAX = 805;

    // forecast hcount & vcount 8 clock cycles ahead to get data from ZBT
    wire [10:0] hcount_f =
        (hcount < HCOUNT_MAX) ? (hcount + 4) :
        (hcount - HCOUNT_MAX);

    wire [9:0] vcount_f =
        (hcount < HCOUNT_MAX) ? vcount :
        (vcount == VCOUNT_MAX) ? 0 :
        vcount + 1;

    // formula to calculate the address: addr = 1024 * y + ceil(x/2)
    wire [18:0] vram_read_addr = { vcount_f, hcount_f[9:1] };

    // each ram location holds 2 pixels; we use the MSBs for even
    // pixels (i.e., hcount is even) and the LSBs for odd pixels
    wire pixel_select = hcount[0];

    reg [17:0] vr_pixel;
    reg [35:0] vr_data_latched;
    reg [35:0] last_vr_data;

    always @(posedge clk) begin
        vr_data_latched <= pixel_select ? vr_data_latched : vram_read_data;
        last_vr_data <= pixel_select ? vr_data_latched : last_vr_data;
    end

    // each 36-bit word from RAM is decoded to 2 bytes
    always @(*) case (pixel_select)
        1: vr_pixel = last_vr_data[17:0];
        0: vr_pixel = last_vr_data[35:18];
    endcase

endmodule // vram_reader
// end of file rtl/video_acquisition/vram_reader.v


// start of file rtl/video_acquisition/adv7185_interface.v
// This file contains the adv7185_interface and adv7185_init modules

////////////////////////////////////////////////////////////////////////////
//
// NTSC decode - 16-bit CCIR656 decoder
// By Javier Castro
// This module takes a stream of LLC data from the adv7185
```

```
// NTSC/PAL video decoder and generates the corresponding pixels,
// that are encoded within the stream, in YCrCb format.

// Make sure that the adv7185 is set to run in 16-bit LLC2 mode.

module adv7185_interface(clk, reset, multiplexed_pixel_data, ycrcb, even_field, vsync, hsync, data_valid);

    // clk - line-locked clock (in this case, LLC1 which runs at 27Mhz)
    // reset - system reset
    // multiplexed_pixel_data - 10-bit input from chip. should map to pins [19:10]
    // ycrcb - 24 bit luminance and chrominance (8 bits each)
    // even_field - field: 1 indicates an even field, 0 an odd field
    // vsync - vertical sync: 1 means vertical sync
    // hsync - horizontal sync: 1 means horizontal sync

    input clk;
    input reset;
    // 10-bit multiplexed extended YCrCb pixel port
    // modified for 10 bit input - should be P[19:10]
    input [9:0] multiplexed_pixel_data;
    output [29:0] ycrcb;
    output even_field;
    output vsync;
    output hsync;
    output data_valid;
    // output [4:0] state;

    parameter SYNC_1 = 0;
    parameter SYNC_2 = 1;
    parameter SYNC_3 = 2;
    parameter SAV_f1_cb0 = 3;
    parameter SAV_f1_y0 = 4;
    parameter SAV_f1_cr1 = 5;
    parameter SAV_f1_y1 = 6;
    parameter EAV_f1 = 7;
    parameter SAV_VBI_f1 = 8;
    parameter EAV_VBI_f1 = 9;
    parameter SAV_f2_cb0 = 10;
    parameter SAV_f2_y0 = 11;
    parameter SAV_f2_cr1 = 12;
    parameter SAV_f2_y1 = 13;
    parameter EAV_f2 = 14;
    parameter SAV_VBI_f2 = 15;
    parameter EAV_VBI_f2 = 16;

    // In the start state, the module doesn't know where
    // in the sequence of pixels, it is looking.

    // Once we determine where to start, the FSM goes through a normal
    // sequence of SAV process_YCrCb EAV... repeat

    // The data stream looks as follows
    // SAV_FF | SAV_00 | SAV_00 | SAV_XY | Cb0 | Y0 | Cr1 | Y1 | Cb2 | Y2 | ... | EAV sequence
    // There are two things we need to do:
    //   1. Find the two SAV blocks (stands for Start Active Video perhaps?)
    //   2. Decode the subsequent data

    reg [4:0] current_state = 5'h00;
    reg [9:0] y = 10'h000;  // luminance
    reg [9:0] cr = 10'h000; // chrominance
    reg [9:0] cb = 10'h000; // more chrominance

    assign     state = current_state;

    always @ (posedge clk)
      begin
     if (reset)
     begin

     end
     else
     begin
         // these states don't do much except allow us to know where we are in the stream.
```

```
        // whenever the synchronization code is seen, go back to the sync_state before
        // transitioning to the new state
        case (current_state)
        SYNC_1: current_state <= (multiplexed_pixel_data == 10'h000) ? SYNC_2 : SYNC_1;
        SYNC_2: current_state <= (multiplexed_pixel_data == 10'h000) ? SYNC_3 : SYNC_1;
        SYNC_3: current_state <= (multiplexed_pixel_data == 10'h200) ? SAV_f1_cb0 :
                    (multiplexed_pixel_data == 10'h274) ? EAV_f1 :
                    (multiplexed_pixel_data == 10'h2ac) ? SAV_VBI_f1 :
                    (multiplexed_pixel_data == 10'h2d8) ? EAV_VBI_f1 :
                    (multiplexed_pixel_data == 10'h31c) ? SAV_f2_cb0 :
                    (multiplexed_pixel_data == 10'h368) ? EAV_f2 :
                    (multiplexed_pixel_data == 10'h3b0) ? SAV_VBI_f2 :
                    (multiplexed_pixel_data == 10'h3c4) ? EAV_VBI_f2 : SYNC_1;

        SAV_f1_cb0: current_state <= (multiplexed_pixel_data == 10'h3ff) ? SYNC_1 : SAV_f1_y0;
        SAV_f1_y0: current_state <= (multiplexed_pixel_data == 10'h3ff) ? SYNC_1 : SAV_f1_cr1;
        SAV_f1_cr1: current_state <= (multiplexed_pixel_data == 10'h3ff) ? SYNC_1 : SAV_f1_y1;
        SAV_f1_y1: current_state <= (multiplexed_pixel_data == 10'h3ff) ? SYNC_1 : SAV_f1_cb0;

        SAV_f2_cb0: current_state <= (multiplexed_pixel_data == 10'h3ff) ? SYNC_1 : SAV_f2_y0;
        SAV_f2_y0: current_state <= (multiplexed_pixel_data == 10'h3ff) ? SYNC_1 : SAV_f2_cr1;
        SAV_f2_cr1: current_state <= (multiplexed_pixel_data == 10'h3ff) ? SYNC_1 : SAV_f2_y1;
        SAV_f2_y1: current_state <= (multiplexed_pixel_data == 10'h3ff) ? SYNC_1 : SAV_f2_cb0;

        // These states are here in the event that we want to cover these signals
        // in the future. For now, they just send the state machine back to SYNC_1
        EAV_f1: current_state <= SYNC_1;
        SAV_VBI_f1: current_state <= SYNC_1;
        EAV_VBI_f1: current_state <= SYNC_1;
        EAV_f2: current_state <= SYNC_1;
        SAV_VBI_f2: current_state <= SYNC_1;
        EAV_VBI_f2: current_state <= SYNC_1;

        endcase
     end
      end // always @ (posedge clk)

    // implement our decoding mechanism

    wire y_enable;
    wire cr_enable;
    wire cb_enable;

