Design Document: Screaming Bird

Yiran Hu (yh3639)

Yuesheng Ma (ym2976)

Yang Li (yl5456)

Chenyang Zhou(cz2791)

Table of Contents

Introduction 2
Block Diagram 3
Data Flow 4
- 1. Microphone/Speaker ↔ Board 4
- 2. Audio CODEC ↔ CODEC Interface 4
- 3. CODEC Interface ↔ FIR Filter/Audio Rom 4
- 4. FIR Filter/Button/Audio Rom ↔ Audio Interface Module 4
- 5. Audio/VGA Interface Module ↔ ARM Core 4
- 6. Audio Driver ↔ Game Logic 5
- 7. Game Logic → VGA Driver 5
- 8. 9. 10. VGA Data flow 5

Hardware 6
- CODEC Interface IP 6
- FIR Filter 9
- Audio Controller (Audio Interface Module) 11
- Audio Rom 13
- VGA Timing 13
- BRAM 15
- VGA Controller (Main Logic) 16
- Sprite Bitmaps 19
- ROM 20
- Color Palette 21
- Qsys--System Connections 22

Software 23
- Game Logic Module 23

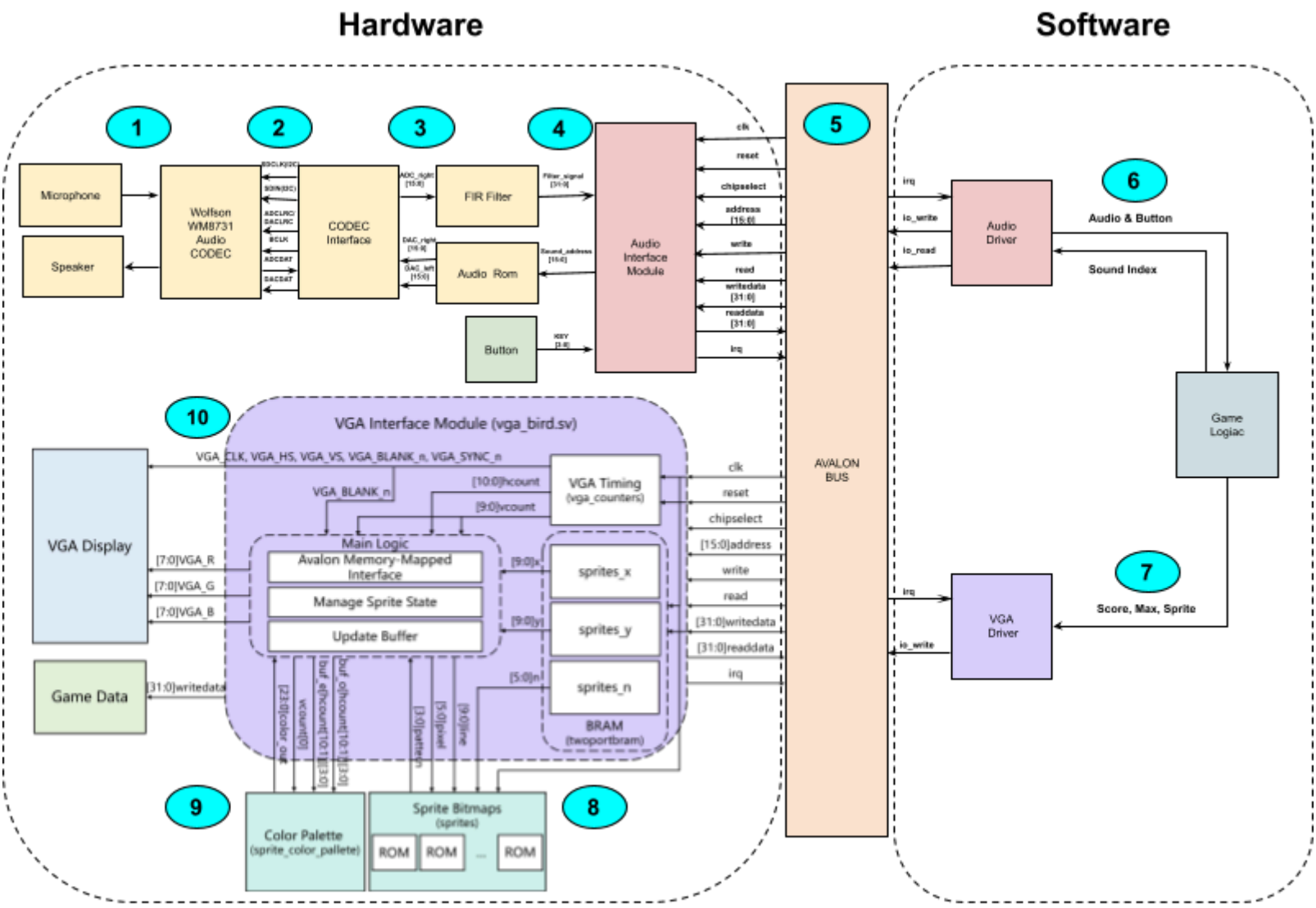
Resource Budget 27
Hardware/Software Interface 28
- Audio Driver 29
- VGA Driver 29
- Registers 30

Milestones 31
Group member contributions 31
Future work 31
Complete listing of Project Files 32
References 33
Code 34
- Hardware 34
- Software 59

Introduction

Screaming Bird is a voice control game based on Flappy Bird, where players only need to use one finger to control the flight status of the bird, allowing it to avoid uneven pipes along the way. In our design, we will replace finger manipulation with sound and use sound to control the rise and fall of birds. When a sound exceeding the threshold is detected, the bird will rise, otherwise it will naturally fall.

Block Diagram



This block diagram provides an overview of the system, which consists of two principal components: hardware and software. The hardware section incorporates an Audio Codec, VGA, and button interface on the board, which manage the input/output of audio data and button control signals, and output display information through the Avalon bus. The software component is responsible for processing the inputs received from the hardware, executing the game logic, and transmitting the display and sound content back to the hardware. Detailed descriptions of the entire process are provided in the subsequent sections.

Data Flow

In this section, we will explore each data transition illustrated in the system block diagram. Each number corresponds to a label in the diagram, and we will describe the data flow and modifications associated with each labeled transition.

1. Microphone/Speaker ↔ Board

The DE1-SoC features the Wolfson WM8731 audio CODEC, which includes microphone input port and audio output port. We will connect a Hiyanco microphone to this mic in port, serving as our external audio source. As for the speaker, we connect it to the line out port.

2. Audio CODEC ↔ CODEC Interface

The Wolfson audio CODEC is configured and controlled through the CODEC Interface IP *AudioCodecDrivers*, primarily developed by Prof. Scott Hauck at Washington University [1]. It utilizes I2C protocols (SDIN and SDCLK) to configure the Audio CODEC and then uses ADCCLK/DACCLK, BLK, ADCDAT/DACDAT lines to manage reading and writing sound data to the Audio CODEC. The details are clarified in the Hardware section.

3. CODEC Interface ↔ FIR Filter/Audio Rom

The CODEC Interface features four 16-bit registers, each configured to store a single left and right audio sample for both the ADC and DAC. Every 1/48k seconds, data from the right channel of the microphone's ADC is transmitted to the FIR Filter Module for further processing. At the same time, the DAC receives sound data for both the right and left channels from the Audio ROM.

4. FIR Filter/Button/Audio Rom ↔ Audio Interface Module

The output from the FIR filter, stored in 32-bit registers, along with the 4-bit button state, is forwarded to the Audio Interface Module. The Audio Interface Module also sends the 16-bit sound address to the Audio Rom to localize the corresponding audio.

5. Audio/VGA Interface Module ↔ ARM Core

The Audio and VGA Interface modules communicate with the ARM Core via the Avalon Bus, which acts as a bridge between the software running on the HPS and the FPGA hardware. The device tree provides essential configuration details for these interfaces, specifying resource allocation, memory mapping, and interrupt handling, ensuring efficient management of the Audio and VGA devices.

Regarding the audio, the Audio Interface Module within the hardware exchanges 32-bit data with the software audio driver. This audio driver reads the 32-bit filtered signal and 32-bit button signal(which is padded with 28 zeros) and writes the 32-bit sound index through the Avalon Bus. The details of these interface registers are illustrated in the Hardware/Software Interface section.

For the VGA display part, the hardware exchanges 32 bit data with software VGA driver. The software sends 32 bit long data consisting of sprite ID, location and index. Then, the hardware can draw them on screen.

6. Audio Driver ↔ Game Logic

After the Audio Driver receives the filters signals and the button states, the Game Logic module processes these signals to control bird and menu selection. Meanwhile, the Game Logic also sends the sound index(including 0, 1 and 2 which refer to stop, flap, and hit sounds respectively) to the audio driver.

7. Game Logic → VGA Driver

The game logic will have a menu, setting and birds and pipes. This can be solved by sending sprite information from software to hardware VGA Driver. Thus, the screen can show these birds, pipes, characters and numbers. The non zero value of dx and dy will change the position of birds and pipes.

8. 9. 10. VGA Data flow

The VGA data flow begins with the vga_counters module generating horizontal (hcount) and vertical (vcount) counters to track the current pixel position based on VGA timing. The CPU writes sprite attributes (x, y coordinates, and note IDs) to dual-port BRAM via the Avalon bus interface. The main vga_bird module processes these sprite attributes, determining the current line within each sprite and updating display buffers (buf_e and buf_o) with the appropriate color codes. These buffers are then converted to RGB values using a color palette, and the RGB signals are synchronized with the VGA timing signals (VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n) to display the final image on the screen.

Hardware

CODEC Interface IP

The CODEC Interface IP manages Audio CODEC's operations through a series of programmable registers accessed via the I2C interface, specifically through the SDCLCK and SDIN lines. The CODEC is equipped with 11 registers, each mapped to specific addresses. These 11 registers are listed below and the details of R7 and R8 registers settings are shown in the following figures. All of this information is derived from the CODEC manual [2].

REGISTER	B 15	B 14	B 13	B 12	B 11	B 10	B 9	B8	B7	B6	B5	B4	B3	B2	B1	B0
R0 (00h)	0	0	0	0	0	0	0	LRIN BOTH	LIN MUTE	0	0	LINVOL				
R1 (02h)	0	0	0	0	0	0	1	RLIN BOTH	RIN MUTE	0	0	RINVOL				
R2 (04h)	0	0	0	0	0	1	0	LRHP BOTH	LZCEN	LHPVOL						
R3 (06h)	0	0	0	0	0	1	1	RLHP BOTH	RZCEN	RHPVOL						
R4 (08h)	0	0	0	0	1	0	0	0	SIDEATT		SIDETONE	DAC SEL	BY PASS	INSEL	MUTE MIC	MIC BOOST
R5 (0Ah)	0	0	0	0	1	0	1	0	0	0	0	HPOR	DAC MU	DEEMPH		ADC HPD
R6 (0Ch)	0	0	0	0	1	1	0	0	PWR OFF	CLK OUTPD	OSCPD	OUTPD	DACPD	ADCPD	MICPD	LINEINPD
R7 (0Eh)	0	0	0	0	1	1	1	0	BCLK INV	MS	LR SWAP	LRP	IWL		FORMAT	
R8 (10h)	0	0	0	1	0	0	0	0	CLKO DIV2	CLKI DIV2	SR				BOSR	USB/ NORM
R9 (12h)	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	ACTIVE
R15(1Eh)	0	0	0	1	1	1	1	RESET								
	ADDRESS							DATA								

REGISTER ADDRESS	BIT	LABEL	DEFAULT	DESCRIPTION
0000111 Digital Audio Interface Format	1:0	FORMAT[1:0]	10	Audio Data Format Select 11 = DSP Mode, frame sync + 2 data packed words 10 = I^2S Format, MSB-First left-1 justified 01 = MSB-First, left justified 00 = MSB-First, right justified
	3:2	IWL[1:0]	10	Input Audio Data Bit Length Select 11 = 32 bits 10 = 24 bits 01 = 20 bits 00 = 16 bits
	4	LRP	0	DACLRC phase control (in left, right or I^2S modes) 1 = Right Channel DAC data when DACLRC high 0 = Right Channel DAC data when DACLRC low (opposite phasing in I^2S mode) or DSP mode A/B select (in DSP mode only) 1 = MSB is available on 2nd BCLK rising edge after DACLRC rising edge 0 = MSB is available on 1st BCLK rising edge after DACLRC rising edge
	5	LRSWAP	0	DAC Left Right Clock Swap 1 = Right Channel DAC Data Left 0 = Right Channel DAC Data Right
	6	MS	0	Master Slave Mode Control 1 = Enable Master Mode 0 = Enable Slave Mode
	7	BCLKINV	0	Bit Clock Invert 1 = Invert BCLK 0 = Don't invert BCLK

REGISTER ADDRESS	BIT	LABEL	DEFAULT	DESCRIPTION	
0001000 Sampling Control	0	USB/ NORMAL	0	Mode Select 1 = USB mode (250/272fs) 0 = Normal mode (256/384fs)	
	1	BOSR	0	Base Over-Sampling Rate	
				USB Mode 0 = 250fs 1 = 272fs	Normal Mode 96/88.2kHz 0 = 256fs 1 = 128fs 1 = 384fs 1 = 192fs
	5:2	SR[3:0]	0000	ADC and DAC sample rate control; See USB Mode and Normal Mode Sample Rate sections for operation	

SAMPLING RATE		MCLK FREQUENCY	SAMPLE RATE REGISTER SETTINGS					DIGITAL FILTER TYPE
ADC	DAC							
kHz	kHz	MHz	BOSR	SR3	SR2	SR1	SR0	
48	48	12.288	0 (256fs)	0	0	0	0	1
		18.432	1 (384fs)	0	0	0	0	

Here, we mainly configure R7 to 00000001(set bits 1:0 to 01 for MSB-First, left-justified format; bits 3:2 to 00 for 16-bit mode; and bit 6 to 0 to set slave mode) and R8 to 00000010 (set bit 0 to 0 for Normal mode; bit 1 to 1 for 384fs; bits 5:2 to 0000 for ADC and DAC 48kHz sample rate) via SDCLCK and SDIN lines. The pertinent configuration details are outlined below.

```
parameter MIN_ROM_ADDRESS = 6'h00;
parameter MAX_ROM_ADDRESS = 6'h32;
parameter AUD_LINE_IN_LC  = 9'h01A;
parameter AUD_LINE_IN_RC  = 9'h01A;
parameter AUD_LINE_OUT_LC = 9'h07B;
parameter AUD_LINE_OUT_RC = 9'h07B;
parameter AUD_ADC_PATH = 9'h0F8;
parameter AUD_DAC_PATH = 9'h006;
parameter AUD_POWER  = 9'h000;
parameter AUD_DATA_FORMAT = 9'h001;//R7
parameter AUD_SAMPLE_CTRL = 9'h002;//R8
parameter AUD_SET_ACTIVE  = 9'h001;
```

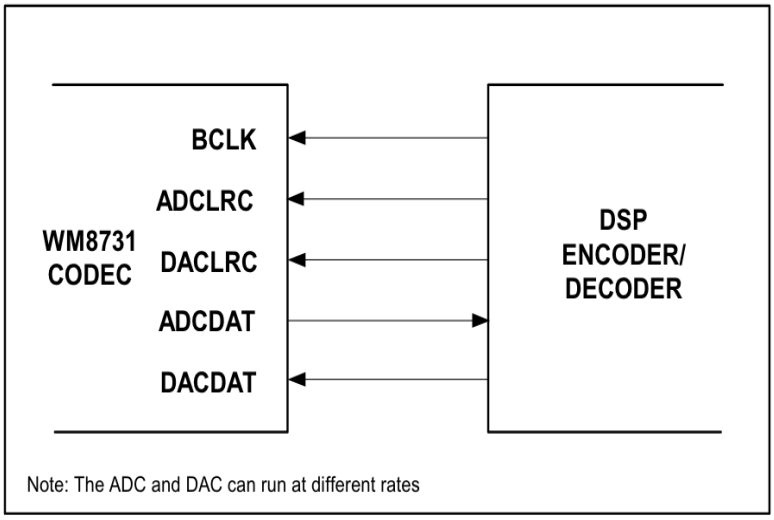
Once set up, the CODEC Interface employs the ADCLRC, BCLK, ADCDAT, and DACDAT lines to manage audio data transfer. It receives microphone input data and sends output data to the Audio CODEC. Given that the sample frequency for both ADC and DAC is set at 48kHz, the ADCLRC and DACLRC signals are derived by dividing the 50MHz system clock down to 48kHz using a clock divider.