    // if y is coming in, enable the register
    // likewise for cr and cb
    assign y_enable = (current_state == SAV_f1_y0) ||
                (current_state == SAV_f1_y1) ||
                (current_state == SAV_f2_y0) ||
                (current_state == SAV_f2_y1);
    assign cr_enable = (current_state == SAV_f1_cr1) ||
                (current_state == SAV_f2_cr1);
    assign cb_enable = (current_state == SAV_f1_cb0) ||
                (current_state == SAV_f2_cb0);

    // even_field, vsync, and hsync only go high when active
    assign {vsync,hsync} = (current_state == SYNC_3) ? multiplexed_pixel_data[7:6] : 2'b00;

    // data is valid when we have all three values: y, cr, cb
    assign data_valid = y_enable;
    assign ycrcb = {y,cr,cb};

     reg even_field = 0;
     always @ (posedge clk) begin
        y <= y_enable ? multiplexed_pixel_data : y;
        cr <= cr_enable ? multiplexed_pixel_data : cr;
        cb <= cb_enable ? multiplexed_pixel_data : cb;
        even_field <= (current_state == SYNC_3) ? multiplexed_pixel_data[8] : even_field;
     end

endmodule
// end of file rtl/video_acquisition/adv7185_interface.v
```

```
// start of file rtl/video_acquisition/zbt_clock_generator.v
//////////////////////////////////////////////////////////////////////////////
// ramclock module

//////////////////////////////////////////////////////////////////////////////
// 6.111 FPGA Labkit -- ZBT RAM clock generation
// Author: Nathan Ickes
//////////////////////////////////////////////////////////////////////////////
// This module generates deskewed clocks for driving the ZBT SRAMs and FPGA
// registers. A special feedback trace on the labkit PCB (which is length
// matched to the RAM traces) is used to adjust the RAM clock phase so that
// rising clock edges reach the RAMs at exactly the same time as rising clock
// edges reach the registers in the FPGA.
//
// The RAM clock signals are driven by DDR output buffers, which further
// ensures that the clock-to-pad delay is the same for the RAM clocks as it is
// for any other registered RAM signal.
//
// When the FPGA is configured, the DCMs are enabled before the chip-level I/O
// drivers are released from tristate. It is therefore necessary to
// artificially hold the DCMs in reset for a few cycles after configuration.
// This is done using a 16-bit shift register. When the DCMs have locked, the
// <lock> output of this mnodule will go high. Until the DCMs are locked, the
// ouput clock timings are not guaranteed, so any logic driven by the
// <fpga_clock> should probably be held inreset until <locked> is high.
//////////////////////////////////////////////////////////////////////////////

module zbt_clock_generator(ref_clock, fpga_clock, ram0_clock, ram1_clock,
            clock_feedback_in, clock_feedback_out, locked);

   input ref_clock;                // Reference clock input
   output fpga_clock;              // Output clock to drive FPGA logic
   output ram0_clock, ram1_clock;  // Output clocks for each RAM chip
   input  clock_feedback_in;       // Output to feedback trace
   output clock_feedback_out;      // Input from feedback trace
   output locked;                  // Indicates that clock outputs are stable

   wire  ref_clk, fpga_clk, ram_clk, fb_clk, lock1, lock2, dcm_reset;

   //////////////////////////////////////////////////////////////////////////////

   //To force ISE to compile the ramclock, this line has to be removed.
   //IBUFG ref_buf (.O(ref_clk), .I(ref_clock));

    assign ref_clk = ref_clock;

   BUFG int_buf (.O(fpga_clock), .I(fpga_clk));

   DCM int_dcm (.CLKFB(fpga_clock),
        .CLKIN(ref_clk),
        .RST(dcm_reset),
        .CLK0(fpga_clk),
        .LOCKED(lock1));
   // synthesis attribute DLL_FREQUENCY_MODE of int_dcm is "LOW"
   // synthesis attribute DUTY_CYCLE_CORRECTION of int_dcm is "TRUE"
   // synthesis attribute STARTUP_WAIT of int_dcm is "FALSE"
   // synthesis attribute DFS_FREQUENCY_MODE of int_dcm is "LOW"
   // synthesis attribute CLK_FEEDBACK of int_dcm  is "1X"
   // synthesis attribute CLKOUT_PHASE_SHIFT of int_dcm is "NONE"
   // synthesis attribute PHASE_SHIFT of int_dcm is 0

   BUFG ext_buf (.O(ram_clock), .I(ram_clk));

   IBUFG fb_buf (.O(fb_clk), .I(clock_feedback_in));

   DCM ext_dcm (.CLKFB(fb_clk),
            .CLKIN(ref_clk),
            .RST(dcm_reset),
            .CLK0(ram_clk),
            .LOCKED(lock2));
   // synthesis attribute DLL_FREQUENCY_MODE of ext_dcm is "LOW"
   // synthesis attribute DUTY_CYCLE_CORRECTION of ext_dcm is "TRUE"
   // synthesis attribute STARTUP_WAIT of ext_dcm is "FALSE"
```

```
   // synthesis attribute DFS_FREQUENCY_MODE of ext_dcm is "LOW"
   // synthesis attribute CLK_FEEDBACK of ext_dcm  is "1X"
   // synthesis attribute CLKOUT_PHASE_SHIFT of ext_dcm is "NONE"
   // synthesis attribute PHASE_SHIFT of ext_dcm is 0

   SRL16 dcm_rst_sr (.D(1'b0), .CLK(ref_clk), .Q(dcm_reset),
             .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   // synthesis attribute init of dcm_rst_sr is "000F";


   OFDDRRSE ddr_reg0 (.Q(ram0_clock), .C0(ram_clock), .C1(~ram_clock),
              .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));
   OFDDRRSE ddr_reg1 (.Q(ram1_clock), .C0(ram_clock), .C1(~ram_clock),
              .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));
   OFDDRRSE ddr_reg2 (.Q(clock_feedback_out), .C0(ram_clock), .C1(~ram_clock),
              .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));

   assign locked = lock1 && lock2;

endmodule
// end of file rtl/video_acquisition/zbt_clock_generator.v


// start of file rtl/video_acquisition/vram_writer.v
// Author: I. Chuang <ichuang@mit.edu>
// Prepare data and address values to fill ZBT memory with NTSC data

module vram_writer(clk, vclk, fvh, data_valid, pixel_data, zbt_write_address, zbt_write_data,
zbt_write_enable, debug_switch);
    // system clock
    input clk;
    // video clock from NTSC camera
    input vclk;
    // 3 sync pulses ("field", vertical, horizontal)
    input [2:0] fvh;
    // true when YCrCb data from ADV7185 is valid
    input data_valid;
    // YCrCb data from NTSC decoder
    input [23:0] pixel_data;
    // calculated address for writing into the ZBT ram
    output [18:0] zbt_write_address;
    // data to be written to the ZBT ram
    output [35:0] zbt_write_data;
    // write enable for ZBT ram
    output zbt_write_enable;
    // switch which determines mode (for debugging)
    input debug_switch;

    `include "../common/global_definitions.v"

    // offset to show video from the top left corner
    parameter VIDEO_FRAME_X_OFFSET = 0;
    parameter VIDEO_FRAME_Y_OFFSET = 0;

    // generate column and row counters based on the NTSC sync pulses

    reg [9:0] ntsc_x = 0;
    reg [9:0] ntsc_y = 0;
    reg [17:0] ntsc_pixel = 0;
    reg ntsc_wen;
    reg old_data_valid;
    reg old_frame;        // frames are even / odd interlaced
    reg ntsc_even_odd;    // decode interlaced frame to this wire

    wire frame = fvh[2];
    wire frame_edge = frame & ~old_frame;

    wire ntsc_hsync = fvh[0];
    wire ntsc_vsync = fvh[1];
    wire ntsc_field = fvh[2];