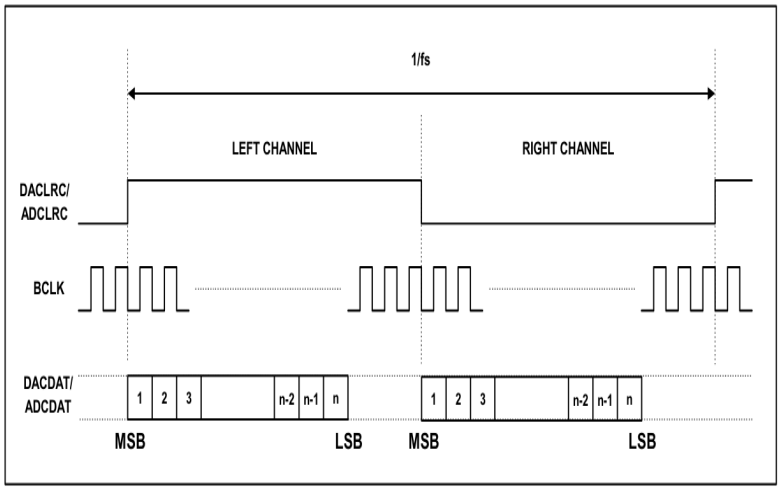
The timing diagram below reveals that in Left Justified mode, the MSB appears on the first rising edge of BCLK following an ADCLRC or DACLRC transition. This mode involves the digital audio interface processing data from both ADC and DAC digital filters.

ADCDAT serves as the output for the ADC's filtered audio data, consisting of left and right channels multiplexed together. The ADCLRC clock aligns this data, indicating whether it corresponds to the Left or Right channel. Both ADCDAT and ADCLRC are synchronized with BCLK, with each bit data transition marked by a change from high to low on BCLK.

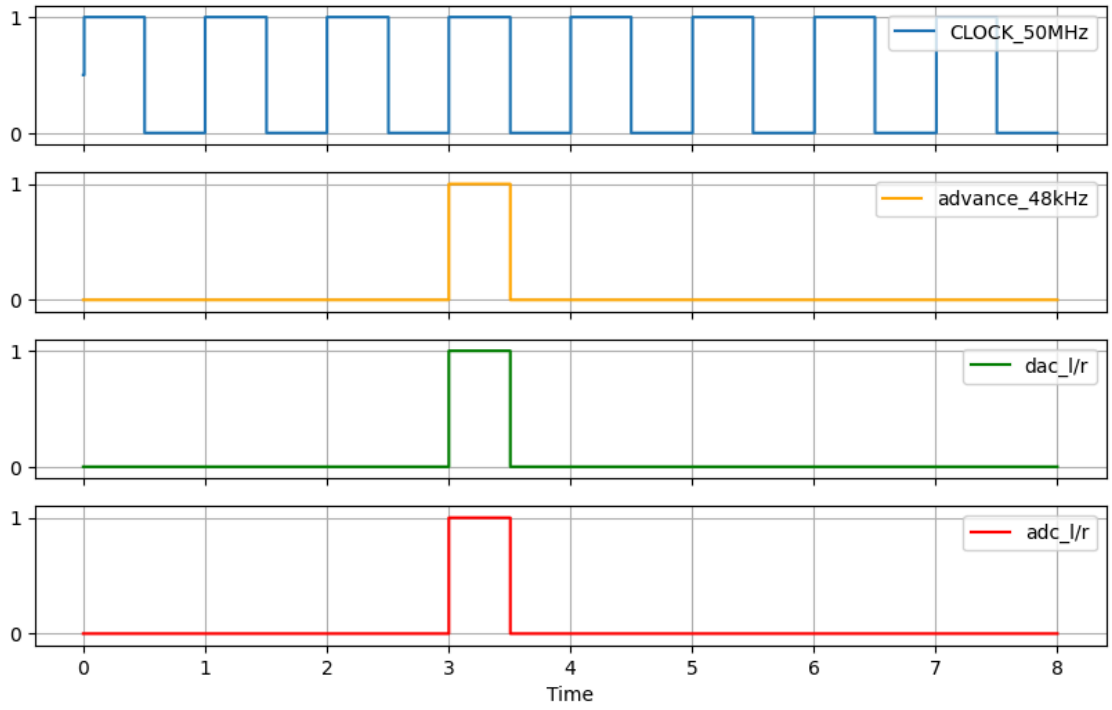
DACDAT, similarly, inputs multiplexed left and right audio data to the DAC filters. DACLRC performs an equivalent role to ADCLRC, aligning the data to indicate the channel being processed. DACDAT and DACLRC also synchronize with BCLK, with identical signaling for data transitions.

In both cases, the BCLK drives the timing for each bit data transition, ensuring smooth and synchronized audio data processing.





Thankfully, the CODEC Interface IP adeptly handles the intricacies of the underlying communication protocol, allowing us to focus on a select set of critical lines at the interface level: CLK, advance, ADC_left/right, and DAC_left/right.



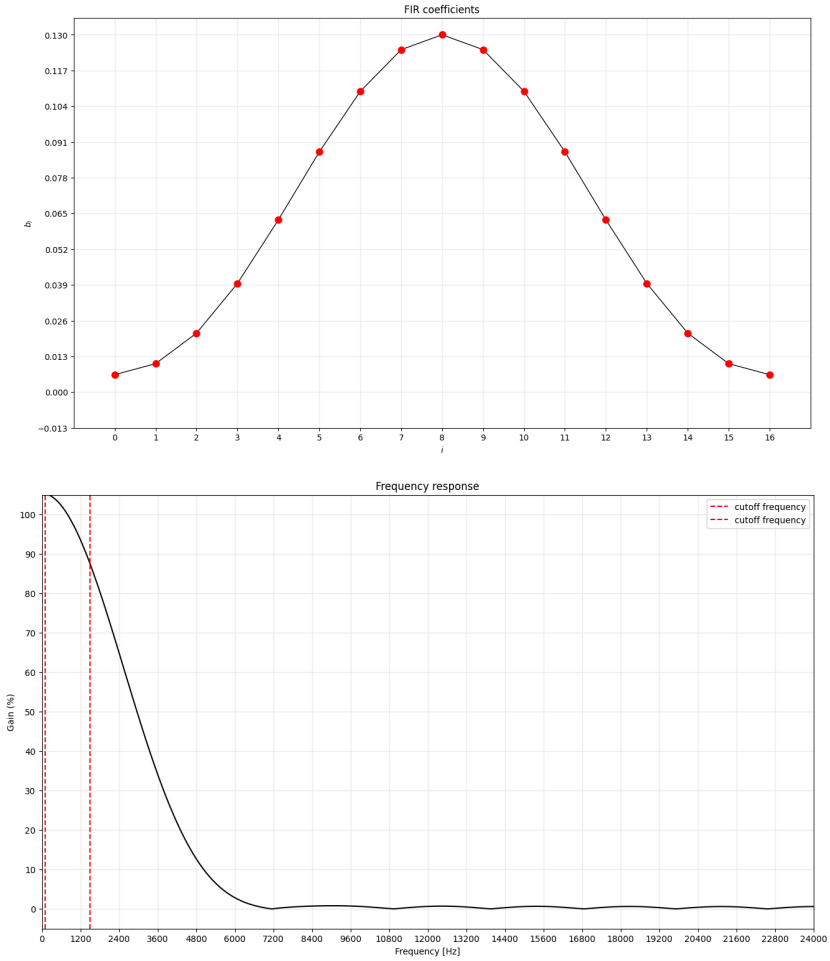
Here, all of these signals are synchronized. As the advance becomes 1, it indicates that it is ready to read the 16-bit audio input data from the microphone (ADCDAT) and send the 16-bit audio output data to the speaker (DACDAT).

FIR Filter

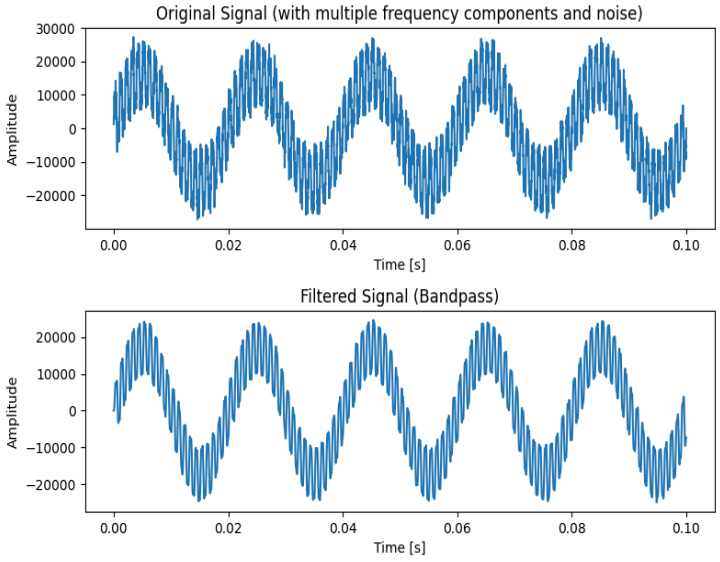
In this project, we designed and tested a hardcore 17-tap FIR bandpass filter from the range of [100Hz, 1500 Hz] with a sample rate of 48kHz. The main purpose of this filter is to isolate high frequencies from ambient environmental noise. The formulation for the FIR filter is as follows:

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k]$$

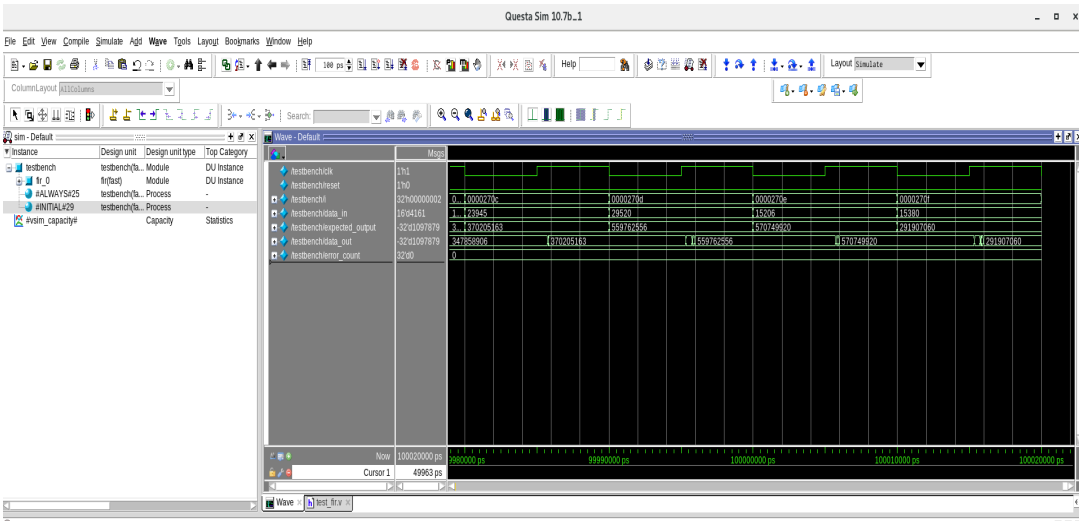
Here, N=17. We calculated the coefficient $h[k]$ of this FIR filter with the Python Package `firwin` [3]. The following figures display these 17 FIR coefficients and the corresponding frequency response. It is obvious that this filter mainly remains the relatively low frequency contents. To transfer the FIR filter coefficients to the FPGA, it's essential to normalize these coefficients from a 0-1 range to a 16-bit format. Consequently, the output signal from the filter will be represented in a 32-bit format.



Additionally, we conducted simulation tests on this FIR filter by inputting random signal data. The figure below provides a clear comparison of the audio signal before and after filtration. It is evident from the comparison that the filter effectively removes high frequency noises from the audio signal.



We created a testbench to evaluate our filter. The figure demonstrates that after filtering the data, the output aligns with the input data, successfully removing the undesired frequencies.



Audio Controller (Audio Interface Module)

The audio controller mainly takes charge in sending the filtered signals and button states and receiving sound index via Avalon bus. It also plays a role to match the speed between the software and the hardware for sound playing.

- **Avalon Memory-Mapped Interface Handler**

This function manages both reading from and writing to the Avalon memory-mapped interface. It guarantees that data transmitted by the Avalon master is accurately stored in the internal registers and memory.

```
always_ff @(posedge clk) begin
      if (chipselect && read) begin
           case (address)
             16'h0 : begin
                   readdata[31:0] <=  filtered_signal;
                   end
             16'h1 : begin
                   readdata[31:0] <=  {28'b0,KEY};
               end
           endcase
       end
      if (chipselect && write) begin
           case (address)
              16'h2:begin
                   sound<=writedata[1:0];
               end
           endcase
       end
   end
```

When the chipselect and write signals are high, it processes the address and writedata signals to update various internal registers:

Address	Function
16'h0	Loads the 32-bit filtered signals
16'h1	Loads the 4-bit KEY states padded with 28 zeros
16'h2	Updates the sound index

- **Speed Matching for Sound Playing**

A notable challenge we encountered was the mismatch in operating frequencies between the software and hardware. We resolved this issue by disregarding the software command if the previous audio was still in playback. In essence, we restart or stop the sound only when no sound is currently active. The related codes are below:

```
always_ff @(posedge clk) begin
  if (advance) begin
    // Detect if sound index has changed

    if (sound != prev_sound) begin
    if (play==0)begin

        case (sound)
            0:begin  //stop
               dac_left_in <= 16'd0;
               dac_right_in <= 16'd0;
               play<=0;
            end

            1: begin  //play flap sound
                sound_address <= 16'd0;
                play<=1;

            end
            2: begin //play hit sound
                sound_address <= 16'd18232;
                play<=1;
            end

        endcase

    end else begin
          sound_address <= sound_address + 1'b1;
      end

    end else begin
      if((sound_address!=16'd18231)&&(sound_address!=16'd46494))begin
        sound_address <= sound_address + 1'b1;
```

```
      end
    end

    // Update the previous sound value
    prev_sound <= sound;

    // only play once, the numbers are the end addresses for flap and hit
    if((sound_address==16'd18231)||(sound_address==16'd46494))begin
       dac_left_in <= 16'd0;
        dac_right_in <= 16'd0;
        play<=0;
     end

   // the sound stops only when no sound is playing
    if ((sound == 0)&& (play==0))begin
        dac_left_in <= 16'd0;
        dac_right_in <= 16'd0;
    // sending the sound data to the CODEC Interface IP
    end else begin
        dac_left_in <= sound_data;
        dac_right_in <= sound_data;

    end
  end
end
```

Audio Rom

To generate the sound effects of a bird flapping and hitting, we sourced the necessary .mp3 audio files from an open-source webpage [4]. These two .mp3 files were then merged and converted into a single .mif file which is structured into two primary sections: the sound address and the sound data. Each sound address serves to map to its corresponding sound data. This .mif file is subsequently loaded into the FPGA using an Intel *altsyncram megafunction* IP [5]. As a result, whenever a sound address is received from the Audio Interface Module, the corresponding sound data is retrieved from the Audio Rom and delivered to the DAC_left and DAC_right of the CODEC Interface IP.

VGA Timing

vga_counter module is used for generating VGA timing signals based on a 50 MHz clock.

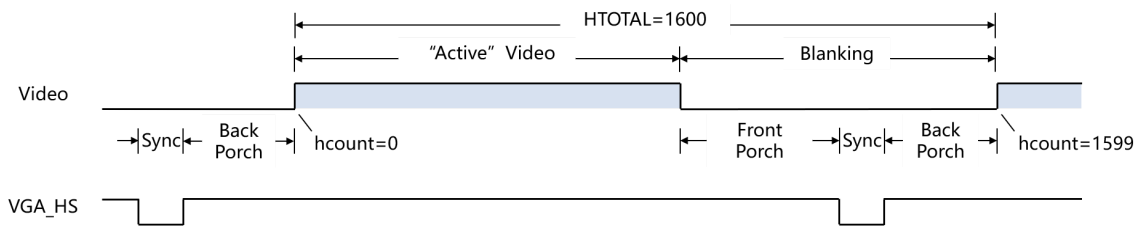
VGA displays images by scanning. The pixels that make up the image are scanned onto the display screen in a top-down, left-to-right order, synchronized with the horizontal sync signal and vertical sync signal. The horizontal sync timing can be divided into 4 stages, and the parameters for these 4 stages are strictly defined. If the parameter configuration is incorrect, the VGA cannot display. VGA can support multiple resolutions, and the parameters for each stage vary with different resolutions. The commonly used VGA resolutions and their corresponding timing parameters are shown in the figure below.

Format	Pixel Clock (MHz)	Horizontal (in Pixels)				Vertical (in Lines)			
		Active Video	Front Porch	Sync Pulse	Back Porch	Active Video	Front Porch	Sync Pulse	Back Porch
640x480, 60Hz	25.175	640	16	96	48	480	11	2	31
640x480, 72Hz	31.500	640	24	40	128	480	9	3	28
640x480, 75Hz	31.500	640	16	96	48	480	11	2	32
640x480, 85Hz	36.000	640	32	48	112	480	1	3	25
800x600, 56Hz	38.100	800	32	128	128	600	1	4	14
800x600, 60Hz	40.000	800	40	128	88	600	1	4	23
800x600, 72Hz	50.000	800	56	120	64	600	37	6	23
800x600, 75Hz	49.500	800	16	80	160	600	1	2	21
800x600, 85Hz	56.250	800	32	64	152	600	1	3	27
1024x768, 60Hz	65.000	1024	24	136	160	768	3	6	29
1024x768, 70Hz	75.000	1024	24	136	144	768	3	6	29
1024x768, 75Hz	78.750	1024	16	96	176	768	1	3	28
1024x768, 85Hz	94.500	1024	48	96	208	768	1	3	36

Our VGA resolution is 640*480, with a refresh rate of 60 Hz. Our clock signal is 50 MHz. For this reason, we multiply the length of each horizontal stage by 2, thereby using hcount to divide the clock, generating a 25 MHz VGA clock signal VGA_CLK.

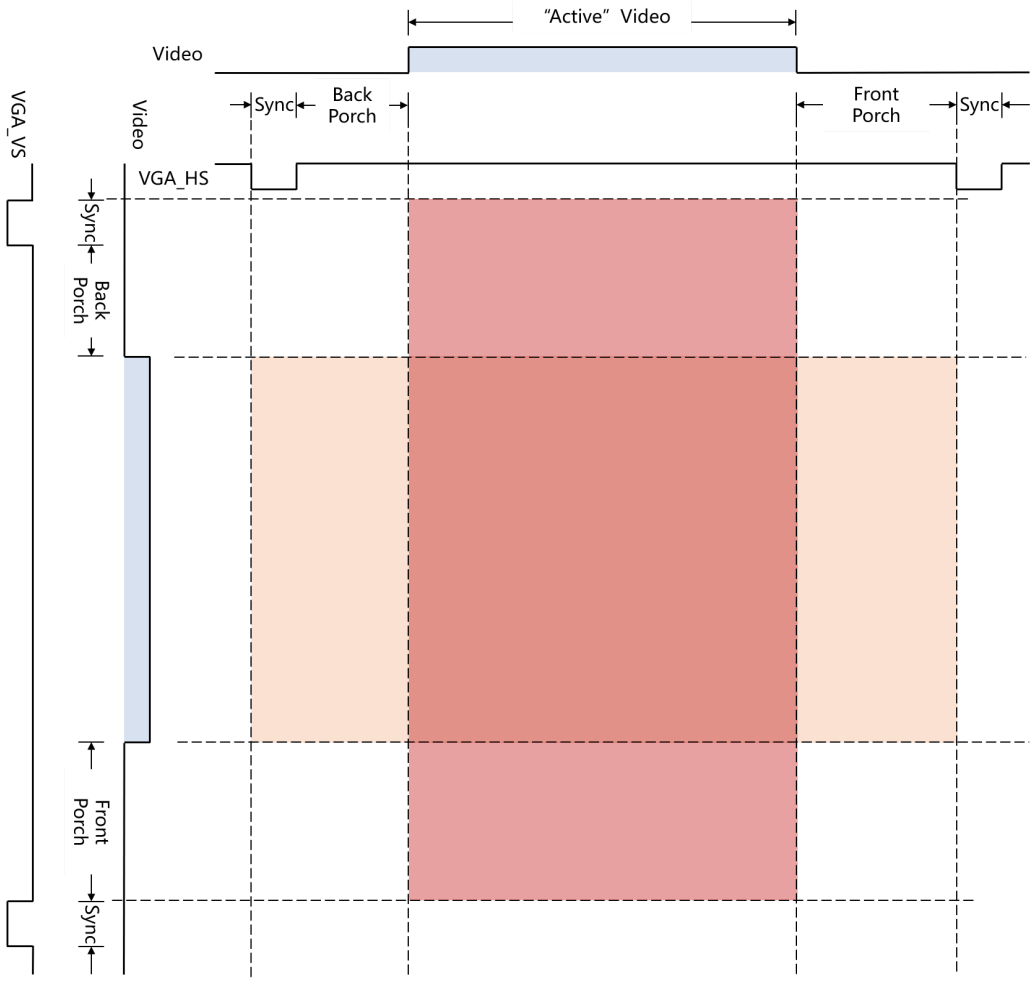
```
parameter HACTIVE      = 11'd 1280, // Parameters for hcount
          HFRONT_PORCH = 11'd 32,
          HSYNC        = 11'd 192,
          HBACK_PORCH  = 11'd 96,
          HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH; // 1600


parameter VACTIVE      = 10'd 480, // Parameters for vcount
          VFRONT_PORCH = 10'd 10,
          VSYNC        = 10'd 2,
          VBACK_PORCH  = 10'd 33,
          VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH; // 525
```

The horizontal timing definition of VGA is shown in the following figure. In our case, hcount is the horizontal counter, hcount[10:1] represents the pixel column. It counts from 0 to 1599 (HTOTAL - 1). Also used to produce a 25 MHz clock (hcount[0]). vcount[9:0] representing the pixel row. It counts from 0 to 524 (VTOTAL - 1). Our horizontal synchronization signal (VGA_HS) and vertical synchronization signal (VGA_VS) are active low during the horizontal sync period.



In this module, every two clk50 cycles correspond to one pixel. Every horizontal line is counted, so the vcount is incremented every time a full horizontal line completes its count. Since hcount completes one cycle every 1600 clk50 cycles, vcount is incremented once every 1600 clk50 cycles.

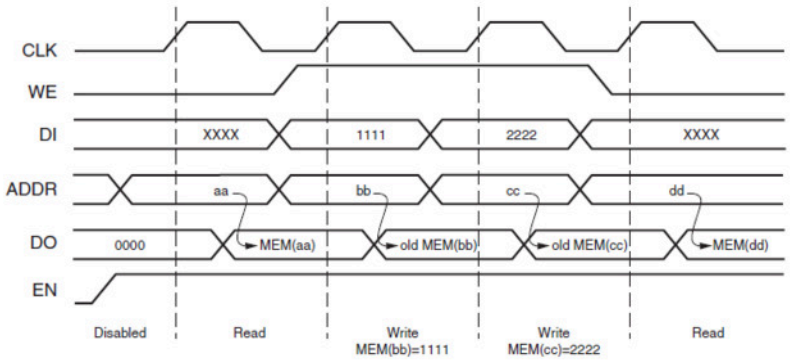
Combining the horizontal synchronization timing diagram with the vertical synchronization timing diagram forms the VGA timing diagram. The red area in the diagram indicates that the video information is only valid in this area during a complete horizontal scanning cycle. The yellow area indicates that the video information is only valid in this area during a complete vertical scanning cycle. The orange area, where the two intersect, is the final display area of the VGA.



BRAM

twoportbram is a parameterized module defining a dual-port block RAM.

This module contains a memory array mem which stores data. On each clock cycle, if write is enabled, data is written to the memory at address wa. Simultaneously, data is read from the memory at address ra and output to data_out.



The game stores sprite data in three BRAM instances for x-coordinates (sprites_x), y-coordinates (sprites_y), and sprite IDs (sprites_n). When address is 4'h5, sprite data from Avalon bus writedata is extracted and stored in sprites_x_cord, sprites_y_cord, and sprites_n_value. During VGA signal generation, the game logic reads sprite data from the BRAMs using sprites_read_address. The data is used to determine sprite positions and which sprite to display.

VGA Controller (Main Logic)

The main logic is defined in the *vga_bird* module, mainly used for: 1) handling the reading from/writing to the Avalon memory-mapped interface, 2) managing the state machine for rendering sprites, and 3) updating buffers buf_e and buf_o with sprite pixel data.

- **Avalon Memory-Mapped Interface Handler**

This function handles the reading from and writing to the Avalon memory-mapped interface. It ensures that data written by Avalon master gets stored correctly in the internal registers and memory.

```
    always_ff @(posedge clk) begin
        if (reset) begin
            score  <=  16'h0;
             combo  <=  16'h0;

        // data from avalon bus writes a piece of data to bram
        end else if (chipselect && write) begin
            case (address)
                4'h3 : begin
                    score <= writedata[15:0];
                    combo <= writedata[31:16];
                end
                // 4'h4 : begin
                //     gamedata <= writedata;
                //     reset_sw <= writedata[0];
                end
                4'h5 : begin
                    sprites_x_cord          <= writedata[9:0];
                    sprites_y_cord          <= writedata[19:10];
                    sprites_n_value         <= writedata[25:20];
                    sprites_write_address   <= writedata[31:26];
```

```
                    sprites_write           <= 1;
                end
            endcase
        end else if (sprites_write) begin
            sprites_write <= 0;
        end
    end
```

Reset Handling: When the reset signal is high, it resets the score and combo registers to zero.

Data Write Handling: When the chipselect and write signals are high, it processes the address and writedata signals to update various internal registers:

Address	Function
4'h3	Updates the score registers.
4'h5	Interprets writedata as a sprite packet, extracting the x-coordinate, y-coordinate, sprite index, and sprite entry, and sets these into the respective registers.

Sprite Write Reset: If a sprite write operation was initiated, it resets the sprites_write flag in the next clock cycle to complete the write operation.

- **Sprite Rendering State Machine**

This function manages the state machine responsible for rendering sprites on the VGA display. It coordinates the process of reading sprite data and updating the frame buffer with the appropriate pixel information.

Reset Handling: Initializes buffers and state variables.

Background Drawing: Updates the background color in the buffers at the beginning of each frame.

State Machine Initialization: Resets the state machine at start of each horizontal line (hcount == 11'd1).

State Machine Execution: Progresses through different states to render sprites:

State	Function
0	Prepares for reading sprite data.
1	Checks if the sprite is active and within the current line.
2	Updates the buffer with the sprite's pixel data if it is active.

Repetition and Completion: Iterates over pixels and sprites until all are processed, then sets the done flag.

- **Buffer Update for Sprite Pixels**

buf_e and buf_o are used to manage even and odd frames or lines for smooth display updates. They are 2D arrays, where the first dimension (640) represents the horizontal resolution (number of pixels in a row), and the second dimension (4), represents the color depth (4-bit color per pixel). This means each pixel can have one of 16 colors. Our color palette is shown in the **Color Palette** section.

This function updates the buffers buf_e and buf_o with sprite pixel data. It ensures that the appropriate pixels for each sprite are written to the frame buffer, which is then displayed on the VGA.

```
    always_ff @(posedge clk) begin
        if(reset) begin
            buf_e <= {640{4'hb}};
            buf_o <= {640{4'hb}};
            ...
        end else begin
            if (vcount[0]) begin     // Edit buffer_even
                if ((hcount[10:1] > 640) && (vcount < 10'd480)) begin
                    for (int i = 0; i < 480; i++)
                        buf_o[i] <= bg_color;
                    for (int i = 480; i < 640; i++)
                        buf_o[i] <= 4'h0;
                end
            end else begin           // Edit buffer_odd
                if ((hcount[10:1] > 640) && (vcount < 10'd480)) begin
                    for (int i = 0; i < 480; i++)
                        buf_e[i] <= bg_color;
                    for (int i = 480; i < 640; i++)
                        buf_e[i] <= 4'h0;
                end
            end
            ...
            if(~done) begin
                case(state)
                    ...
                    4'd2 : begin
                        // Update buffers with sprite pixel data
                        if (vcount[0]) begin
                            if (pattern != 4'h0) begin
                                if (xposition < 10'd640)
                                    buf_e[xposition] <= pattern;
                            end
                        end else begin
                            if (pattern != 4'h0) begin
                                if (xposition < 10'd640)
                                    buf_o[xposition] <= pattern;
                            end
                        end
            ...
```

Buffer Initialization: Sets the initial color for the buffers based on the bg_color.

Background Update: Clears the buffers with the background color at the start of each frame.

Buffer Update: Writes the sprite's pixel data to the correct buffer (buf_e or buf_o) based on the current line being drawn.

Sprite Bitmaps

The *sprites* module is responsible for selecting and fetching sprite data from different ROMs based on a sprite index (n_sprite). Each sprite is represented by a ROM initialized with pixel data that defines the sprite's appearance. The inputs are the index of the sprite to be fetched, the line number and pixel number of the sprite to be read, and the clock signal. The output is the color code of the current pixel of the selected sprite.

```
always_comb begin
    case (n_sprite)
        6'd1  : color_code = spr_rom_data[6'd1];  // Sprite 1
        6'd2  : color_code = spr_rom_data[6'd2];  // Sprite 2
        ...
        default : color_code = 4'h0; // Default color code
    endcase
end
```

Address Calculation: spr_rom_addr is calculated using the formula (line << 5) + pixel. This creates a unique address for each pixel within a sprite.

Sprite ROM Instances: Multiple instances of the rom_sync module are created, each representing a different sprite. Each ROM is initialized with a text file containing the pixel data for that sprite. The ROM instances read the pixel data corresponding to spr_rom_addr.

Sprite Data Selection: The always_comb block selects the color code from the appropriate ROM based on n_sprite.

The sprite index and corresponding element are shown in the following table.

Category	Sprite Index	Element
Digits	6'd1-6'd10	1, 2, 3, 4, 5, 6, 7, 8, 9, 0
Letters	6'd11	B
	6'd12	C
	6'd13	E
	6'd14	M
	6'd15	O
	6'd16	R
	6'd17	S

	6'd26	A
	6'd27	X
	6'd35	Y
	6'd36	D
	6'd37	H
White Letters	6'd18	E white
	6'd19	U white
	6'd20	M white
	6'd21	V white
	6'd22	S white
	6'd23	T white
	6'd24	I white
	6'd25	N white
	6'd31	G white
	6'd33	A white
	6'd34	R white
	6'd38	O white
Bird/Pipe	6'd28	bird
	6'd29	pipe1
	6'd30	pipe2
	6'd32	dead bird

ROM

The *rom_sync* module is a parameterized synchronous ROM (Read-Only Memory). It reads data from memory initialized with contents from a file and outputs the data synchronously with the clock.