    // LLC1 is reference
    always @ (posedge vclk) begin
```

```
    old_frame <= frame;
    old_data_valid <= data_valid;

    // if data valid, write it
    ntsc_wen <= data_valid && !ntsc_field & ~old_data_valid;

    // toggle even/odd frame flag on every frame edge
    ntsc_even_odd <= frame_edge ^ ntsc_even_odd;

    if (!ntsc_field) begin
        if (ntsc_hsync)
            ntsc_x <= VIDEO_FRAME_X_OFFSET;
        else if (!ntsc_vsync && data_valid && (ntsc_x < XVGA_H_RES))
            ntsc_x <= ntsc_x + 1;

        if (ntsc_vsync)
            ntsc_y <= VIDEO_FRAME_Y_OFFSET;
        else if (ntsc_hsync && (ntsc_y < XVGA_V_RES))
            ntsc_y <= ntsc_y + 1;

        if (debug_switch) begin
            if (data_valid) ntsc_pixel <= {
                {3{ntsc_x[8]}}, {3{ntsc_x[7]}}, {3{ntsc_x[6]}},
                {3{ntsc_y[8]}}, {3{ntsc_y[7]}}, {3{ntsc_y[6]}}
            };
        end else begin
            if (data_valid) ntsc_pixel <= {
                pixel_data[23:18],
                pixel_data[15:10],
                pixel_data[7:2]
            };
        end
    end
end

// create a fifo (depth = 2) to synchronize NTSC signals with system clock
reg [9:0] ntsc_x_fifo[1:0];
reg [9:0] ntsc_y_fifo[1:0];
reg [17:0] ntsc_pixel_fifo[1:0];
reg ntsc_wen_fifo[1:0];
reg ntsc_even_odd_fifo[1:0];

always @(posedge clk) begin
    { ntsc_x_fifo[1], ntsc_x_fifo[0] } <= { ntsc_x_fifo[0], ntsc_x };
    { ntsc_y_fifo[1], ntsc_y_fifo[0] } <= { ntsc_y_fifo[0], ntsc_y };
    { ntsc_pixel_fifo[1], ntsc_pixel_fifo[0] } <= { ntsc_pixel_fifo[0], ntsc_pixel };
    { ntsc_wen_fifo[1], ntsc_wen_fifo[0] } <= { ntsc_wen_fifo[0], ntsc_wen };
    { ntsc_even_odd_fifo[1], ntsc_even_odd_fifo[0] } <= { ntsc_even_odd_fifo[0], ntsc_even_odd };
end

reg [35:0] output_data;
wire [35:0] debug_data;
wire [18:0] output_addr, debug_addr;

// create another fifo (depth = 4) to ??? (I don't know what it is for...)
reg [39:0] x_delay_fifo, y_delay_fifo;
reg [3:0] wen_delay_fifo, eo_delay_fifo;

always @ (posedge clk) begin
    x_delay_fifo <= { x_delay_fifo[29:0], ntsc_x_fifo[1] };
    y_delay_fifo <= { y_delay_fifo[29:0], ntsc_y_fifo[1] };
    wen_delay_fifo <= { wen_delay_fifo[2:0], ntsc_wen_fifo[1] };
    eo_delay_fifo <= { eo_delay_fifo[2:0], ntsc_even_odd_fifo[1] };
end

wire [9:0] x_oldest = x_delay_fifo[39:30];
wire [8:0] y_oldest = y_delay_fifo[38:30];

// detect edges on write enable signal
reg wen_oldest;
always @(posedge clk) wen_oldest <= ntsc_wen_fifo[1];
//always @(posedge clk) wen_oldest <= ntsc_wen_fifo_out;
wire we_edge = ntsc_wen_fifo[1] & ~wen_oldest;
```

```
    // shift each set of four bytes into a large register for the ZBT
    always @(posedge clk)
        if (we_edge) output_data <= { output_data[17:0], ntsc_pixel_fifo[1] };

    // compute address to store data into, using the formula:
    // addr = 1024 * y + ceil(x/2)
    // because the frames are interlaced, the LSB of the y coordinate is
    // actually the even/odd bit
    assign output_addr = { y_oldest[8:0], eo_delay_fifo[3], x_oldest[9:1] };

    // update the output address and data only when four bytes ready
    wire zbt_write_enable = we_edge & (x_delay_fifo[30] == 1'b0);

    reg [18:0] zbt_write_address;
    reg [35:0] zbt_write_data;
    always @(posedge clk) begin
        if (zbt_write_enable) begin
            zbt_write_address <= output_addr;
            zbt_write_data <= output_data;
        end
    end

    wire [35:0] colorbars_pixel_data = {
        { 18{x_oldest[4]} },
        { 18{y_oldest[4]} }
    };

    assign debug_data = colorbars_pixel_data;

endmodule // vram_writer
// end of file rtl/video_acquisition/vram_writer.v


// start of file rtl/video_acquisition/adv7185_init.v
////////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- ADV7185 Video Decoder Configuration Init
//
// Created:
// Author: Nathan Ickes
//
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
// Register 0
////////////////////////////////////////////////////////////////////////////////

`define INPUT_SELECT                            4'h0
  // 0: CVBS on AIN1 (composite video in)
  // 7: Y on AIN2, C on AIN5 (s-video in)
  // (These are the only configurations supported by the 6.111 labkit hardware)
`define INPUT_MODE                              4'h0
  // 0: Autodetect: NTSC or PAL (BGHID), w/o pedestal
  // 1: Autodetect: NTSC or PAL (BGHID), w/pedestal
  // 2: Autodetect: NTSC or PAL (N), w/o pedestal
  // 3: Autodetect: NTSC or PAL (N), w/pedestal
  // 4: NTSC w/o pedestal
  // 5: NTSC w/pedestal
  // 6: NTSC 4.43 w/o pedestal
  // 7: NTSC 4.43 w/pedestal
  // 8: PAL BGHID w/o pedestal
  // 9: PAL N w/pedestal
  // A: PAL M w/o pedestal
  // B: PAL M w/pedestal
  // C: PAL combination N
  // D: PAL combination N w/pedestal
  // E-F: [Not valid]

`define ADV7185_REGISTER_0 {`INPUT_MODE, `INPUT_SELECT}

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

```
// Register 1
//////////////////////////////////////////////////////////////////////////////

`define VIDEO_QUALITY                          2'h0
  // 0: Broadcast quality
  // 1: TV quality
  // 2: VCR quality
  // 3: Surveillance quality
`define SQUARE_PIXEL_IN_MODE                   1'b0
  // 0: Normal mode
  // 1: Square pixel mode
`define DIFFERENTIAL_INPUT                     1'b0
  // 0: Single-ended inputs
  // 1: Differential inputs
`define FOUR_TIMES_SAMPLING                    1'b0
  // 0: Standard sampling rate
  // 1: 4x sampling rate (NTSC only)
`define BETACAM                                1'b0
  // 0: Standard video input
  // 1: Betacam video input
`define AUTOMATIC_STARTUP_ENABLE               1'b1
  // 0: Change of input triggers reacquire
  // 1: Change of input does not trigger reacquire

`define ADV7185_REGISTER_1 {`AUTOMATIC_STARTUP_ENABLE, 1'b0, `BETACAM, `FOUR_TIMES_SAMPLING,
`DIFFERENTIAL_INPUT, `SQUARE_PIXEL_IN_MODE, `VIDEO_QUALITY}

//////////////////////////////////////////////////////////////////////////////
// Register 2
//////////////////////////////////////////////////////////////////////////////

`define Y_PEAKING_FILTER                       3'h4
  // 0: Composite =  4.5dB,  s-video =  9.25dB
  // 1: Composite =  4.5dB,  s-video =  9.25dB
  // 2: Composite =  4.5dB,  s-video =  5.75dB
  // 3: Composite =  1.25dB, s-video =  3.3dB
  // 4: Composite =  0.0dB,  s-video =  0.0dB
  // 5: Composite = -1.25dB, s-video = -3.0dB
  // 6: Composite = -1.75dB, s-video = -8.0dB
  // 7: Composite = -3.0dB,  s-video = -8.0dB
`define CORING                                 2'h0
  // 0: No coring
  // 1: Truncate if Y < black+8
  // 2: Truncate if Y < black+16
  // 3: Truncate if Y < black+32

`define ADV7185_REGISTER_2 {3'b000, `CORING, `Y_PEAKING_FILTER}

//////////////////////////////////////////////////////////////////////////////
// Register 3
//////////////////////////////////////////////////////////////////////////////

`define INTERFACE_SELECT                       2'h0
  // 0: Philips-compatible
  // 1: Broktree API A-compatible
  // 2: Broktree API B-compatible
  // 3: [Not valid]
`define OUTPUT_FORMAT                          4'h0
  // 0: 10-bit @ LLC, 4:2:2 CCIR656
  // 1: 20-bit @ LLC, 4:2:2 CCIR656
  // 2: 16-bit @ LLC, 4:2:2 CCIR656
  // 3: 8-bit @ LLC, 4:2:2 CCIR656
  // 4: 12-bit @ LLC, 4:1:1
  // 5-F: [Not valid]
  // (Note that the 6.111 labkit hardware provides only a 10-bit interface to
  // the ADV7185.)
`define TRISTATE_OUTPUT_DRIVERS                1'b0
  // 0: Drivers tristated when ~OE is high
  // 1: Drivers always tristated
`define VBI_ENABLE                             1'b0
  // 0: Decode lines during vertical blanking interval
  // 1: Decode only active video regions
```

```
`define ADV7185_REGISTER_3 {`VBI_ENABLE, `TRISTATE_OUTPUT_DRIVERS, `OUTPUT_FORMAT, `INTERFACE_SELECT}

////////////////////////////////////////////////////////////////////////////
// Register 4
////////////////////////////////////////////////////////////////////////////

`define OUTPUT_DATA_RANGE                       1'b0
  // 0: Output values restricted to CCIR-compliant range
  // 1: Use full output range
`define BT656_TYPE                              1'b0
  // 0: BT656-3-compatible
  // 1: BT656-4-compatible

`define ADV7185_REGISTER_4 {`BT656_TYPE, 3'b000, 3'b110, `OUTPUT_DATA_RANGE}

////////////////////////////////////////////////////////////////////////////
// Register 5
////////////////////////////////////////////////////////////////////////////


`define GENERAL_PURPOSE_OUTPUTS                 4'b0000
`define GPO_0_1_ENABLE                          1'b0
  // 0: General purpose outputs 0 and 1 tristated
  // 1: General purpose outputs 0 and 1 enabled
`define GPO_2_3_ENABLE                          1'b0
  // 0: General purpose outputs 2 and 3 tristated
  // 1: General purpose outputs 2 and 3 enabled
`define BLANK_CHROMA_IN_VBI                     1'b1
  // 0: Chroma decoded and output during vertical blanking
  // 1: Chroma blanked during vertical blanking
`define HLOCK_ENABLE                            1'b0
  // 0: GPO 0 is a general purpose output
  // 1: GPO 0 shows HLOCK status

`define ADV7185_REGISTER_5 {`HLOCK_ENABLE, `BLANK_CHROMA_IN_VBI, `GPO_2_3_ENABLE, `GPO_0_1_ENABLE,
`GENERAL_PURPOSE_OUTPUTS}

////////////////////////////////////////////////////////////////////////////
// Register 7
////////////////////////////////////////////////////////////////////////////

`define FIFO_FLAG_MARGIN                        5'h10
  // Sets the locations where FIFO almost-full and almost-empty flags are set
`define FIFO_RESET                              1'b0
  // 0: Normal operation
  // 1: Reset FIFO. This bit is automatically cleared
`define AUTOMATIC_FIFO_RESET                    1'b0
  // 0: No automatic reset
  // 1: FIFO is autmatically reset at the end of each video field
`define FIFO_FLAG_SELF_TIME                     1'b1
  // 0: FIFO flags are synchronized to CLKIN
  // 1: FIFO flags are synchronized to internal 27MHz clock

`define ADV7185_REGISTER_7 {`FIFO_FLAG_SELF_TIME, `AUTOMATIC_FIFO_RESET, `FIFO_RESET, `FIFO_FLAG_MARGIN}

////////////////////////////////////////////////////////////////////////////
// Register 8
////////////////////////////////////////////////////////////////////////////

`define INPUT_CONTRAST_ADJUST                       8'h80

`define ADV7185_REGISTER_8 {`INPUT_CONTRAST_ADJUST}

////////////////////////////////////////////////////////////////////////////
// Register 9
////////////////////////////////////////////////////////////////////////////

`define INPUT_SATURATION_ADJUST                     8'h8C

`define ADV7185_REGISTER_9 {`INPUT_SATURATION_ADJUST}

////////////////////////////////////////////////////////////////////////////
// Register A
```

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

`define INPUT_BRIGHTNESS_ADJUST                      8'h00

`define ADV7185_REGISTER_A {`INPUT_BRIGHTNESS_ADJUST}

////////////////////////////////////////////////////////////////////////////
// Register B
////////////////////////////////////////////////////////////////////////////

`define INPUT_HUE_ADJUST                             8'h00

`define ADV7185_REGISTER_B {`INPUT_HUE_ADJUST}

////////////////////////////////////////////////////////////////////////////
// Register C
////////////////////////////////////////////////////////////////////////////

`define DEFAULT_VALUE_ENABLE                    1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values
`define DEFAULT_VALUE_AUTOMATIC_ENABLE          1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values if lock is lost
`define DEFAULT_Y_VALUE                         6'h0C
  // Default Y value

`define ADV7185_REGISTER_C {`DEFAULT_Y_VALUE, `DEFAULT_VALUE_AUTOMATIC_ENABLE, `DEFAULT_VALUE_ENABLE}

////////////////////////////////////////////////////////////////////////////
// Register D
////////////////////////////////////////////////////////////////////////////

`define DEFAULT_CR_VALUE                        4'h8
  // Most-significant four bits of default Cr value
`define DEFAULT_CB_VALUE                        4'h8
  // Most-significant four bits of default Cb value

`define ADV7185_REGISTER_D {`DEFAULT_CB_VALUE, `DEFAULT_CR_VALUE}

////////////////////////////////////////////////////////////////////////////
// Register E
////////////////////////////////////////////////////////////////////////////

`define TEMPORAL_DECIMATION_ENABLE              1'b0
  // 0: Disable
  // 1: Enable
`define TEMPORAL_DECIMATION_CONTROL             2'h0
  // 0: Supress frames, start with even field
  // 1: Supress frames, start with odd field
  // 2: Supress even fields only
  // 3: Supress odd fields only
`define TEMPORAL_DECIMATION_RATE                4'h0
  // 0-F: Number of fields/frames to skip