Memory Initialization: The initial block loads the memory contents from the file specified by INIT_F.

```
initial begin
    if (INIT_F != 0) begin
```

```
        $display("Creating rom_sync from init file '%s'.", INIT_F);
        $readmemh(INIT_F, memory);
    end
end
```

Data Output: The always_ff block updates the data output on the rising edge of the clk signal, ensuring synchronous operation.

```
always_ff @(posedge clk) begin
    data <= memory[addr];
end
```

Color Palette

The *sprite_color_pallete* module maps 4-bit color codes to 24-bit RGB color values. It selects between two frame buffers based on the select signal and outputs the corresponding RGB color.

```
always_comb begin
    case(color_code)
        4'h0 : color = 24'hFFFFFF;
        4'h1 : color = 24'hFFFFFF;
        4'h2 : color = 24'h646361;
        ...
        default : color = 24'h000000;
    endcase
end
```

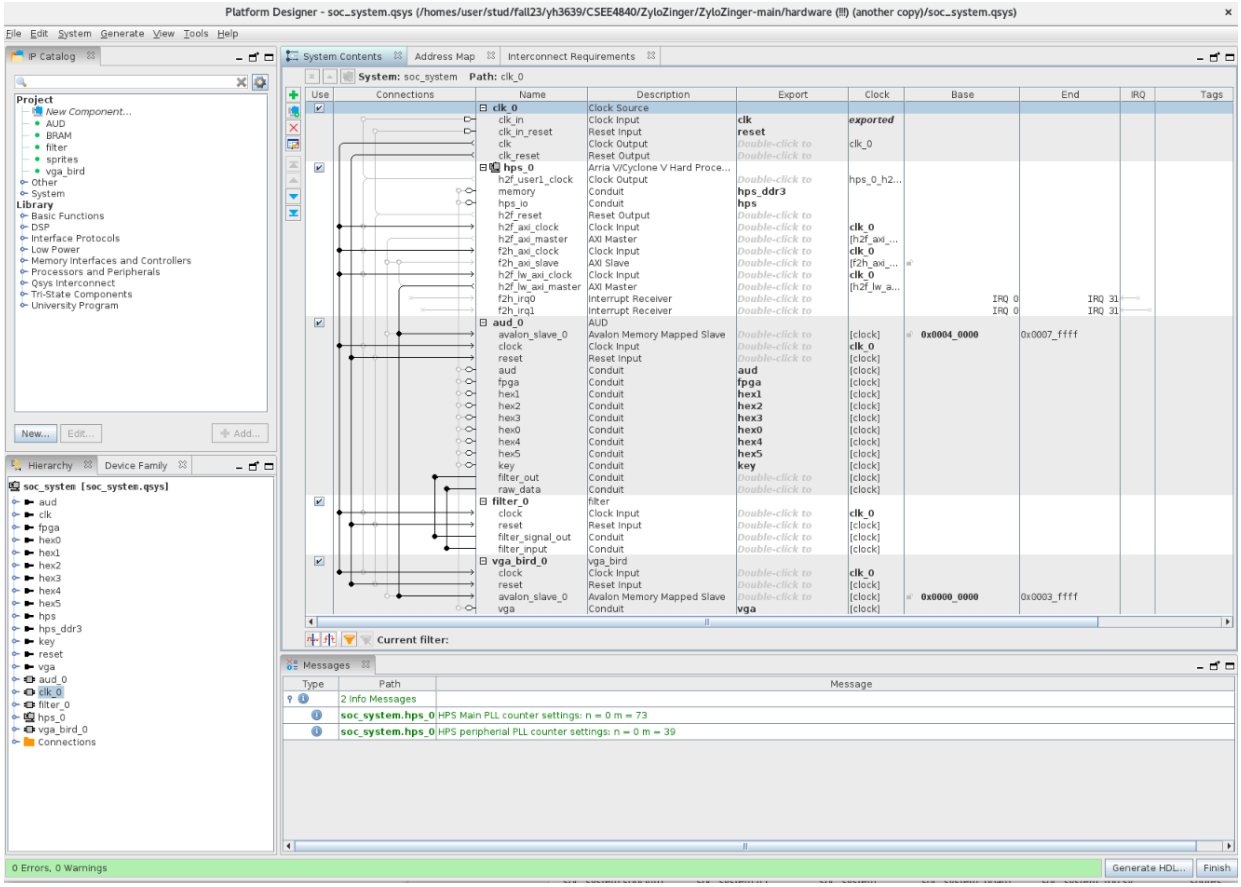
The color palette is shown as follows:

Color Code	Hex Value	RGB Value	Color Name
0x0	0xFFFFFF	(255, 255, 255)	White
0x1	0xFFFFFF	(255, 255, 255)	White
0x2	0x646361	(100, 99, 97)	Dark Gray
0x3	0x0a0808	(10, 8, 8)	Almost Black
0x4	0xfdc603	(253, 198, 3)	Golden Yellow
0x5	0x5f2a04	(95, 42, 4)	Brown
0x6	0xea7e02	(234, 126, 2)	Orange
0x7	0xdab6ff	(218, 182, 255)	Lavender
0x8	0x101f06	(16, 31, 6)	Very Dark Green
0x9	0x7ed012	(126, 208, 18)	Lime Green

0xa	0x0bad01	(11, 173, 1)	Bright Green
Default	0x000000	(0, 0, 0)	Black

Qsys--System Connections

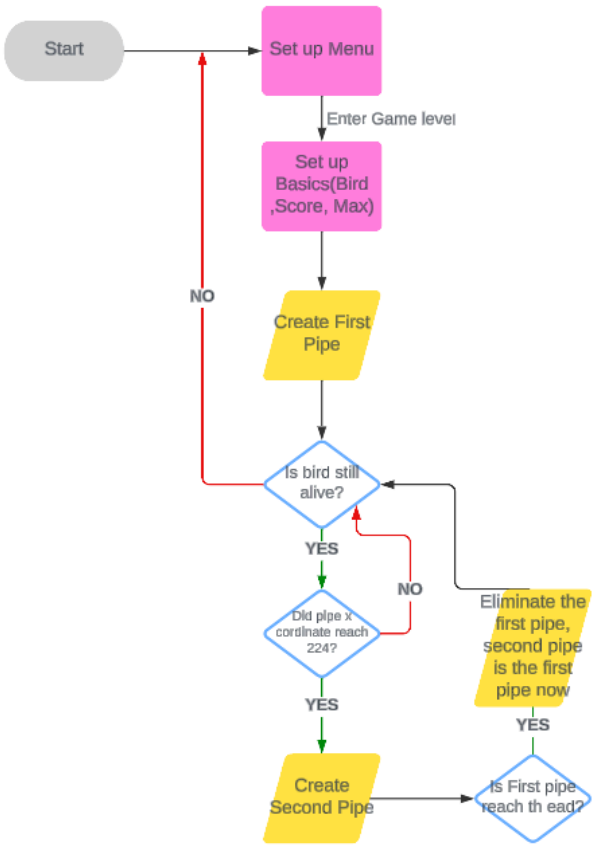
All SystemVerilog modules are added and connected in the Qsys platform builder from Altera. The connections are shown below.



Software

Game Logic Module

Game Logic Flow Diagram:



Game Mechanics:

1. Player Control
   - The player controls the bird's altitude by making sound. Whenever the volume of the sound made by the user reaches a threshold, the bird will flip its wings and ascend a short distance.
   - Gravity constantly pulls the bird downward by setting the bird.dy to -1
   - The player must time their sounds carefully to navigate the bird through openings between the pipes.

2. Obstacles
   - Green pipes serve as obstacles for the bird to navigate through. They are positioned at different heights, with gaps between them for the bird to fly through.
   - Pipes move from right to left across the screen at a constant speed.
   - If the bird collides with any part of a pipe or the ground, the game ends.

3. Scorings
    - The player earns points by successfully navigating the bird through the gaps between pipes.
    - Each successful passage through a pair of pipes earns one point.
    - The player's score is displayed on the screen during gameplay.

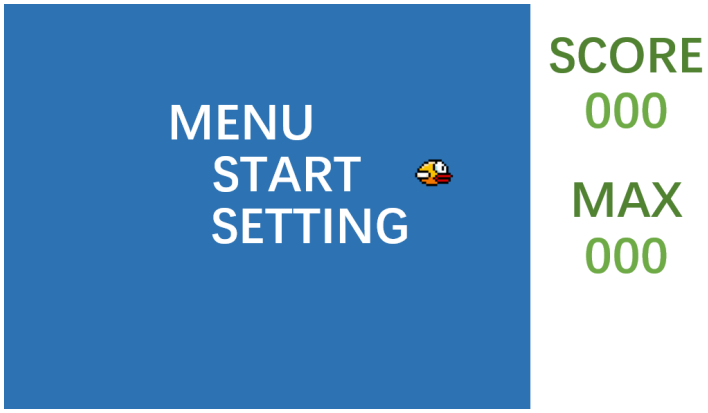
4. Max Score
    - The system will retain the maximum score achieved by the player, ensuring it remains unchanged even if the player restarts the program.
    - Different game levels will each have their own unique maximum score, and these scores will not be interchangeable between levels.

5. Difficulty
    - The gamer could set up difficult in the beginning of the game, i.e. Easy, Medium, Hard
    - The speed and length of the moving pipes increases as the player select harder game level

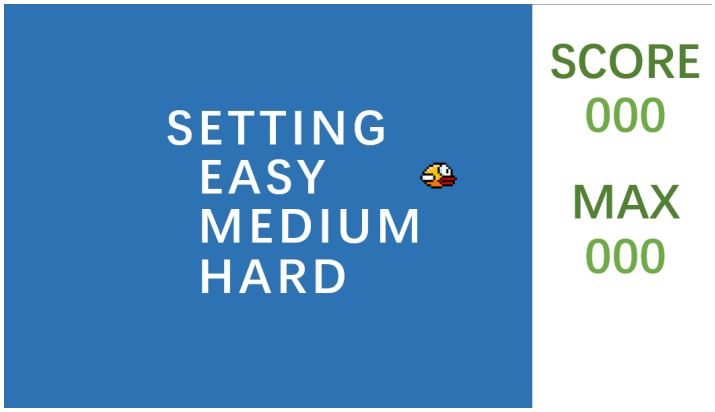
6. Game Over
    - The game ends when the bird collides with a pipe or the ground.
    - Upon game over, the player's final score is displayed on the screen.
    - The player can choose to restart the game to try again.

User Interface

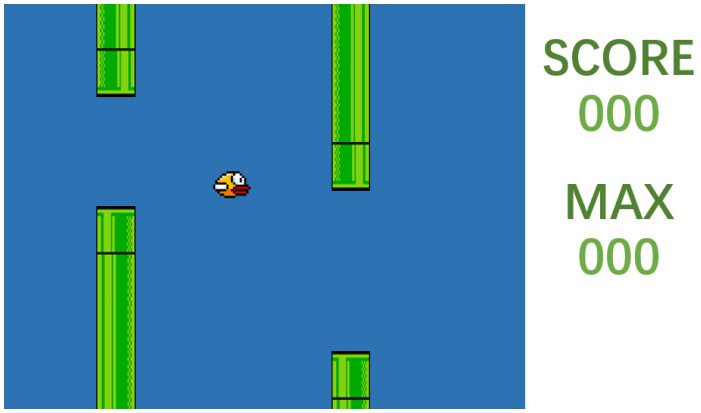
1. Game Menu
    - Display: In the menu section, the user can choose "Start" or "Settings" by clicking buttons on the board. The first button from the right is "Next," the second button from the right is "Confirm," and a bird icon on the right side indicates the current selection.



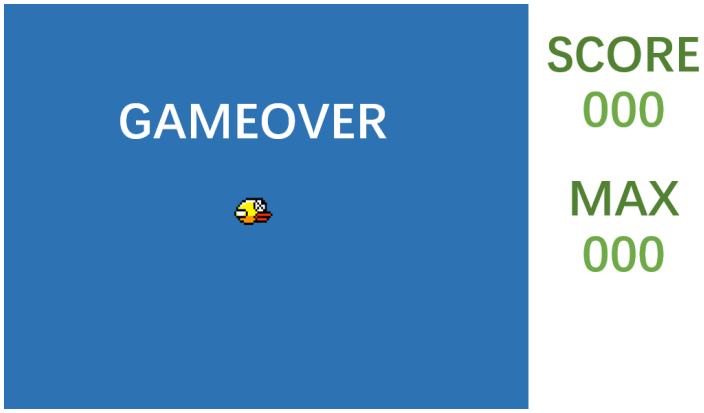
    - Display: In the Setting Module, the user can choose levels from EASY, MEDIUM, and HARD by clicking the "Next" button, adjusting the game level accordingly. After selecting a level, the user can click the "Confirm" button to set the level and return to the main menu. When the user clicks "Start," the game will begin.



2. Game Screen
   - Display: After starting the game, the user needs to make sounds into the microphone to make the bird flap and pass through the pipes. Each time the bird successfully passes a pipe, the score increases by 1.



3. Game Over Screen
   - Display: When the bird collides with a pipe, touches the ground, or the top of the screen, the game-over screen will appear. This screen will display a dead bird and show your score and the maximum score on the right side of the screen as follows,



Physics Simulation

1. Gravity Effect
   - Apply a constant downward acceleration (gravity) to the bird's vertical speed, by setting bird.dy to -1 when the game starts.
   - Update the bird's position based on its speed at each frame.

2. Rise on Command
   - When a "scream" is detected, temporarily apply an upward force that counteracts gravity, simulating a rise, by setting the bird.dy to 20.

3. Pipe construction
   - Randomly generate pipes of varying lengths and speeds, corresponding to the game level.
   - Generate the first pipe when the game starts, and subsequent pipes whenever the previous pipe reaches the 224 x coordinate.
   - Pipes disappear when they reach the leftmost edge of the screen.

4. Collision Detection
   - Check if the bird collides with obstacles (pipes) or the ground.
   - If a collision occurs, the game will play the collision sound and display the game over screen.
   - After the game over screen is shown, the game will return to the menu after 3 seconds.

Control Algorithm

1. Listen for Commands:
   - Continuously monitor the microphone for sound exceeding the threshold.
   - Trigger a "rise" action whenever such sound is detected.

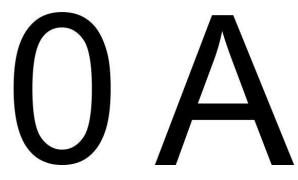
2. Update Game Physics:
   - At each frame, update the bird's physics based on gravity and any active "rise" commands.
   - Adjust the bird's position and handle collisions accordingly.

3. Game Flow Control:
   - Manage game states (start, playing, game over) and transitions between them.
   - Reset game parameters at the start or after a game over to prepare for a new game.

Resource Budget

The cyclone V has two types of BRAMs, MLABs and M10Ks. According to the document, we have 138 Kb MlAB memory and 1400 Kb M10K memory.
The memory required is listed as shown in the table below.

1. Graphic Resources

Element	Number	Pixel size	Size	Example
Bird/Dead bird	2	32*32	1Kb	
Pipe	2	32*32	1Kb	
Character	34	32*32	1Kb	

2. Audio Resources

Element	Time	Sample Frequency	Bit	Size
Flap sound	0.38s	48k Hz	16	0.38*48k*16=35.63KB
Hit sound	0.59s	48k Hz	16	0.59*48k*16=55.25KB

Hardware/Software Interface

In FPGA game design, the software-hardware interface is crucial for facilitating communication between the software and the hardware. This interface ensures that commands, data, and control signals can be exchanged efficiently and correctly. In our work, we use *ioctl* (input/output control) calls to manage the communication between hardware and software.

ioctl is a system call in Linux used to control devices. It provides a way to send device-specific commands from user-space applications to the kernel-space device drivers. Here are the main four steps to implement it in our design.

1. Device Driver Implementation

The driver exposes an interface through a device file, in our case, /dev folder. And the driver defines specific ioctl commands that correspond to various control and data transfer operations.

2. Defining ioctl Commands

```
#define AUD_READ_DATA          _IOR(AUD_MAGIC, 1, aud_arg_t *)
#define AUD_READ_BUTTON      _IOR(AUD_MAGIC, 2, aud_arg_t *)
#define AUD_WRITE_SOUND      _IOW(AUD_MAGIC, 3, aud_arg_t *)
```

3. Handling ioctl in the Driver

```
static long aud_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
	aud_arg_t vla;
	switch (cmd) {
		case AUD_READ_DATA:
			read_memory(&vla.memory);
			if (copy_to_user((aud_arg_t *) arg, &vla, sizeof(aud_arg_t)))
				return -EACCES;
			break;
		case AUD_WRITE_SOUND:
			if (copy_from_user(&vla, (aud_arg_t *) arg, sizeof(aud_arg_t)))
				return -EACCES;
			write_sound(&vla.memory);
			break;
		case AUD_READ_BUTTON:
			read_button(&vla.memory);
			if (copy_to_user((aud_arg_t *) arg, &vla, sizeof(aud_arg_t)))
				return -EACCES;
			break;
		default:
			return -EINVAL;
	}
	return 0;
}
```

4. User-Space Application

```
int vga_bird_fd;
int aud_fd;
...
int check_receive_audio(int counter, float* sum_audio_data, int aud_fd, aud_mem_t*
amt) {
	amt->data = get_aud_data(aud_fd);
	//printf("amt=%d\n",amt->data);
	if ((counter%10)==0) {
		if (*sum_audio_data / 10 > 50000000){
			printf("after ten counter, with hit, the mean is: %f\n",
(*sum_audio_data/10));
			*sum_audio_data = 0;
			return 1;
		}
		//printf("after ten counter, without hit, the mean is: %f\n",
(*sum_audio_data/10));
		*sum_audio_data = 0;
		return 0;
	} else{
		*sum_audio_data += abs(amt->data);
		return 0;
	}
	return 0;
}
```

Audio Driver

The audio driver implemented in aud.c. This driver defines the basic operations required to interact with the audio hardware, including reading and writing memory, setting address limits, and reading button states. The device registers are mapped using the device tree, and memory regions are requested and mapped to facilitate register access. The driver's operations are exposed via an ioctl interface, allowing user-space applications to perform read and write operations. The aud_probe function handles the initialization by registering the device, mapping resources, and arranging register access, while the aud_remove function cleans up by unmapping and releasing resources.

VGA Driver

This driver is implemented in vga_bird.c. It manages interactions with VGA hardware, including updating video data packets, scores, and combo values. The driver registers are defined and mapped using the device tree, ensuring proper access to memory regions. User-space applications can interact with the driver through an ioctl interface for reading and writing operations. The initialization function, vga_bird_probe, registers the device, maps necessary resources, and configures register access. Cleanup is handled by the vga_bird_remove function, which releases resources and unmaps memory. The driver is integrated as a platform driver, which allows automatic detection and initialization of the device when the corresponding hardware is present. This setup ensures efficient management and operation of the VGA video generator hardware.