`define ADV7185_REGISTER_E {1'b0, `TEMPORAL_DECIMATION_RATE, `TEMPORAL_DECIMATION_CONTROL,
`TEMPORAL_DECIMATION_ENABLE}

////////////////////////////////////////////////////////////////////////////
// Register F
////////////////////////////////////////////////////////////////////////////

`define POWER_SAVE_CONTROL                      2'h0
  // 0: Full operation
  // 1: CVBS only
  // 2: Digital only
  // 3: Power save mode
`define POWER_DOWN_SOURCE_PRIORITY              1'b0
  // 0: Power-down pin has priority
  // 1: Power-down control bit has priority
`define POWER_DOWN_REFERENCE                    1'b0
  // 0: Reference is functional
  // 1: Reference is powered down
```

```
`define POWER_DOWN_LLC_GENERATOR                1'b0
  // 0: LLC generator is functional
  // 1: LLC generator is powered down
`define POWER_DOWN_CHIP                         1'b0
  // 0: Chip is functional
  // 1: Input pads disabled and clocks stopped
`define TIMING_REACQUIRE                        1'b0
  // 0: Normal operation
  // 1: Reacquire video signal (bit will automatically reset)
`define RESET_CHIP                              1'b0
  // 0: Normal operation
  // 1: Reset digital core and I2C interface (bit will automatically reset)

`define ADV7185_REGISTER_F {`RESET_CHIP, `TIMING_REACQUIRE, `POWER_DOWN_CHIP, `POWER_DOWN_LLC_GENERATOR,
`POWER_DOWN_REFERENCE, `POWER_DOWN_SOURCE_PRIORITY, `POWER_SAVE_CONTROL}

////////////////////////////////////////////////////////////////////////////////
// Register 33
////////////////////////////////////////////////////////////////////////////////

`define PEAK_WHITE_UPDATE                       1'b1
  // 0: Update gain once per line
  // 1: Update gain once per field
`define AVERAGE_BIRIGHTNESS_LINES               1'b1
  // 0: Use lines 33 to 310
  // 1: Use lines 33 to 270
`define MAXIMUM_IRE                             3'h0
  // 0: PAL: 133, NTSC: 122
  // 1: PAL: 125, NTSC: 115
  // 2: PAL: 120, NTSC: 110
  // 3: PAL: 115, NTSC: 105
  // 4: PAL: 110, NTSC: 100
  // 5: PAL: 105, NTSC: 100
  // 6-7: PAL: 100, NTSC: 100
`define COLOR_KILL                              1'b1
  // 0: Disable color kill
  // 1: Enable color kill

`define ADV7185_REGISTER_33 {1'b1, `COLOR_KILL, 1'b1, `MAXIMUM_IRE, `AVERAGE_BIRIGHTNESS_LINES,
`PEAK_WHITE_UPDATE}

`define ADV7185_REGISTER_10 8'h00
`define ADV7185_REGISTER_11 8'h00
`define ADV7185_REGISTER_12 8'h00
`define ADV7185_REGISTER_13 8'h45
`define ADV7185_REGISTER_14 8'h18
`define ADV7185_REGISTER_15 8'h60
`define ADV7185_REGISTER_16 8'h00
`define ADV7185_REGISTER_17 8'h01
`define ADV7185_REGISTER_18 8'h00
`define ADV7185_REGISTER_19 8'h10
`define ADV7185_REGISTER_1A 8'h10
`define ADV7185_REGISTER_1B 8'hF0
`define ADV7185_REGISTER_1C 8'h16
`define ADV7185_REGISTER_1D 8'h01
`define ADV7185_REGISTER_1E 8'h00
`define ADV7185_REGISTER_1F 8'h3D
`define ADV7185_REGISTER_20 8'hD0
`define ADV7185_REGISTER_21 8'h09
`define ADV7185_REGISTER_22 8'h8C
`define ADV7185_REGISTER_23 8'hE2
`define ADV7185_REGISTER_24 8'h1F
`define ADV7185_REGISTER_25 8'h07
`define ADV7185_REGISTER_26 8'hC2
`define ADV7185_REGISTER_27 8'h58
`define ADV7185_REGISTER_28 8'h3C
`define ADV7185_REGISTER_29 8'h00
`define ADV7185_REGISTER_2A 8'h00
`define ADV7185_REGISTER_2B 8'hA0
`define ADV7185_REGISTER_2C 8'hCE
`define ADV7185_REGISTER_2D 8'hF0
`define ADV7185_REGISTER_2E 8'h00
`define ADV7185_REGISTER_2F 8'hF0
```

```
`define ADV7185_REGISTER_30 8'h00
`define ADV7185_REGISTER_31 8'h70
`define ADV7185_REGISTER_32 8'h00
`define ADV7185_REGISTER_34 8'h0F
`define ADV7185_REGISTER_35 8'h01
`define ADV7185_REGISTER_36 8'h00
`define ADV7185_REGISTER_37 8'h00
`define ADV7185_REGISTER_38 8'h00
`define ADV7185_REGISTER_39 8'h00
`define ADV7185_REGISTER_3A 8'h00
`define ADV7185_REGISTER_3B 8'h00

`define ADV7185_REGISTER_44 8'h41
`define ADV7185_REGISTER_45 8'hBB

`define ADV7185_REGISTER_F1 8'hEF
`define ADV7185_REGISTER_F2 8'h80

module adv7185_init (reset, clock_27mhz, source, tv_in_reset_b,
            tv_in_i2c_clock, tv_in_i2c_data);

   input reset;
   input clock_27mhz;
   output tv_in_reset_b; // Reset signal to ADV7185
   output tv_in_i2c_clock; // I2C clock output to ADV7185
   output tv_in_i2c_data; // I2C data line to ADV7185
   input source; // 0: composite, 1: s-video

   initial begin
      $display("ADV7185 Initialization values:");
      $display("  Register 0:  0x%X", `ADV7185_REGISTER_0);
      $display("  Register 1:  0x%X", `ADV7185_REGISTER_1);
      $display("  Register 2:  0x%X", `ADV7185_REGISTER_2);
      $display("  Register 3:  0x%X", `ADV7185_REGISTER_3);
      $display("  Register 4:  0x%X", `ADV7185_REGISTER_4);
      $display("  Register 5:  0x%X", `ADV7185_REGISTER_5);
      $display("  Register 7:  0x%X", `ADV7185_REGISTER_7);
      $display("  Register 8:  0x%X", `ADV7185_REGISTER_8);
      $display("  Register 9:  0x%X", `ADV7185_REGISTER_9);
      $display("  Register A:  0x%X", `ADV7185_REGISTER_A);
      $display("  Register B:  0x%X", `ADV7185_REGISTER_B);
      $display("  Register C:  0x%X", `ADV7185_REGISTER_C);
      $display("  Register D:  0x%X", `ADV7185_REGISTER_D);
      $display("  Register E:  0x%X", `ADV7185_REGISTER_E);
      $display("  Register F:  0x%X", `ADV7185_REGISTER_F);
      $display("  Register 33: 0x%X", `ADV7185_REGISTER_33);
   end