Registers

The table outlines various registers used in a system to manage audio, button inputs, scoring, and sprite information. Each register is identified by its address and name, and detailed by its function and the number of bytes it occupies. The audio-related registers handle reading microphone input and sending sound data, each occupying 4 bytes. Button inputs from hardware are captured in another 4-byte register. The scoring system for the game is also managed through a dedicated 4-byte register. Additionally, sprite information, which includes the index, ID, and coordinates, is transmitted from software to hardware using a register that occupies 4 bytes.

Address	Register name	Description	Bytes
0	audio_read_data	Audio read data, return the microphone input	4
4	button_value	Button input from hardware	4
8	sound	Audio send data, send the sound index	4
12	score	Score for this game	4
20	Sprite array	Send the Sprite array info from software to hardware. It consists of the index, ID, x and y.	4

Milestones

Milestone 1

1. Implement audio interface on the FPGA board.
2. Implement display interface on the FPGA board.

Milestone 2

1. Software driver interface with hardware
2. Implement the Game logic

Milestone 3

1. Complete Graphics
2. Finish the project

Group member contributions

Yang Li: Contributed to the project by assisting in writing and debugging the software code and writing sections of the report.

Yiran Hu: Studied the principles of VGA display. Used Python to convert images into 32x32 sprites components, and displayed the sprite on the VGA. Wrote a testbench for FIR filter and conducted functionality tests. Participated in designing the game user interface and logic with the software team members to bring them to life. Studied the hardware-software interface and learned how to use ioctl.

Yuesheng Ma: Studied the Codec Interface IP and utilized it to configure the audio codec via I2C as well as sending and receiving the audio. Generated the .mif file from mp3 and successfully made the speaker play the correct sound based on the software instruction. Designed the FIR filter with Python and implemented the FPGA codes.

Chenyang Zhou: Focused on the software part and tested the interface between software and hardware. Designed and implemented the game logic.

Future work

In future iterations of the game, several enhancements are planned to improve the user experience and gameplay. First, efforts will be made to refine the bird's movement, making it smoother and more realistic. This will involve fine-tuning the physics and response to microphone input. Second, additional difficulty levels will be introduced, including more challenging scenarios where the pipes can move up and down, adding dynamic obstacles for the player to navigate. Additionally, continuous background music (BGM) will be integrated to enhance the immersive experience, although this will require managing increased memory usage. Lastly, the game's interface will be improved for better usability and visual appeal, providing a more engaging and polished experience for the players.

Complete listing of Project Files

Hardware: aud.sv, bram.sv, filter.sv, global_variables.sv, sprite.sv, vga_bird.sv, audio_rom.v. All PNG file that saved in sprite folder (including all letters, numbers, bird and pipes), one .mif file(output.mif)

Software:aud.c, aud.h, interface.c, interface.h, test2_update.c, vga_bird.c, vga_bird.h

References

[1] Huack, Scott, and Kyle Gagner. “Audio Tutorial - Class.ece.uw.edu.” UW Electrical & Computer Engineering, University of Washington, 18 May 2015, https://class.ece.uw.edu/271/huack2/de1/audio/Audio_Tutorial.pdf.
[2] Wolfson Microelectronics. "WM8731 Audio CODEC Datasheet." SparkFun, https://cdn.sparkfun.com/datasheets/Dev/Arduino/Shields/WolfsonWM8731.pdf.
[3] SciPy Contributors. "scipy.signal.firwin — SciPy v1.8.0 Reference Guide." SciPy.org, 2022, https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.firwin.html.
[4] "Flappy Bird Sounds." 101soundboards.com, https://www.101soundboards.com/boards/10178-flappy-bird-sounds.
[5] Intel. "altsyncram - Intel Quartus Prime Standard Edition User Guide: Vol. 1 Design and Synthesis." Intel.com, Accessed 20 May 2024, https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/hdl/mega/mega_file_altsynch_ram.htm.

Code

Hardware

sprites.sv

```
/*
 * Sprite ROM selector and color decoders as well as synchronous ROM module
 * Columbia University
 */

module sprites(
      input logic [5:0]               n_sprite,
      input logic [9:0]               line,
      input logic [5:0]               pixel,
      input logic                     clk,
      output logic [3:0]              color_code);

      logic [9:0] spr_rom_addr ;

      assign spr_rom_addr = (line<<5) + pixel;

      logic [3:0] spr_rom_data [42:0];
      // sprites individually stored in roms
      // numbers
      rom_sync #(
        .WIDTH(4),
        .WORDS(1024),
        .INIT_F("./sprites/Sprite_rom/1.txt")
   ) num_1 (
      .clk(clk),
        .addr(spr_rom_addr),
        .data(spr_rom_data[6'd1])
   );
      rom_sync #(
        .WIDTH(4),
        .WORDS(1024),
        .INIT_F("./sprites/Sprite_rom/2.txt")
   ) num_2 (
      .clk(clk),
        .addr(spr_rom_addr),
        .data(spr_rom_data[6'd2])
   );
      rom_sync #(
        .WIDTH(4),
        .WORDS(1024),
        .INIT_F("./sprites/Sprite_rom/3.txt")
   ) num_3 (
      .clk(clk),
        .addr(spr_rom_addr),
        .data(spr_rom_data[6'd3])
   );
```

```
    rom_sync #(
      .WIDTH(4),
      .WORDS(1024),
      .INIT_F("./sprites/Sprite_rom/4.txt")
  ) num_4 (
    .clk(clk),
      .addr(spr_rom_addr),
      .data(spr_rom_data[6'd4])
  );
    rom_sync #(
      .WIDTH(4),
      .WORDS(1024),
      .INIT_F("./sprites/Sprite_rom/5.txt")
  ) num_5 (
    .clk(clk),
      .addr(spr_rom_addr),
      .data(spr_rom_data[6'd5])
  );
    rom_sync #(
      .WIDTH(4),
      .WORDS(1024),
      .INIT_F("./sprites/Sprite_rom/6.txt")
  ) num_6 (
    .clk(clk),
      .addr(spr_rom_addr),
      .data(spr_rom_data[6'd6])
  );
    rom_sync #(
      .WIDTH(4),
      .WORDS(1024),
      .INIT_F("./sprites/Sprite_rom/7.txt")
  ) num_7 (
    .clk(clk),
      .addr(spr_rom_addr),
      .data(spr_rom_data[6'd7])
  );
    rom_sync #(
      .WIDTH(4),
      .WORDS(1024),
      .INIT_F("./sprites/Sprite_rom/8.txt")
  ) num_8 (
    .clk(clk),
      .addr(spr_rom_addr),
      .data(spr_rom_data[6'd8])
  );
    rom_sync #(
      .WIDTH(4),
      .WORDS(1024),
      .INIT_F("./sprites/Sprite_rom/9.txt")
  ) num_9 (
    .clk(clk),
      .addr(spr_rom_addr),
      .data(spr_rom_data[6'd9])
  );
```

```
    rom_sync #(
      .WIDTH(4),
      .WORDS(1024),
      .INIT_F("./sprites/Sprite_rom/0.txt")
  ) num_10 (
    .clk(clk),
      .addr(spr_rom_addr),
      .data(spr_rom_data[6'd10])
  );

    // Letters
    rom_sync #(
      .WIDTH(4),
      .WORDS(1024),
      .INIT_F("./sprites/Sprite_rom/B.txt")
  ) num_11 (
    .clk(clk),
      .addr(spr_rom_addr),
      .data(spr_rom_data[6'd11])
  );

    rom_sync #(
      .WIDTH(4),
      .WORDS(1024),
      .INIT_F("./sprites/Sprite_rom/C.txt")
  ) num_12 (
    .clk(clk),
      .addr(spr_rom_addr),
      .data(spr_rom_data[6'd12])
  );
    rom_sync #(
      .WIDTH(4),
      .WORDS(1024),
      .INIT_F("./sprites/Sprite_rom/E.txt")
  ) num_13 (
    .clk(clk),
      .addr(spr_rom_addr),
      .data(spr_rom_data[6'd13])
  );
    rom_sync #(
      .WIDTH(4),
      .WORDS(1024),
      .INIT_F("./sprites/Sprite_rom/M.txt")
  ) num_14 (
    .clk(clk),
      .addr(spr_rom_addr),
      .data(spr_rom_data[6'd14])
  );
    rom_sync #(
      .WIDTH(4),
      .WORDS(1024),
      .INIT_F("./sprites/Sprite_rom/O.txt")
  ) num_15 (
    .clk(clk),
```

```
    .addr(spr_rom_addr),
    .data(spr_rom_data[6'd15])
);

  rom_sync #(
    .WIDTH(4),
    .WORDS(1024),
    .INIT_F("./sprites/Sprite_rom/R.txt")
) num_16 (
  .clk(clk),
    .addr(spr_rom_addr),
    .data(spr_rom_data[6'd16])
);
  rom_sync #(
    .WIDTH(4),
    .WORDS(1024),
    .INIT_F("./sprites/Sprite_rom/S.txt")
) num_17(
  .clk(clk),
    .addr(spr_rom_addr),
    .data(spr_rom_data[6'd17])
);
  // NOTE BLOCKS
  rom_sync #( //blue
    .WIDTH(4),
    .WORDS(1024),
    .INIT_F("./sprites/Sprite_rom/E_white_32.txt")
) num_18 (
  .clk(clk),
    .addr(spr_rom_addr),
    .data(spr_rom_data[6'd18])
);
  rom_sync #(
    .WIDTH(4),
    .WORDS(1024),
    .INIT_F("./sprites/Sprite_rom/U_white_32.txt")
) num_19 (
  .clk(clk),
    .addr(spr_rom_addr),
    .data(spr_rom_data[6'd19])
);
  rom_sync #( //orange
    .WIDTH(4),
    .WORDS(1024),
    .INIT_F("./sprites/Sprite_rom/M_white_32.txt")
) num_20 (
  .clk(clk),
    .addr(spr_rom_addr),
    .data(spr_rom_data[6'd20])
);
  rom_sync #(
    .WIDTH(4),
    .WORDS(1024),
    .INIT_F("./sprites/Sprite_rom/V_white_32.txt")
```

```
        ) num_21 (
           .clk(clk),
            .addr(spr_rom_addr),
            .data(spr_rom_data[6'd21])
        );
           rom_sync #( //pink
            .WIDTH(4),
            .WORDS(1024),
            .INIT_F("./sprites/Sprite_rom/S_white_32.txt")
        ) num_22 (
           .clk(clk),
            .addr(spr_rom_addr),
            .data(spr_rom_data[6'd22])
        );
           rom_sync #(
            .WIDTH(4),
            .WORDS(1024),
            .INIT_F("./sprites/Sprite_rom/T_white_32.txt")
        ) num_23 (
           .clk(clk),
            .addr(spr_rom_addr),
            .data(spr_rom_data[6'd23])
        );
           rom_sync #(//purple
            .WIDTH(4),
            .WORDS(1024),
            .INIT_F("./sprites/Sprite_rom/I_white_32.txt")
        ) num_24 (
           .clk(clk),
            .addr(spr_rom_addr),
            .data(spr_rom_data[6'd24])
        );
           rom_sync #(
            .WIDTH(4),
            .WORDS(1024),
            .INIT_F("./sprites/Sprite_rom/N_white_32.txt")
        ) num_25 (
           .clk(clk),
            .addr(spr_rom_addr),
            .data(spr_rom_data[6'd25])
        );
           rom_sync #(//purple
            .WIDTH(4),
            .WORDS(1024),
            .INIT_F("./sprites/Sprite_rom/A.txt")
        ) num_26 (
           .clk(clk),
            .addr(spr_rom_addr),
            .data(spr_rom_data[6'd26])
        );
           rom_sync #(
            .WIDTH(4),
            .WORDS(1024),
            .INIT_F("./sprites/Sprite_rom/X.txt")
```

```
        ) num_27 (
           .clk(clk),
            .addr(spr_rom_addr),
            .data(spr_rom_data[6'd27])
        );
            rom_sync #(
            .WIDTH(4),
            .WORDS(1024),
            .INIT_F("./sprites/Sprite_rom/bird.txt")
        ) num_28 (
           .clk(clk),
            .addr(spr_rom_addr),
            .data(spr_rom_data[6'd28])
        );
           rom_sync #(
            .WIDTH(4),
            .WORDS(1024),
            .INIT_F("./sprites/Sprite_rom/pipe1.txt")
        ) num_29 (
           .clk(clk),
            .addr(spr_rom_addr),
            .data(spr_rom_data[6'd29])
        );
           rom_sync #(
            .WIDTH(4),
            .WORDS(1024),
            .INIT_F("./sprites/Sprite_rom/pipe2.txt")
        ) num_30 (
           .clk(clk),
            .addr(spr_rom_addr),
            .data(spr_rom_data[6'd30])
        );
           rom_sync #(
            .WIDTH(4),
            .WORDS(1024),
            .INIT_F("./sprites/Sprite_rom/G_white_32.txt")
        ) num_31 (
           .clk(clk),
            .addr(spr_rom_addr),
            .data(spr_rom_data[6'd31])
        );
           rom_sync #(
           .WIDTH(4),
           .WORDS(1024),
           .INIT_F("./sprites/Sprite_rom/bird_dead.txt")
        ) num_32 (
           .clk(clk),
           .addr(spr_rom_addr),
           .data(spr_rom_data[6'd32])
        );
           rom_sync #(
           .WIDTH(4),
           .WORDS(1024),
           .INIT_F("./sprites/Sprite_rom/A_w.txt")
```

```
        ) num_33 (
           .clk(clk),
           .addr(spr_rom_addr),
           .data(spr_rom_data[6'd33])
        );
           rom_sync #(
           .WIDTH(4),
           .WORDS(1024),
           .INIT_F("./sprites/Sprite_rom/R_w.txt")
        ) num_34 (
           .clk(clk),
           .addr(spr_rom_addr),
           .data(spr_rom_data[6'd34])
        );
           rom_sync #(
           .WIDTH(4),
           .WORDS(1024),
           .INIT_F("./sprites/Sprite_rom/Y.txt")
        ) num_35 (
           .clk(clk),
           .addr(spr_rom_addr),
           .data(spr_rom_data[6'd35])
        );
           rom_sync #(
           .WIDTH(4),
           .WORDS(1024),
           .INIT_F("./sprites/Sprite_rom/D.txt")
        ) num_36 (
           .clk(clk),
           .addr(spr_rom_addr),
           .data(spr_rom_data[6'd36])
        );
           rom_sync #(
           .WIDTH(4),
           .WORDS(1024),
           .INIT_F("./sprites/Sprite_rom/H.txt")
        ) num_37 (
           .clk(clk),
           .addr(spr_rom_addr),
           .data(spr_rom_data[6'd37])
        );
           rom_sync #(
           .WIDTH(4),
           .WORDS(1024),
           .INIT_F("./sprites/Sprite_rom/O_w.txt")
        ) num_38 (
           .clk(clk),
           .addr(spr_rom_addr),
           .data(spr_rom_data[6'd38])
        );

           always_comb begin
                  case (n_sprite)
                         // numbers
```

```
                        6'd1  : color_code = spr_rom_data[6'd1];  // 1
                        6'd2  : color_code = spr_rom_data[6'd2];  // 2
                        6'd3  : color_code = spr_rom_data[6'd3];  // 3
                        6'd4  : color_code = spr_rom_data[6'd4];  // 4
                        6'd5  : color_code = spr_rom_data[6'd5];  // 5
                        6'd6  : color_code = spr_rom_data[6'd6];  // 6
                        6'd7  : color_code = spr_rom_data[6'd7];  // 7
                        6'd8  : color_code = spr_rom_data[6'd8];  // 8
                        6'd9  : color_code = spr_rom_data[6'd9];  // 9
                        6'd10 : color_code = spr_rom_data[6'd10]; // 10
                        // letters
                        6'd11 : color_code = spr_rom_data[6'd11]; // B
                        6'd12 : color_code = spr_rom_data[6'd12]; // C
                        6'd13 : color_code = spr_rom_data[6'd13]; // E
                        6'd14 : color_code = spr_rom_data[6'd14]; // M
                        6'd15 : color_code = spr_rom_data[6'd15]; // O
                        6'd16 : color_code = spr_rom_data[6'd16]; // R
                        6'd17 : color_code = spr_rom_data[6'd17]; // S
                        // notes
                        6'd18 : color_code = spr_rom_data[6'd18]; // E w
                        6'd19 : color_code = spr_rom_data[6'd19]; // U w
                        6'd20 : color_code = spr_rom_data[6'd20]; // M w
                        6'd21 : color_code = spr_rom_data[6'd21]; // V w
                        6'd22 : color_code = spr_rom_data[6'd22]; // S w
                        6'd23 : color_code = spr_rom_data[6'd23]; // T w
                        6'd24 : color_code = spr_rom_data[6'd24]; // I w
                        6'd25 : color_code = spr_rom_data[6'd25]; // N w
                        // A/X added later
                        6'd26 : color_code = spr_rom_data[6'd26]; // A
                        6'd27 : color_code = spr_rom_data[6'd27]; // X
                        // 6'd28 : color_code = spr_rom_data[6'd28]; //
BACKGROUND
                        6'd28 : color_code = spr_rom_data[6'd28]; // BIRD
                        6'd29 : color_code = spr_rom_data[6'd29]; // PIPE1
                        6'd30 : color_code = spr_rom_data[6'd30]; // PIPE2
                        6'd31 : color_code = spr_rom_data[6'd31]; // G w
                        6'd32 : color_code = spr_rom_data[6'd32]; // dead bird
                        6'd33 : color_code = spr_rom_data[6'd33]; // A_w
                        6'd34 : color_code = spr_rom_data[6'd34]; // R_w
                        6'd35 : color_code = spr_rom_data[6'd35]; // Y
                        6'd36 : color_code = spr_rom_data[6'd36]; // D
                        6'd37 : color_code = spr_rom_data[6'd37]; // H
                        6'd38 : color_code = spr_rom_data[6'd38]; // O_w
                        default : begin
                                color_code <= 4'h0;
                        end
                endcase
        end
endmodule

module sprite_color_pallete(
        input logic [3:0]   color_code_o,
        input logic [3:0]   color_code_e,
        input logic         select,
```

```
            output logic [23:0] color
            );
            logic [3:0] color_code;
            assign color_code = (select) ? color_code_o : color_code_e;
            always_comb begin
                  case(color_code)
                        //sprite colors
                        4'h0 : color = 24'hFFFFFF;
                        4'h1 : color = 24'hFFFFFF;
                        4'h2 : color = 24'h646361;
                        4'h3 : color = 24'h0a0808;
                        4'h4 : color = 24'hfdc603;
                        4'h5 : color = 24'h5f2a04;
                        4'h6 : color = 24'hea7e02;
                        4'h7 : color = 24'hdab6ff;
                        4'h8 : color = 24'h101f06;
                        4'h9 : color = 24'h7ed012;
                        4'ha : color = 24'h0bad01;
                        default : color = 24'h000000;
                  endcase
            end
      endmodule

      module rom_sync #(
          parameter WIDTH=4,
          parameter WORDS=1024,
          parameter INIT_F="",
          parameter ADDRW=10
          ) (
          input wire logic clk,
          input wire logic [ADDRW-1:0] addr,
          output     logic [WIDTH-1:0] data
          );

          logic [WIDTH-1:0] memory [WORDS];

          initial begin
              if (INIT_F != 0) begin
                  $display("Creating rom_sync from init file '%s'.", INIT_F);
                  $readmemh(INIT_F, memory);
              end
          end

          always_ff @(posedge clk) begin
              data <= memory[addr];
          end
      endmodule
```

vga_bird.sv

```
/*
 * Avalon memory-mapped peripheral that generates VGA
 * Columbia University
```

```
 */
`include "./bram.sv"
module vga_bird(
    input logic         clk,
    input logic         reset,
    input logic [31:0]  writedata,
    input logic         write,
    input logic         chipselect,
    input logic [15:0]  address,

    //vga ports
    output logic [7:0]  VGA_R, VGA_G, VGA_B,
    output logic        VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
    output logic        VGA_SYNC_n
    );

    // Display
    logic [10:0]        hcount;
    logic [9:0]         vcount;

    vga_counters counters(.clk50(clk), .*);

    //avalon bus address data mapping

    // display up to 16 falling notes on screen
    // writedata note packet
    // |  31:26  |      25:20      |      19:10       |      9:0       |
    // |  index  |  note_id 6 bits |   y cord 10 bits | x-cord 10 bits |
    // |  0-63   |      0-63       |      0-639       |     0-479      |
    // ex:   32h'1a004101 => en = 1, id = 10, type = 0100, y-cord = 256.

    //store up to 16 notes. each note is built with 32 8x8 pixel sprites
    logic           sprites_write;
    logic [5:0]     sprites_write_address;
    logic [5:0]     sprites_read_address;
    logic [9:0]     sprites_x_cord;
    logic [9:0]     sprites_y_cord;
    logic [5:0]     sprites_n_value;
    logic [9:0]     x, y;
    logic [5:0]     n;

    twoportbram #(
        .RAM_WIDTH(10),
        .RAM_ADDR_BITS(6),
        .RAM_WORDS(7'h40)
    ) sprites_x (
        .clk(clk),
        .ra(sprites_read_address),
        .wa(sprites_write_address),
        .write(sprites_write),
        .data_in(sprites_x_cord),
        .data_out(x)
    );
    twoportbram #(
```

```
        .RAM_WIDTH(10),
        .RAM_ADDR_BITS(6),
        .RAM_WORDS(7'h40)
    ) sprites_y (
        .clk(clk),
        .ra(sprites_read_address),
        .wa(sprites_write_address),
        .write(sprites_write),
        .data_in(sprites_y_cord),
        .data_out(y)
    );
    twoportbram #(
        .RAM_WIDTH(6),
        .RAM_ADDR_BITS(6),
        .RAM_WORDS(7'h40)
    ) sprites_n (
        .clk(clk),
        .ra(sprites_read_address),
        .wa(sprites_write_address),
        .write(sprites_write),
        .data_in(sprites_n_value),
        .data_out(n)
    );

    logic [15:0]        score;
    logic [15:0]        combo;

   // logic [31:0]        menu;    // | 1 bit   | 8 bits |   8 bits  |s
                                   // | display | item   | submenu # |
    logic [31:0]        gamedata;
    logic               reset_sw;

    always_ff @(posedge clk) begin
        if (reset) begin
            score  <=  16'h0;
               combo  <=  16'h0;

        // data from avalon bus writes a piece of data to bram
        end else if (chipselect && write) begin
            case (address)
                4'h3 : begin
                    score <= writedata[15:0];
                     combo <= writedata[31:16];
                end

                4'h4 : begin
                    gamedata <= writedata;
                    reset_sw <= writedata[0];
                end
                4'h5 : begin
                    sprites_x_cord          <= writedata[9:0];
                    sprites_y_cord          <= writedata[19:10];
                    sprites_n_value         <= writedata[25:20];
                    sprites_write_address   <= writedata[31:26];
```

```
                sprites_write           <= 1;
            end
        endcase
    end else if (sprites_write) begin
        sprites_write <= 0;
    end
end

logic [9:0]         line;
logic [3:0]         pattern;
assign line = (vcount >= y) ? (vcount - y) : 10'd32;
// logic [5:0]         length;
// logic [5:0]         n;
// sprites in sprites.sv are ordered from least prioety to highest priorety

   sprites sprites0(
    .n_sprite (n),
    .clk (clk),
    .pixel (pixel),
    .line (line),
    .color_code (pattern)
    );


// logic [9:0] x, y;
logic [639:0][3:0] buf_e;
logic [639:0][3:0] buf_o;
logic [3:0] state;
logic [8:0] sprite_index;
logic [5:0] pixel;
logic [9:0] xposition;
logic done;
assign sprites_read_address = sprite_index[5:0];
assign xposition = x + {4'd0, pixel};
//paint basic background
logic [3:0] bg_color;
always_ff @(posedge clk) begin
    if      (vcount == 10'd0)   bg_color <= 4'hc;
end

always_ff @(posedge clk) begin
    if(reset) begin
        buf_e <= {640{4'hb}}; // some red
        buf_o <= {640{4'hb}}; // some red
        state <= 0;
        sprite_index <= 0;
        pixel <= 0;
        done <= 1;
    end else begin
        if (vcount[0]) begin     // output buffer_odd, edit buffer_even
            if ((hcount[10:1] > 640) && (vcount < 10'd480)) begin
                for (int i = 0; i < 480; i++)
                    buf_o[i] <= bg_color;
                for (int i = 480; i < 640; i++)
```

```
                        buf_o[i] <= 4'h0;
                end
            end else begin          // output buffer_even, edit buffer_odd
                if ((hcount[10:1] > 640) && (vcount < 10'd480)) begin
                    for (int i = 0; i < 480; i++)
                        buf_e[i] <= bg_color;
                    for (int i = 480; i < 640; i++)
                        buf_e[i] <= 4'h0;
                end
            end

            if (hcount == 11'd1) begin
                done <= 0;  // 0/1
                state <= 4'd0; // 0-2
                sprite_index <= 0; // 0-127
                pixel <= 0; // 0-31
            end
            if(~done)begin
                case(state)
                    4'd0 : begin
                        // n, x, y ready next clk cycle
                        pixel <= 0;
                        state <= 4'd1;
                    end
                    4'd1 : begin
                        if ((n == 0) || (line >= 32)) begin // not in this line orr
empty sprite
                            pixel <= 6'd0;
                            state <= 4'd0;
                            if (sprite_index < 9'd63)      // check if there are
more sprites to check (existence of 32 potential sprites)
                                sprite_index <= sprite_index + 8'd1;
                            else
                                done <= 1;
                        end else begin
                            state <= 4'd2;
                        end
                    end
                    4'd2 : begin
                        if (vcount[0]) begin        // output buffer_odd, edit
buffer_even
                            if (pattern != 4'h0) begin
                                if (xposition < 10'd640)
                                    buf_e[xposition] <= pattern;
                            end
                        end else begin              // output buffer_even, edit
buffer_odd
                            if (pattern != 4'h0) begin
                                if (xposition < 10'd640)
                                    buf_o[xposition] <= pattern;
                            end
                        end
                                                // repeat writing state pixel is 0-30
or under
```

```
                        if (pixel < 6'd31) begin
                            pixel <= pixel + 1;
                            state <= state;
                        end else begin              // on pixel == 31 (32 pixel)
                            pixel <= 6'd0;
                            state <= 4'd0;
                            if (sprite_index < 9'd63)      // repeat on sprite
number 0-62
                                sprite_index <= sprite_index + 8'd1;
                            else                           // stop on 63
                                done <= 1;
                        end
                    end
                    default:;
                endcase
            end
        end
    end

    //logic [3:0] color_code_out;
    logic [23:0] color_out;
       sprite_color_pallete colors(
             .color_code_o (buf_o[hcount[10:1]][3:0]),
        .color_code_e (buf_e[hcount[10:1]][3:0]),
        .select (vcount[0]),
             .color (color_out)
       );

   always_comb begin
        {VGA_R, VGA_G, VGA_B} = 24'h0;
        if (VGA_BLANK_n) begin
            if (vcount > 10'd1 && vcount < 10'd480) {VGA_R, VGA_G, VGA_B} =
color_out;
        end
    end


endmodule

module vga_counters(
    input logic             clk50, reset,
    output logic [10:0]     hcount,  // hcount[10:1] is pixel column
    output logic [9:0]      vcount,  // vcount[9:0] is pixel row
    output logic            VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0             1279       1599 0
 *
 *             _______________           ________
 * ___________|    Video      |___________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
```

```
 *              ____________________          ____________
 * |____|       VGA_HS          |____|
 */
// Parameters for hcount
    parameter
          HACTIVE      = 11'd 1280,
          HFRONT_PORCH = 11'd 32,
          HSYNC        = 11'd 192,
          HBACK_PORCH  = 11'd 96,
          HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +
          HBACK_PORCH; // 1600

          // Parameters for vcount
          parameter
          VACTIVE      = 10'd 480,
          VFRONT_PORCH = 10'd 10,
          VSYNC        = 10'd 2,
          VBACK_PORCH  = 10'd 33,
          VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
          VBACK_PORCH; // 525

          logic endOfLine;

    always_ff @(posedge clk50 or posedge reset)
        if (reset)          hcount <= 0;
        else if (endOfLine) hcount <= 0;
        else                hcount <= hcount + 11'd 1;

    assign endOfLine = hcount == HTOTAL - 1;

    logic endOfField;

    always_ff @(posedge clk50 or posedge reset)
        if (reset)          vcount <= 0;
        else if (endOfLine)
            if (endOfField)   vcount <= 0;
            else              vcount <= vcount + 10'd 1;

    assign endOfField = vcount == VTOTAL - 1;

    // Horizontal sync: from 0x520 to 0x5DF (0x57F)
    // 101 0010 0000 to 101 1101 1111
    assign VGA_HS = !( (hcount[10:8] == 3'b101) &
                     !(hcount[7:5] == 3'b111));
    assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

    assign VGA_SYNC_n = 1'b0; // For putting sync on the green signal; unused

    // Horizontal active: 0 to 1279     Vertical active: 0 to 479
    // 101 0000 0000  1280          01 1110 0000  480
    // 110 0011 1111  1599          10 0000 1100  524
    assign VGA_BLANK_n =     !( hcount[10] & (hcount[9] | hcount[8]) ) &
                            !( vcount[9] | (vcount[8:5] == 4'b1111) );
```

```
        /* VGA_CLK is 25 MHz
         *
         * clk50    __|  |__|  |__|
         *
         *
         * hcount[0]__|     |_____|
         */
    assign VGA_CLK = hcount[0]; // 25 MHz clock: rising edge sensitive

endmodule
```

bram.sv

```
/*
 * two port Bram module from
http://www.cs.columbia.edu/~sedwards/classes/2023/4840-spring/memory.pdf
 * Columbia University
 */

 module twoportbram
    #(
        parameter                           RAM_WIDTH = 24,
        parameter                           RAM_ADDR_BITS = 16,
        parameter                           RAM_WORDS = 16'hffff
    ) (
    input logic                             clk,
    input logic [RAM_ADDR_BITS - 1:0]       ra, wa,
    input logic                             write,
    input logic [RAM_WIDTH - 1:0]           data_in,
    output logic [RAM_WIDTH - 1:0]          data_out
    );

    logic [RAM_WIDTH - 1:0] mem [RAM_WORDS - 1:0];

    always_ff @(posedge clk) begin
        if (write) mem[wa] <= data_in;
        data_out <= mem[ra];
    end
endmodule
```