   // Generate a 1MHz for the I2C driver (resulting I2C clock rate is 250kHz)
   reg [7:0] clk_div_count, reset_count;
   reg clock_slow;
   wire reset_slow;

   initial
     begin
     clk_div_count <= 8'h00;
     // synthesis attribute init of clk_div_count is "00";
     clock_slow <= 1'b0;
     // synthesis attribute init of clock_slow is "0";
      end

   always @(posedge clock_27mhz)
     if (clk_div_count == 26)
       begin
       clock_slow <= ~clock_slow;
       clk_div_count <= 0;
        end
     else
       clk_div_count <= clk_div_count+1;

   always @(posedge clock_27mhz)
     if (reset)
       reset_count <= 100;
```

```
  else
    reset_count <= (reset_count==0) ? 0 : reset_count-1;

assign reset_slow = reset_count != 0;

// I2C driver
reg load;
reg [7:0] data;
wire ack, idle;

i2c_interface i2c_interface(.reset(reset_slow), .clock4x(clock_slow), .data(data), .load(load),
    .ack(ack), .idle(idle), .scl(tv_in_i2c_clock),
    .sda(tv_in_i2c_data));

// State machine
reg [7:0] state;
reg tv_in_reset_b;
reg old_source;

always @(posedge clock_slow)
   if (reset_slow)
 begin
    state <= 0;
    load <= 0;
    tv_in_reset_b <= 0;
    old_source <= 0;
 end
   else
 case (state)
   8'h00:
     begin
        // Assert reset
        load <= 1'b0;
        tv_in_reset_b <= 1'b0;
        if (!ack)
      state <= state+1;
     end
   8'h01:
     state <= state+1;
   8'h02:
     begin
        // Release reset
        tv_in_reset_b <= 1'b1;
        state <= state+1;
             end
   8'h03:
     begin
        // Send ADV7185 address
        data <= 8'h8A;
        load <= 1'b1;
        if (ack)
      state <= state+1;
     end
   8'h04:
     begin
        // Send subaddress of first register
        data <= 8'h00;
        if (ack)
      state <= state+1;
     end
   8'h05:
     begin
        // Write to register 0
        data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
        if (ack)
      state <= state+1;
     end
   8'h06:
     begin
        // Write to register 1
        data <= `ADV7185_REGISTER_1;
        if (ack)
      state <= state+1;
```

```
  end
8'h07:
  begin
     // Write to register 2
     data <= `ADV7185_REGISTER_2;
     if (ack)
   state <= state+1;
  end
8'h08:
  begin
     // Write to register 3
     data <= `ADV7185_REGISTER_3;
     if (ack)
   state <= state+1;
  end
8'h09:
  begin
     // Write to register 4
     data <= `ADV7185_REGISTER_4;
     if (ack)
   state <= state+1;
  end
8'h0A:
  begin
     // Write to register 5
     data <= `ADV7185_REGISTER_5;
     if (ack)
   state <= state+1;
  end
8'h0B:
  begin
     // Write to register 6
     data <= 8'h00; // Reserved register, write all zeros
     if (ack)
   state <= state+1;
  end
8'h0C:
  begin
     // Write to register 7
     data <= `ADV7185_REGISTER_7;
     if (ack)
   state <= state+1;
  end
8'h0D:
  begin
     // Write to register 8
     data <= `ADV7185_REGISTER_8;
     if (ack)
   state <= state+1;
  end
8'h0E:
  begin
     // Write to register 9
     data <= `ADV7185_REGISTER_9;
     if (ack)
   state <= state+1;
  end
8'h0F: begin
    // Write to register A
    data <= `ADV7185_REGISTER_A;
 if (ack)
    state <= state+1;
end
8'h10:
  begin
     // Write to register B
     data <= `ADV7185_REGISTER_B;
     if (ack)
   state <= state+1;
  end
8'h11:
  begin
     // Write to register C
```

```
      data <= `ADV7185_REGISTER_C;
      if (ack)
    state <= state+1;
   end
 8'h12:
   begin
      // Write to register D
      data <= `ADV7185_REGISTER_D;
      if (ack)
    state <= state+1;
   end
 8'h13:
   begin
      // Write to register E
      data <= `ADV7185_REGISTER_E;
      if (ack)
    state <= state+1;
   end
 8'h14:
   begin
      // Write to register F
      data <= `ADV7185_REGISTER_F;
      if (ack)
    state <= state+1;
   end
 8'h15:
   begin
      // Wait for I2C transmitter to finish
      load <= 1'b0;
      if (idle)
    state <= state+1;
   end
 8'h16:
   begin
      // Write address
      data <= 8'h8A;
      load <= 1'b1;
      if (ack)
    state <= state+1;
   end
 8'h17:
   begin
      data <= 8'h33;
      if (ack)
    state <= state+1;
   end
 8'h18:
   begin
      data <= `ADV7185_REGISTER_33;
      if (ack)
    state <= state+1;
   end
 8'h19:
   begin
      load <= 1'b0;
      if (idle)
    state <= state+1;
   end

 8'h1A: begin
    data <= 8'h8A;
    load <= 1'b1;
    if (ack)
      state <= state+1;
 end
 8'h1B:
   begin
      data <= 8'h33;
      if (ack)
    state <= state+1;
   end
 8'h1C:
   begin
```

```
            load <= 1'b0;
            if (idle)
           state <= state+1;
         end
       8'h1D:
         begin
            load <= 1'b1;
            data <= 8'h8B;
            if (ack)
           state <= state+1;
         end
       8'h1E:
         begin
            data <= 8'hFF;
            if (ack)
           state <= state+1;
         end
       8'h1F:
         begin
            load <= 1'b0;
            if (idle)
           state <= state+1;
         end
       8'h20:
         begin
            // Idle
            if (old_source != source) state <= state+1;
            old_source <= source;
         end
       8'h21: begin
          // Send ADV7185 address
          data <= 8'h8A;
          load <= 1'b1;
          if (ack) state <= state+1;
       end
       8'h22: begin
          // Send subaddress of register 0
          data <= 8'h00;
          if (ack) state <= state+1;
       end
       8'h23: begin
          // Write to register 0
          data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
          if (ack) state <= state+1;
       end
       8'h24: begin
          // Wait for I2C transmitter to finish
          load <= 1'b0;
          if (idle) state <= 8'h20;
       end
        endcase