aud.sv

```
/*
 * Avalon memory-mapped peripheral that handles audio inputs
 * Utilizes an audio driver that handles Altera driver files.
 * Functions to both store 1.5 seconds of 48khz audio data into BRAM for the Avalon
bus to read out
 * Also can send to software the readout the result of the detector module.
 * Columbia University
 */

`include "global_variables.sv"
```

```
`include "./AudioCodecDrivers/audio_driver.sv"
`include "audio_rom.v"

//`define RAM_ADDR_BITS 5'd16
//`define RAM_WORDS 16'd48000


module hex7seg(input logic  [3:0] a,
               output logic [6:0] y);

   /* Replace this comment and the code below it with your solution */
    always_comb
        case (a)        //      gfe_dcba
            4'h0:        y = 7'b100_0000;
            4'h1:        y = 7'b111_1001;
            4'h2:        y = 7'b010_0100;
            4'h3:        y = 7'b011_0000;
            4'h4:        y = 7'b001_1001;
            4'h5:        y = 7'b001_0010;
            4'h6:        y = 7'b000_0010;
            4'h7:        y = 7'b111_1000;
            4'h8:        y = 7'b000_0000;
            4'h9:        y = 7'b001_0000;
            4'hA:        y = 7'b000_1000;
            4'hB:        y = 7'b000_0011;
            4'hC:        y = 7'b100_0110;
            4'hD:        y = 7'b010_0001;
            4'hE:        y = 7'b000_0110;
            4'hF:        y = 7'b000_1110;
            default:     y = 7'b111_1111;
        endcase
endmodule

module audio_control(

        // 7-segment LED displays; HEX0 is rightmost
        output logic [6:0]        HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,

        //Audio pin assignments
        //Used because Professor Scott Hauck and Kyle Gagner
        output logic              FPGA_I2C_SCLK,
        inout                     FPGA_I2C_SDAT,
        output logic              AUD_XCK,
        input logic               AUD_ADCLRCK,
        input logic               AUD_DACLRCK,
        input logic               AUD_BCLK,
        input logic               AUD_ADCDAT,
        output logic              AUD_DACDAT,

        //Driver IO ports
        input logic               clk,
        input logic               reset,
        input logic [31:0]        writedata,
        input logic               write,
```

```
        input logic                read,
        input                      chipselect,
        input logic [15:0]         address,
        output logic [31:0]        readdata,

        //filter control
        output logic [15:0]        raw_data,
       input  logic [31:0]        filtered_signal,
        //button
       input logic [3:0]  KEY

        );
    //Audio Controller
    reg [15:0]      dac_left_in;
    reg [15:0]      dac_right_in;
    wire [15:0]     adc_left_out;
    wire [15:0]     adc_right_out;
    wire advance;

    // wire advance;

    //Device drivers from Altera modified by Professor Scott Hauck and Kyle Gagner
in Verilog
    audio_driver aDriver(
        .CLOCK_50(clk),
        .reset(reset),
        .dac_left(dac_left_in),
        .dac_right(dac_right_in),
        .adc_left(adc_left_out),
        .adc_right(adc_right_out),
        .advance(advance),
        .FPGA_I2C_SCLK(FPGA_I2C_SCLK),
        .FPGA_I2C_SDAT(FPGA_I2C_SDAT),
        .AUD_XCK(AUD_XCK),
        .AUD_DACLRCK(AUD_DACLRCK),
        .AUD_ADCLRCK(AUD_ADCLRCK),
        .AUD_BCLK(AUD_BCLK),
        .AUD_ADCDAT(AUD_ADCDAT),
        .AUD_DACDAT(AUD_DACDAT)
        );

    //Instantiate hex decoders
    logic [23:0]    hexout_buffer;
    hex7seg h5( .a(buffer[3:0]),.y(HEX5) ), // left digit
            h4( .a(KEY[3:0]),.y(HEX4) ),
            h3( .a(sound),.y(HEX3) ),
            h2( .a(play),.y(HEX2) ),
            h1( .a(KEY[3:0]),.y(HEX1) ),
            h0( .a(buffer[3:0]),.y(HEX0) );

  // Debounce variables
wire [15:0] buffer;
    always_comb begin
       //audioInMono = (adc_right_out>>1) + (adc_left_out>>1);
```

```
       buffer=adc_right_out;
    End

wire [15:0] sound_address;

reg [15:0] sound_data;
wire [1:0] sound;

wire play;
wire [1:0] prev_sound;
audio_rom(sound_address, clk, sound_data);

      always_ff @(posedge clk) begin : IOcalls
           if (advance) begin
    raw_data <= buffer;

    // Detect if KEY has changed
    if (sound != prev_sound) begin
    if (play==0)begin
        case (sound)
            0:begin
          dac_left_in <= 16'd0;
            dac_right_in <= 16'd0;
            play<=0;
        end

            1: begin
                sound_address <= 16'd0;
                play<=1;

            end
            2: begin
                sound_address <= 16'd18232;
                play<=1;
            end
            // Add more cases if needed
        endcase
    end else begin
          sound_address <= sound_address + 1'b1;
  end
    end else begin
      if((sound_address!=16'd18231)&&(sound_address!=16'd46494))begin
        sound_address <= sound_address + 1'b1;
      end
    end

    // Update the previous KEY value
    prev_sound <= sound;
    if((sound_address==16'd18231)||(sound_address==16'd46494))begin
       dac_left_in <= 16'd0;
        dac_right_in <= 16'd0;
        play<=0;
     end
    if ((sound == 0)&& (play==0))begin
```

```
       dac_left_in <= 16'd0;
        dac_right_in <= 16'd0;

    end else begin
       dac_left_in <= sound_data;
        dac_right_in <= sound_data;
    end
end
        if (chipselect && read) begin
            case (address)
                16'h0000 : begin
                    readdata[31:0] <=  filtered_signal;
                end
             16'h0001 : begin
                    readdata[31:0] <=  {28'b0,KEY};
                end
            endcase
        end
       if (chipselect && write) begin
            case (address)
                16'h0002 : begin
                    sound<=writedata[1:0];
                end
            endcase
        end
    end
   logic [31:0]    driverReading = 31'd0;
    wire sampleBeingTaken;
    assign sampleBeingTaken = driverReading[0];

    //Map timer(Sample) counter output
    parameter readOutSize = 16'hffff;
    //Sample inputs/Audio passthrough

endmodule
```

filter.sv

```
module filter(
input logic clk,
input logic reset,
input logic [15:0] input_signal,
output logic [31:0] filtered_signal);

       parameter word_width = 16;
       parameter order =17;
       wire signed [15:0] coef [0:16];
       reg signed [15:0] delay_pipeline[0:16];
       assign coef[0] = 208;
       assign coef[1] = 339;
       assign coef[2] = 703;
       assign coef[3] = 1296;
       assign coef[4] = 2055;
```

```
assign coef[5] = 2864;
assign coef[6] = 3585;
assign coef[7] = 4083;
 assign coef[8]= 4260;
assign coef[9] = 4083;
assign coef[10] = 3585;
assign coef[11] = 2864;
assign coef[12] = 2055;
assign coef[13] = 1296;
assign coef[14] = 703;
assign coef[15] = 339;
assign coef[16] = 208;

reg signed [31:0]product[0:16];
reg signed [31:0]sum;
reg signed [15:0]data_in_buf;
reg signed [31:0]data_out;
always_ff@(posedge clk or posedge reset)
begin

  if (reset)
   begin
    data_in_buf<=0;
   end
  else begin
    data_in_buf<=input_signal;
   end
end
always_ff@(posedge clk or posedge reset)
  begin
    if(reset) begin
       delay_pipeline[0]<=0;
       delay_pipeline[1]<=0;
       delay_pipeline[2]<=0;
       delay_pipeline[3]<=0;
       delay_pipeline[4]<=0;
       delay_pipeline[5]<=0;
       delay_pipeline[6]<=0;
       delay_pipeline[7]<=0;
       delay_pipeline[8]<=0;
       delay_pipeline[9]<=0;
       delay_pipeline[10]<=0;
       delay_pipeline[11]<=0;
       delay_pipeline[12]<=0;
       delay_pipeline[13]<=0;
       delay_pipeline[14]<=0;
       delay_pipeline[15]<=0;
       delay_pipeline[16]<=0;

      end
      else begin
      delay_pipeline[0]<=data_in_buf;
      delay_pipeline[1]<=delay_pipeline[0];
      delay_pipeline[2]<=delay_pipeline[1];
```

```
        delay_pipeline[3]<=delay_pipeline[2];
        delay_pipeline[4]<=delay_pipeline[3];
        delay_pipeline[5]<=delay_pipeline[4];
        delay_pipeline[6]<=delay_pipeline[5];
        delay_pipeline[7]<=delay_pipeline[6];
        delay_pipeline[8]<=delay_pipeline[7];
        delay_pipeline[9]<=delay_pipeline[8];
        delay_pipeline[10]<=delay_pipeline[9];
        delay_pipeline[11]<=delay_pipeline[10];
        delay_pipeline[12]<=delay_pipeline[11];
        delay_pipeline[13]<=delay_pipeline[12];
        delay_pipeline[14]<=delay_pipeline[13];
        delay_pipeline[15]<=delay_pipeline[14];
        delay_pipeline[16]<=delay_pipeline[15];
        end
      end
  always_ff@(posedge clk or posedge reset)
  begin
    if (reset)begin
    product[0]<=0;
    product[1]<=0;
    product[2]<=0;
    product[3]<=0;
    product[4]<=0;
    product[5]<=0;
    product[6]<=0;
    product[7]<=0;
    product[8]<=0;
    product[9]<=0;
    product[10]<=0;
    product[11]<=0;
    product[12]<=0;
    product[13]<=0;
    product[14]<=0;
    product[15]<=0;
    product[16]<=0;

    end
    else begin
    product[0]<=coef[0]*delay_pipeline[0];
    product[1]<=coef[1]*delay_pipeline[1];
    product[2]<=coef[2]*delay_pipeline[2];
    product[3]<=coef[3]*delay_pipeline[3];
    product[4]<=coef[4]*delay_pipeline[4];
    product[5]<=coef[5]*delay_pipeline[5];
    product[6]<=coef[6]*delay_pipeline[6];
    product[7]<=coef[7]*delay_pipeline[7];
    product[8]<=coef[8]*delay_pipeline[8];
    product[9]<=coef[9]*delay_pipeline[9];
    product[10]<=coef[10]*delay_pipeline[10];
    product[11]<=coef[11]*delay_pipeline[11];
    product[12]<=coef[12]*delay_pipeline[12];
    product[13]<=coef[13]*delay_pipeline[13];
    product[14]<=coef[14]*delay_pipeline[14];
```

```
          product[15]<=coef[15]*delay_pipeline[15];
          product[16]<=coef[16]*delay_pipeline[16];
          end
        end

        always_ff@(posedge clk or posedge reset)
        begin
          if(reset)
          begin
          sum<=0;
          end
          else begin

sum<=product[0]+product[1]+product[2]+product[3]+product[4]+product[5]+product[6]+p
roduct[7]+product[8]+product[9]+product[10]+product[11]+product[12]+product[13]+pro
duct[14]+product[15]+product[16];
          end
        end
        always_ff@(posedge clk or posedge reset)
        begin
          if(reset)
          begin
          data_out<=0;
          end
          else begin
         filtered_signal = sum;
          end
        end
endmodule
```

global_variables.sv

```
// Audio Codec Macros
`define AUDIO_IN_GAIN 9'h010
`define AUDIO_OUT_GAIN 9'h061

`define SAMPLE_RATE_CNTRL 9'd0  //No particularly helpful, but see page 44 of
datasheet of more info:
https://statics.cirrus.com/pubs/proDatasheet/WM8731_v4.9.pdf
```

audio_rom.v

```
// megafunction wizard: %ROM: 1-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: audio_rom.v
// Megafunction Name(s):
//                  altsyncram
//
// Simulation Library Files(s):
```

```
//                  altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 21.1.0 Build 842 10/21/2021 SJ Lite Edition
// ************************************************************


//Copyright (C) 2021  Intel Corporation. All rights reserved.
//Your use of Intel Corporation's design tools, logic functions
//and other software and tools, and any partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Intel Program License
//Subscription Agreement, the Intel Quartus Prime License Agreement,
//the Intel FPGA IP License Agreement, or other applicable license
//agreement, including, without limitation, that your use is for
//the sole purpose of programming logic devices manufactured by
//Intel and sold by Intel or its authorized distributors.  Please
//refer to the applicable agreement for further details, at
//https://fpgasoftware.intel.com/eula.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module audio_rom (
	address,
	clock,
	q);

	input  [15:0]  address;
	input    clock;
	output [15:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
	tri1     clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

	wire [15:0] sub_wire0;
	wire [15:0] q = sub_wire0[15:0];

	altsyncram   altsyncram_component (
				.address_a (address),
				.clock0 (clock),
				.q_a (sub_wire0),
				.aclr0 (1'b0),
				.aclr1 (1'b0),
				.address_b (1'b1),
```

```
                        .addressstall_a (1'b0),
                        .addressstall_b (1'b0),
                        .byteena_a (1'b1),
                        .byteena_b (1'b1),
                        .clock1 (1'b1),
                        .clocken0 (1'b1),
                        .clocken1 (1'b1),
                        .clocken2 (1'b1),
                        .clocken3 (1'b1),
                        .data_a ({16{1'b1}}),
                        .data_b (1'b1),
                        .eccstatus (),
                        .q_b (),
                        .rden_a (1'b1),
                        .rden_b (1'b1),
                        .wren_a (1'b0),
                        .wren_b (1'b0));
    defparam
            altsyncram_component.address_aclr_a = "NONE",
            altsyncram_component.clock_enable_input_a = "BYPASS",
            altsyncram_component.clock_enable_output_a = "BYPASS",
            altsyncram_component.init_file = "output.mif",
            altsyncram_component.intended_device_family = "Cyclone V",
            altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
            altsyncram_component.lpm_type = "altsyncram",
            altsyncram_component.numwords_a = 46495,
            altsyncram_component.operation_mode = "ROM",
            altsyncram_component.outdata_aclr_a = "NONE",
            altsyncram_component.outdata_reg_a = "UNREGISTERED",
            altsyncram_component.widthad_a = 16,
            altsyncram_component.width_a = 16,
            altsyncram_component.width_byteena_a = 1;


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
// Retrieval info: PRIVATE: AclrByte NUMERIC "0"
// Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: Clken NUMERIC "0"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
```

```
// Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
// Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
// Retrieval info: PRIVATE: MIFfilename STRING "combined_audio.mif"
// Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "16000"
// Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
// Retrieval info: PRIVATE: RegAddr NUMERIC "1"
// Retrieval info: PRIVATE: RegOutput NUMERIC "0"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
// Retrieval info: PRIVATE: SingleClock NUMERIC "1"
// Retrieval info: PRIVATE: UseDQRAM NUMERIC "0"
// Retrieval info: PRIVATE: WidthAddr NUMERIC "14"
// Retrieval info: PRIVATE: WidthData NUMERIC "8"
// Retrieval info: PRIVATE: rden NUMERIC "0"
// Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
// Retrieval info: CONSTANT: ADDRESS_ACLR_A STRING "NONE"
// Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "BYPASS"
// Retrieval info: CONSTANT: INIT_FILE STRING
"../../audio_with_interface/combined_audio.mif"
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
// Retrieval info: CONSTANT: LPM_HINT STRING "ENABLE_RUNTIME_MOD=NO"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
// Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "16000"
// Retrieval info: CONSTANT: OPERATION_MODE STRING "ROM"
// Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
// Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED"
// Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "14"
// Retrieval info: CONSTANT: WIDTH_A NUMERIC "8"
// Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
// Retrieval info: USED_PORT: address 0 0 14 0 INPUT NODEFVAL "address[13..0]"
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
// Retrieval info: USED_PORT: q 0 0 8 0 OUTPUT NODEFVAL "q[7..0]"
// Retrieval info: CONNECT: @address_a 0 0 14 0 address 0 0 14 0
// Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
// Retrieval info: CONNECT: q 0 0 8 0 @q_a 0 0 8 0
// Retrieval info: GEN_FILE: TYPE_NORMAL audio_rom.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL audio_rom.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL audio_rom.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL audio_rom.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL audio_rom_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL audio_rom_bb.v FALSE
// Retrieval info: LIB_FILE: altera_mf
```

Software

aud.c

```
/* * Device driver for the audio recognition hardware
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
```

```
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod aud.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree aud.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "aud.h"

#define DRIVER_NAME "aud"

/* Device registers */
#define AUD_AMP(x) (x)       //0
#define MENU_BUTTON(x) ((x)+4)  //1
#define SOUND(x) ((x)+8)   // 2

/*
 * Information about our device
 */
struct aud_dev {
       struct resource res; /* Resource: our registers */
       void __iomem *virtbase; /* Where registers can be accessed in memory */
       aud_mem_t memory;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
// static void read_audio(aud_amp_t *audio)
// {
//     audio->amplitude = ioread32(AUD_AMP(dev.virtbase));
       // dev.audio = *audio;
// }
```

```
static void write_sound(aud_mem_t *memory)
{
	iowrite32(memory->sound, SOUND(dev.virtbase));
	dev.memory = *memory;
}


static void read_memory(aud_mem_t *memory)
{
	memory->data = ioread32(AUD_AMP(dev.virtbase));
}

static void read_button(aud_mem_t *memory){
	memory->button_value = ioread32(MENU_BUTTON(dev.virtbase));
}
	// dev.audio = *audio;
/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long aud_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
	aud_arg_t vla;
	switch (cmd) {
		case AUD_READ_DATA:
			read_memory(&vla.memory);
			if (copy_to_user((aud_arg_t *) arg, &vla, sizeof(aud_arg_t)))
				return -EACCES;
			break;
		case AUD_WRITE_SOUND:
			if (copy_from_user(&vla, (aud_arg_t *) arg, sizeof(aud_arg_t)))
				return -EACCES;
			write_sound(&vla.memory);
			break;
		case AUD_READ_BUTTON:
			read_button(&vla.memory);
			if (copy_to_user((aud_arg_t *) arg, &vla, sizeof(aud_arg_t)))
				return -EACCES;
			break;
		default:
			return -EINVAL;
	}
	return 0;
}

/* The operations our device knows how to do */
static const struct file_operations aud_fops = {
	.owner		= THIS_MODULE,
	.unlocked_ioctl = aud_ioctl,
};
```

```
/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice aud_misc_device = {
	.minor		= MISC_DYNAMIC_MINOR,
	.name		= DRIVER_NAME,
	.fops		= &aud_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init aud_probe(struct platform_device *pdev)
{
	int ret;

	/* Register ourselves as a misc device: creates /dev/aud */
	ret = misc_register(&aud_misc_device);

	/* Get the address of our registers from the device tree */
	ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
	if (ret) {
		ret = -ENOENT;
		goto out_deregister;
	}

	/* Make sure we can use these registers */
	if (request_mem_region(dev.res.start, resource_size(&dev.res),
			       DRIVER_NAME) == NULL) {
		ret = -EBUSY;
		goto out_deregister;
	}

	/* Arrange access to our registers */
	dev.virtbase = of_iomap(pdev->dev.of_node, 0);
	if (dev.virtbase == NULL) {
		ret = -ENOMEM;
		goto out_release_mem_region;
	}

	return 0;

out_release_mem_region:
	release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
	misc_deregister(&aud_misc_device);
	return ret;
}

/* Clean-up code: release resources */
static int aud_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&aud_misc_device);
```

```
	return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id aud_of_match[] = {
	{ .compatible = "csee4840,aud-1.0" },
	{},
};
MODULE_DEVICE_TABLE(of, aud_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver aud_driver = {
	.driver	= {
		.name  = DRIVER_NAME,
		.owner = THIS_MODULE,
		.of_match_table = of_match_ptr(aud_of_match),
	},
	.remove	= __exit_p(aud_remove),
};

/* Called when the module is loaded: set things up */
static int __init aud_init(void)
{
	pr_info(DRIVER_NAME ": init\n");
	return platform_driver_probe(&aud_driver, aud_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit aud_exit(void)
{
	platform_driver_unregister(&aud_driver);
	pr_info(DRIVER_NAME ": exit\n");
}

module_init(aud_init);
module_exit(aud_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("asy2126");
MODULE_DESCRIPTION("Audio Input driver");
```