endmodule


// end of file rtl/video_acquisition/adv7185_init.v


// start of file rtl/video_acquisition/xvga_timing_generator.v

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

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

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

```
   reg [9:0] vcount;     // line number

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

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

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

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

      blank <= next_vblank | (next_hblank & ~hreset);
   end
endmodule
// end of file rtl/video_acquisition/xvga_timing_generator.v


// start of file rtl/video_acquisition/zbt_pipelined_interface.v
// Author: I. Chuang <ichuang@mit.edu>
//
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user.  The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data hold
// times around the clock positive edge to work reliably.
//

////////////////////////////////////////////////////////////////////////////////
// Ike's simple ZBT RAM driver for the MIT 6.111 labkit
//
// Data for writes can be presented and clocked in immediately; the actual
// writing to RAM will happen two cycles later.
//
// Read requests are processed immediately, but the read data is not available
// until two cycles after the intial request.
//
// A clock enable signal is provided; it enables the RAM clock when high.

module zbt_pipelined_interface(clk, cen, we, read_address, write_address, write_data, read_data,
        ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

   input clk;            // system clock
   input cen;            // clock enable for gating ZBT cycles
   input we;            // write enable (active HIGH)
   //input [18:0] addr;        // memory address
   input [18:0] read_address;
   input [18:0] write_address;
   input [35:0] write_data;    // data to write
   output [35:0] read_data;    // data read from memory
   output      ram_clk;    // physical line to ram clock
   output      ram_we_b;    // physical line to ram we_b
   output [18:0] ram_address;    // physical line to ram address
   inout [35:0]  ram_data;    // physical line to ram data
   output      ram_cen_b;    // physical line to ram clock enable
```

```
    wire [18:0] addr = we ? write_address : read_address;

   // clock enable (should be synchronous and one cycle high at a time)
   wire      ram_cen_b = ~cen;

   // create delayed ram_we signal: note the delay is by two cycles!
   // ie we present the data to be written two cycles after we is raised
   // this means the bus is tri-stated two cycles after we is raised.

   reg [1:0]   we_delay;

   always @(posedge clk)
     we_delay <= cen ? {we_delay[0],we} : we_delay;

   // create two-stage pipeline for write data

   reg [35:0]  write_data_old1;
   reg [35:0]  write_data_old2;
   always @(posedge clk)
     if (cen)
       {write_data_old2, write_data_old1} <= {write_data_old1, write_data};

   // wire to ZBT RAM signals

   assign      ram_we_b = ~we;
   assign      ram_clk = 1'b0;  // gph 2011-Nov-10
                                // set to zero as place holder

   assign      ram_address = addr;

   assign      ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};
   assign      read_data = ram_data;

endmodule // zbt_pipelined_interface
// end of file rtl/video_acquisition/zbt_pipelined_interface.v


// start of file rtl/infrared/sirc_player.v
module sirc_player(
    input reset,
    input clock,
    input time_base_75us,
    // carrier signal present on output pin (IR LED) when transmitter is active
    input carrier,
    // when asserted high, player registers input value and starts transmitting
    input start,
    // command field of the SIRC protocol
    input [6:0] sirc_command,
    // address field of the SIRC protocol
    input [4:0] sirc_address,
    // when high, ir transmitter is active
    output tx_enable,
    // IR LED output, conjunction of tx enable and carrier wave
    output ir_led,

    // debug
    output [2:0] current_state
);
    parameter SIRC_LENGTH = 12;

    `include "sirc_player_states.v"

    reg [2:0] state, next_state;

    // register value to be transmitted when state is LOAD_TX_DATA
    reg [SIRC_LENGTH-1:0] tx_buffer;
    always @(posedge clock) begin
        if (reset) begin
            tx_buffer <= 0;
        end else if (state == PST_LOAD_TX_BUFFER) begin
            tx_buffer <= { sirc_address, sirc_command };
```

```
    end
end

// keep track of the number of bits sent
wire current_bit;
reg [3:0] sent_bits_count;
always @(posedge clock) begin
    if (reset) begin
        sent_bits_count <= 0;
    end else begin
        if (state == PST_LOAD_TX_BUFFER)
            sent_bits_count <= 0;

        if (state == PST_DATA_SPACE && next_state == PST_DATA_MARK) begin
            if (sent_bits_count < SIRC_LENGTH - 1)
                sent_bits_count <= sent_bits_count + 1;
        end
    end
end

assign current_bit = tx_buffer[sent_bits_count];

wire next_bit;
assign next_bit =
    (state == PST_START_SPACE) ? tx_buffer[0] :
    (sent_bits_count < SIRC_LENGTH - 1) ? tx_buffer[sent_bits_count + 1] :
    0;


// timer counter to delay FSM transition into the next state
// timer inputs
wire i_timer_start;
wire [5:0] i_timer_value_max;
// timer internal state
reg [5:0] timer_value;
reg [5:0] timer_value_max;
// timer outputs
reg o_timer_expired;

always @(posedge clock) begin
    if (reset) begin
        timer_value <= 0;
        timer_value_max <= 0;
        o_timer_expired <= 0;
    end else if (i_timer_start) begin
        timer_value <= 0;
        timer_value_max <= i_timer_value_max - 1;
        o_timer_expired <= 0;
    end else if (time_base_75us) begin
        if (timer_value < timer_value_max) begin
            timer_value <= timer_value + 1;
        end else begin
            o_timer_expired <= 1;
        end
    end
end

assign i_timer_start = (state != next_state);
assign i_timer_value_max =
    (next_state == PST_START_MARK)  ? 32 :
    (next_state == PST_START_SPACE) ?  8 :
    (next_state == PST_DATA_MARK)   ? (next_bit ? 16 : 8) :
    (next_state == PST_DATA_SPACE)  ?  8 :
    0;

always @(*) begin
    next_state = state;
    case (state)
        PST_IDLE: begin
            if (start) next_state = PST_LOAD_TX_BUFFER;
        end
        PST_LOAD_TX_BUFFER: begin
            next_state = PST_SYNC_75us_PULSE;
```

```
            end
            PST_SYNC_75us_PULSE: begin
                if (time_base_75us) next_state = PST_START_MARK;
            end
            PST_START_MARK: begin
                if (o_timer_expired) next_state = PST_START_SPACE;
            end
            PST_START_SPACE: begin
                if (o_timer_expired) next_state = PST_DATA_MARK;
            end
            PST_DATA_MARK: begin
                if (o_timer_expired) next_state = PST_DATA_SPACE;
            end
            PST_DATA_SPACE: begin
                if (o_timer_expired)
                    next_state = (sent_bits_count < SIRC_LENGTH-1) ? PST_DATA_MARK : PST_IDLE;
            end
            default: begin
                next_state = PST_IDLE;
            end
        endcase
    end

    // effectuate state changes
    always @(posedge clock) begin
        if (reset) begin
            state <= PST_IDLE;
        end else begin
            state <= next_state;
            if (state != next_state) $display("player state changing: %0d --> %0d", state, next_state);
        end
    end

    assign tx_enable = (state == PST_DATA_MARK || state == PST_START_MARK);
    assign ir_led = tx_enable ? carrier : 0;

    // debug
    assign current_state = state;

endmodule
// end of file rtl/infrared/sirc_player.v


// start of file rtl/infrared/sirc_player_states.v
parameter PST_IDLE = 0;
parameter PST_LOAD_TX_BUFFER = 1;
parameter PST_START_MARK = 2;
parameter PST_START_SPACE = 3;
parameter PST_DATA_MARK = 4;
parameter PST_DATA_SPACE = 5;
parameter PST_SYNC_75us_PULSE = 6;
// end of file rtl/infrared/sirc_player_states.v


// start of file rtl/infrared/infrared_subsystem.v
`default_nettype none

module infrared_subsystem(
    input clock,
    input reset,
    input start,
    input [6:0] ir_command,
    input [4:0] ir_address,
    output ir_led
);
    `include "../common/global_definitions.v"

    ////////////////////////////////////////////////////////////////////////
    // Instantiate clock dividers to generate time bases used by system modules
    ////////////////////////////////////////////////////////////////////////

    wire time_base_75us;
```

```
    time_base_generator #(
        .CLOCK_FREQUENCY_IN_HZ   (CLOCK_FREQUENCY        ),
        .TIME_BASE_PERIOD_IN_US  (75                     )
    ) clock_divider_75us (
        .clock                   (clock                  ),
        .reset                   (reset                  ),
        .time_base               (time_base_75us         )
    );

    ////////////////////////////////////////////////////////////////////////////
    // Create the 40 kHz carrier for the Tx signal
    ////////////////////////////////////////////////////////////////////////////

    // 40 kHz carrier for the modulated signal
    wire tx_carrier_40kHz;
    // counter to generate the 40 kHz carrier
    reg [10:0] carrier_40kHz_counter;

    always @(posedge clock) begin
        if (reset)
            carrier_40kHz_counter <= 0;
        else
            //carrier_40kHz_counter <= (carrier_40kHz_counter == 674) ? 0 : carrier_40kHz_counter + 1;
            carrier_40kHz_counter <= (carrier_40kHz_counter == 1620) ? 0 : carrier_40kHz_counter + 1;
    end

    //assign tx_carrier_40kHz = (carrier_40kHz_counter < 169);
    assign tx_carrier_40kHz = (carrier_40kHz_counter < 406);

    ////////////////////////////////////////////////////////////////////////////
    // Instantiate player of Sony Infrared commands
    ////////////////////////////////////////////////////////////////////////////

    wire tx_enable;

    sirc_player sirc_player(
        .reset(reset),
        .clock(clock),
        .time_base_75us(time_base_75us),
        .carrier(tx_carrier_40kHz),
        .start(start),
        .sirc_command(ir_command),
        .sirc_address(ir_address),
        .tx_enable(tx_enable),
        .ir_led(ir_led),
        .current_state()
    );

endmodule
// end of file rtl/infrared/infrared_subsystem.v


// start of file rtl/common/time_base_generator.v
////////////////////////////////////////////////////////////////////////////////
// Clock divider to provide a 1 Hz time base for the anti-theft system.
// The base pulse is active only one clock cycle per second.
////////////////////////////////////////////////////////////////////////////////
module time_base_generator #(
    parameter CLOCK_FREQUENCY_IN_HZ = 27_000_000,
    parameter TIME_BASE_PERIOD_IN_US = 75,
    parameter SPEED_UP_FACTOR = 1
)(
    input clock, reset,
    // output time base pulse, active one clock cycle
    output reg time_base
);

    // calculate initial valuer for the counter
    parameter INITIAL_COUNT =
        TIME_BASE_PERIOD_IN_US * (CLOCK_FREQUENCY_IN_HZ / 1_000_000) / SPEED_UP_FACTOR;

    // implement counter behavior
    reg [31:0] count_value;
```

```
    always @(posedge clock) begin
        if (reset) begin
            count_value = INITIAL_COUNT;
        end else begin
            if (count_value > 0)
                count_value = count_value - 1;
            else
                count_value = INITIAL_COUNT - 1;

            //$display("count_value: %0d", count_value);
            //$display("INITIAL_COUNT: %0d", INITIAL_COUNT);
        end

        time_base = (count_value == 0);
    end

endmodule
// end of file rtl/common/time_base_generator.v


// start of file rtl/common/global_definitions.v
// global system clock in Hz
parameter CLOCK_FREQUENCY = 64_800_000;

parameter XVGA_H_RES = 1024;
parameter XVGA_V_RES = 768;

// horizontal resolution; # of columns in an NTSC frame
parameter NTSC_H_RES = 720;
// vertical resolution; # of lines in an NTSC frame
parameter NTSC_V_RES = 522;

// define a bounding box for pixels considered in the center of mass calculation
parameter CENTER_OF_MASS_X_MIN = 15; //51
parameter CENTER_OF_MASS_X_MAX = 710;//690
parameter CENTER_OF_MASS_Y_MIN = 25; //47
parameter CENTER_OF_MASS_Y_MAX = 500;//492

parameter MINIMUM_RED_PIXELS_COUNT_FOR_DETECTION = 300;
parameter MINIMUM_GREEN_PIXELS_COUNT_FOR_DETECTION = 400;
// end of file rtl/common/global_definitions.v


// start of file rtl/user_interface_and_control.v
module user_interface_and_control(
    input clock,
    input reset,
    // 1 when a gesture has been detected by the gesture recognition module
    input gesture_detected,
    // gesture ID of the current gesture; valid when gesture_detected = 1
    input [7:0] gesture_id,
    // start pulse for the audio subsystem
    output audio_start,
    // sound ID of the audio file to be played; asserted when audio_start = 1
    output [7:0] sound_select,
    // start pulse for the infrared subsystem
    output ir_start,
    // number of the IR command to be transmitted; asserted when ir_start = 1
    output [6:0] ir_command,
    // address of the IR command to be transmitted; asserted when ir_start = 1
    output [4:0] ir_address,
    output [31:0] debug_data
);

    `include "../common/global_definitions.v"

    parameter UICST_IDLE = 0;
    parameter UICST_START_AUDIO = 1;
    parameter UICST_SEND_IR_START = 2;
    parameter UICST_SEND_IR_COMMAND = 3;
    parameter UICST_WAIT_IR_INTERVAL = 4;

    // generate a 45ms timebase to repeat the same IR command many times
```

```
wire time_base_45ms;
wire time_base_45ms_reset;
time_base_generator #(
    .CLOCK_FREQUENCY_IN_HZ(CLOCK_FREQUENCY),
    .TIME_BASE_PERIOD_IN_US(45_000)
) time_base_generator_45ms (
    .clock(clock),
    .reset(time_base_45ms_reset),
    .time_base(time_base_45ms)
);

reg [3:0] current_state, next_state;
// repetitions counter to send the same RI command many times
reg [7:0] ir_command_repeat_counter;
// true when fsm will assume a new state on clock edge
wire state_change_due = (current_state != next_state);

// state transition logic
always @(*) begin
    next_state = current_state;
    case (current_state)
        UICST_IDLE:
            if (gesture_detected) next_state = UICST_START_AUDIO;
        UICST_START_AUDIO:
            next_state = UICST_SEND_IR_START;
        UICST_SEND_IR_START:
            next_state = UICST_SEND_IR_COMMAND;
        UICST_SEND_IR_COMMAND:
            if (ir_command_repeat_counter < 10)
                next_state = UICST_WAIT_IR_INTERVAL;
            else
                next_state = UICST_IDLE;
        UICST_WAIT_IR_INTERVAL:
            // TODO: watch out for glitches...
            if (time_base_45ms)
                next_state = UICST_SEND_IR_START;
        default:
            next_state = UICST_IDLE;
    endcase
end

// assume next state
always @(posedge clock) current_state <= next_state;

// synchronize 45 ms pulses with the sending of IR commands
assign time_base_45ms_reset = (current_state == UICST_SEND_IR_START);

reg [7:0] detected_gesture_id;
always @(posedge clock) begin
    if (reset) begin
        detected_gesture_id <= 0;
        ir_command_repeat_counter <= 0;
    end else begin
        // register gesture ID for handing it over to other modules in future states
        if ((current_state == UICST_IDLE) && state_change_due)
            detected_gesture_id <= gesture_id;

        // increment repetitions counter during idle state
        if (current_state == UICST_IDLE)
            ir_command_repeat_counter <= 0;
        // increment repetitions counter upon entering wait state
        else if (state_change_due && (next_state == UICST_WAIT_IR_INTERVAL))
            ir_command_repeat_counter <= ir_command_repeat_counter + 1;
    end
end

// produce audio start pulse
assign audio_start = (current_state == UICST_START_AUDIO);
// audio file ID is the gesture ID of the last detected gesture
assign sound_select = detected_gesture_id;

// produce IR start pulse
assign ir_start = (current_state == UICST_SEND_IR_START);
```

```
    // hardcoded address for the TV
    assign ir_address = 7'd1;
    // determine IR command corresponding to the last detected gesture
    assign ir_command =
        (detected_gesture_id == 0) ? 21 : // TV power
        (detected_gesture_id == 1) ? 20 : // TV mute
        (detected_gesture_id == 2) ? 18 : // TV volume +
        (detected_gesture_id == 3) ? 19 : // TV volume -
        (detected_gesture_id == 4) ? 16 : // 7'b000_0000 : // TV channel +
        (detected_gesture_id == 5) ? 17 : // 7'b111_1111 : // TV channel -
        ir_command;
endmodule
// end of file rtl/user_interface_and_control.v
```