aud.h

```
#ifndef _AUD_H
#define _AUD_H

#include <linux/ioctl.h>

// recieve
// typedef struct {
//     int data;
```

```
// } aud_amp_t;

typedef struct {
       int data;
       int sound;

  int button_value;
} aud_mem_t;

typedef struct {
//     aud_amp_t audio;
       aud_mem_t memory;
} aud_arg_t;

typedef struct {
  int data;
  int sound;
  int button_value;
} mem;


#define AUD_MAGIC 'q'

/* ioctls and their arguments */
#define AUD_READ_DATA          _IOR(AUD_MAGIC, 1, aud_arg_t *)
#define AUD_READ_BUTTON     _IOR(AUD_MAGIC, 2, aud_arg_t *)
#define AUD_WRITE_SOUND     _IOW(AUD_MAGIC, 3, aud_arg_t *)

#endif
```

interfaces.c

```
#include <stdio.h>
#include "interfaces.h"
#include "vga_bird.h"
#include "aud.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

void send_sound(const aud_mem_t *c, const int aud_fd) {
       aud_arg_t amt;
       amt.memory = *c;
       if (ioctl(aud_fd, AUD_WRITE_SOUND, &amt)) {
              perror("ioctl(AUD_WRITE_SOUND) failed");
              return;
       }
}
```

```
int get_aud_data(const int aud_fd) {
	aud_arg_t aat;
	if (ioctl(aud_fd, AUD_READ_DATA, &aat)) {
		perror("ioctl(AUD_READ_DATA) failed");
		return 0;
	}
	return aat.memory.data;
}

int get_button_value(const int aud_fd) {
	aud_arg_t aat;
	if (ioctl(aud_fd, AUD_READ_BUTTON, &aat)) {
		perror("ioctl(AUD_READ_BUTTON) failed");
		return 0;
	}
	return aat.memory.button_value;
}

void send_sprite_positions(const vga_bird_data_t *c, const int vga_bird_fd) {
	vga_bird_arg_t vzat;
	vzat.packet = *c;
	if (ioctl(vga_bird_fd, VGA_BIRD_WRITE_PACKET, &vzat)) {
		perror("ioctl(VGA_BIRD_WRITE_PACKET) failed");
		return;
	}
}
void send_score(const vga_bird_data_t *c, const int vga_bird_fd) {
	vga_bird_arg_t vzat;
	vzat.packet = *c;
	if (ioctl(vga_bird_fd, VGA_BIRD_WRITE_SCORE, &vzat)) {
		perror("ioctl(VGA_BIRD_WRITE_SCORE) failed");
		return;
	}
}
void send_combo(const vga_bird_data_t *c, const int vga_bird_fd) {
	vga_bird_arg_t vzat;
	vzat.packet = *c;
	if (ioctl(vga_bird_fd, VGA_BIRD_WRITE_COMBO, &vzat)) {
		perror("ioctl(VGA_BIRD_WRITE_COMBO) failed");
		return;
	}
}
```

interfaces.h

```
#ifndef _VGA_BIRD_H_
#define _VGA_BIRD_H_

#include <linux/ioctl.h>
#include "vga_bird.h"
#include "aud.h"

void send_sound(const aud_mem_t *c, const int aud_fd);
```

```
int get_aud_data(const int aud_fd);

int get_button_value(const int aud_fd);

void send_sprite_positions(const vga_bird_data_t *c, const int vga_bird_fd);

void send_score(const vga_bird_data_t *c, const int vga_bird_fd);
void send_combo(const vga_bird_data_t *c, const int vga_bird_fd);

#endif
```

vga_bird.c

```
/* * Device driver for the VGA video generator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod vga_bird.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree vga_bird.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_bird.h"

#define DRIVER_NAME "vga_bird"

/* Device registers */
```

```
#define SCORE_N(x) ((x)+12) //3
#define COMBO_N(x) ((x)+16) //4
#define DATA_N(x) ((x)+20) //5

/*
 * Information about our device
 */
struct vga_bird_dev {
	struct resource res; /* Resource: our registers */
	void __iomem *virtbase; /* Where registers can be accessed in memory */
	vga_bird_data_t packet;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_packet(vga_bird_data_t *packet)
{
	int i;
	for (i = 0; i < SIZE; i++) {
		iowrite32(packet->data[i], DATA_N(dev.virtbase) );
	}
	dev.packet = *packet;
}

static void write_score(vga_bird_data_t *packet)
{
	iowrite32(packet->score, SCORE_N(dev.virtbase));
	dev.packet = *packet;
}

static void write_combo(vga_bird_data_t *packet)
{
	iowrite32(packet->combo, COMBO_N(dev.virtbase));
	dev.packet = *packet;
}
/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long vga_bird_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
	vga_bird_arg_t vla;

	switch (cmd) {
		case VGA_BIRD_WRITE_PACKET:
			if (copy_from_user(&vla, (vga_bird_arg_t *) arg,
sizeof(vga_bird_arg_t)))
				return -EACCES;
			write_packet(&vla.packet);
			break;
		case VGA_BIRD_WRITE_SCORE:
```

```
                        if (copy_from_user(&vla, (vga_bird_arg_t *) arg,
sizeof(vga_bird_arg_t)))
                                return -EACCES;
                        write_score(&vla.packet);
                        break;
                case VGA_BIRD_WRITE_COMBO:
                        if (copy_from_user(&vla, (vga_bird_arg_t *) arg,
sizeof(vga_bird_arg_t)))
                                return -EACCES;
                        write_combo(&vla.packet);
                        break;
                case VGA_BIRD_READ_PACKET:
                        vla.packet = dev.packet;
                        if (copy_to_user((vga_bird_arg_t *) arg, &vla,
sizeof(vga_bird_arg_t)))
                                return -EACCES;
                        break;
                default:
                        return -EINVAL;
        }

        return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_bird_fops = {
        .owner          = THIS_MODULE,
        .unlocked_ioctl = vga_bird_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice vga_bird_misc_device = {
        .minor          = MISC_DYNAMIC_MINOR,
        .name           = DRIVER_NAME,
        .fops           = &vga_bird_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_bird_probe(struct platform_device *pdev)
{
        int ret;

        /* Register ourselves as a misc device: creates /dev/vga_bird */
        ret = misc_register(&vga_bird_misc_device);

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
                ret = -ENOENT;
                goto out_deregister;
        }
```

```
	/* Make sure we can use these registers */
	if (request_mem_region(dev.res.start, resource_size(&dev.res),
			       DRIVER_NAME) == NULL) {
		ret = -EBUSY;
		goto out_deregister;
	}

	/* Arrange access to our registers */
	dev.virtbase = of_iomap(pdev->dev.of_node, 0);
	if (dev.virtbase == NULL) {
		ret = -ENOMEM;
		goto out_release_mem_region;
	}

	return 0;

out_release_mem_region:
	release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
	misc_deregister(&vga_bird_misc_device);
	return ret;
}

/* Clean-up code: release resources */
static int vga_bird_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&vga_bird_misc_device);
	return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_bird_of_match[] = {
	{ .compatible = "csee4840,vga_bird-1.0" },
	{},
};
MODULE_DEVICE_TABLE(of, vga_bird_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_bird_driver = {
	.driver	= {
		.name  = DRIVER_NAME,
		.owner = THIS_MODULE,
		.of_match_table = of_match_ptr(vga_bird_of_match),
	},
	.remove	= __exit_p(vga_bird_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_bird_init(void)
```

```
{
	pr_info(DRIVER_NAME ": init\n");
	return platform_driver_probe(&vga_bird_driver, vga_bird_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit vga_bird_exit(void)
{
	platform_driver_unregister(&vga_bird_driver);
	pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_bird_init);
module_exit(vga_bird_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Screaming Bird, Columbia University");
MODULE_DESCRIPTION("VGA bird driver");
```

vga_bird.h

```
#ifndef _VGA_BIRD_H
#define _VGA_BIRD_H

#include <linux/ioctl.h>

//number of supported sprites
#define SIZE 42

typedef struct {
  int data[SIZE];
  int score;
  int combo;
} vga_bird_data_t;

typedef struct {
	vga_bird_data_t packet;
} vga_bird_arg_t;

typedef struct {
	int x, y, dx, dy, id, index, hit;
} sprite;

typedef struct {
  int pipe_num, top_pipe_max_index, bottom_pipe_max_index;
} vga_pipe_position_t;

#define VGA_BIRD_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_BIRD_WRITE_PACKET _IOW(VGA_BIRD_MAGIC, 4, vga_bird_arg_t *)
#define VGA_BIRD_WRITE_SCORE _IOW(VGA_BIRD_MAGIC, 5, vga_bird_arg_t *)
#define VGA_BIRD_WRITE_COMBO _IOW(VGA_BIRD_MAGIC, 6, vga_bird_arg_t *)
```

```
#define VGA_BIRD_READ_PACKET _IOR(VGA_BIRD_MAGIC, 7, vga_bird_arg_t *)

#endif
```

test2_update.c

```
/*
 * Userspace program that communicates with the aud and vga_bird device driver
 * through ioctls
 * Columbia University
 */

#include <stdio.h>
#include "interfaces.h"
#include "vga_bird.h"
#include "aud.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

#define X_MAX 639
#define Y_MAX 479
#define NEXT 14
#define CONFIRM 13
#define ARROW_INDEX 22
#define BIRD_ID 28

int vga_bird_fd;
int aud_fd;
FILE *log_file;

enum difficulty {EASY, MEDIUM, HARD};


int check_bird_pass_pipe(sprite *sprites, vga_pipe_position_t *pipe_info_first,
vga_pipe_position_t *pipe_info_second) {
    int bird_x = sprites[15].x;
       int first_pipe_position_x = sprites[16].x;
       int second_pipe_position_x = sprites[29].x;
    int bird_passed = 1;

    // Check if the bird has passed the first pipe
    if ((pipe_info_first->pipe_num != 0 && bird_x ==
sprites[pipe_info_first->top_pipe_max_index].x + 32 )||(pipe_info_second->pipe_num
!= 0 && bird_x == sprites[pipe_info_second->top_pipe_max_index].x + 32)) {
        return bird_passed;
       }
    return 0;
```

```
}
void updateBall(sprite *obj) {
	obj->x += obj->dx;
	obj->y += obj->dy;
}

int check_bird_score(sprite *sprites, vga_pipe_position_t *pipe_info_second, int
score){
	int first_pipe_position_x = sprites[16].x;
	int second_pipe_position_x = sprites[29].x;
	if (first_pipe_position_x == 200) {
		score++;
	} else if (second_pipe_position_x == 200) {
		score++;
	}
	return score;
}

int check_bird_position(sprite *sprites, vga_pipe_position_t *pipe_info_first,
vga_pipe_position_t *pipe_info_second) {
	int bird_position_x = sprites[15].x;
	int first_pipe_position_x = sprites[16].x;
	int second_pipe_position_x = sprites[29].x;
	int second_top_pipe_position_y;
	int second_bottom_pipe_position_y;
	if (pipe_info_second->pipe_num != 0) {
		second_top_pipe_position_y =
sprites[pipe_info_second->top_pipe_max_index].y;
		second_bottom_pipe_position_y =
sprites[pipe_info_second->bottom_pipe_max_index].y;
	}
	int first_top_pipe_position_y =
sprites[pipe_info_first->top_pipe_max_index].y;
	int first_bottom_pipe_position_y =
sprites[pipe_info_first->bottom_pipe_max_index].y;

	int bird_position_y = sprites[15].y;
	if (bird_position_y >= 480 || bird_position_y <= 0) {
		// if this bird hits the floor.
		printf("bird hits the floor.\n");
		return 1;
	}
	//top 4, bottom 5

	if (first_pipe_position_x > (bird_position_x - 32) && first_pipe_position_x
< (bird_position_x + 32)) {
		if ((bird_position_y + 4) < (first_top_pipe_position_y + 32) ||
(bird_position_y + 32 - 5) > first_bottom_pipe_position_y) {
			printf("bird hits the first pipe.\n");
			printf("pipe info first top pipe max index: %d, bottom pipe max
index: %d\n", pipe_info_first->top_pipe_max_index,
pipe_info_first->bottom_pipe_max_index);
			printf("bird position x: %d, y: %d\n", bird_position_x,
bird_position_y);
```

```
                    printf("first pipe position x: %d, top y: %d, bottom y: %d\n",
first_pipe_position_x, first_top_pipe_position_y, first_bottom_pipe_position_y);
                    aud_mem_t c;
                    c.sound = 2;
                    printf("send sound\n");
                    send_sound(&c, aud_fd);
                    return 1;
              }
       }

       if (pipe_info_second->pipe_num != 0) {
              if (second_pipe_position_x > (bird_position_x - 32) &&
second_pipe_position_x < (bird_position_x + 32)) {
                    if ((bird_position_y + 4) < (second_top_pipe_position_y + 32)
|| (bird_position_y + 32 - 5) > second_bottom_pipe_position_y) {
                           printf("bird hits the second pipe.\n");
                           printf("pipe info second top pipe max index: %d, bottom
pipe max index: %d\n", pipe_info_second->top_pipe_max_index,
pipe_info_second->bottom_pipe_max_index);
                           printf("bird position x: %d, y: %d\n", bird_position_x,
bird_position_y);
                           printf("second pipe position x: %d, top y: %d, bottom y:
%d\n", second_pipe_position_x, second_top_pipe_position_y,
second_bottom_pipe_position_y);
                           aud_mem_t c;
                           c.sound = 2;
                           printf("send sound\n");
                           send_sound(&c, aud_fd);
                           return 1;
                    }
              }
       }



       return 0;
}


// The idea is that every pipe will be at most 13 long, so the sprites will be from
16 to 28. Another will be from 29 to 41.
void create_pipe(sprite *sprites, vga_pipe_position_t *pipe_info, int
pipe_index_start, int difficulty_level) {
       // head 30, tail 31.

       // use the sprites starts from 16
       // first do the down pipe.
       int dx = -1;
       if (difficulty_level == EASY) {
              dx = -1;
       } else if (difficulty_level == MEDIUM) {
              dx = -2;
       } else if (difficulty_level == HARD) {
              dx = -3;
```

```
	}
	int pipe_num;
	int top_pipe_num;
	int bottom_pipe_num;
	pipe_num = 6 + rand() % (10 - 6 + 1);
	top_pipe_num = 2 + rand() % (pipe_num - 6 + 1);
	//every pipe has at least 3 with at least one top and one bottom.
	bottom_pipe_num = pipe_num - top_pipe_num;
	if (difficulty_level == EASY) {
		pipe_num = 6 + rand() % (10 - 6 + 1);
		top_pipe_num = 2 + rand() % (pipe_num - 6 + 1);
		//every pipe has at least 3 with at least one top and one bottom.
		bottom_pipe_num = pipe_num - top_pipe_num;
		dx = -1;
	} else if (difficulty_level == MEDIUM) {
		pipe_num = 7 + rand() % (11 - 7 + 1);
		top_pipe_num = 2 + rand() % (pipe_num - 6 + 1);
		bottom_pipe_num = pipe_num - top_pipe_num;
		dx = -2;
	} else if (difficulty_level == HARD) {
		pipe_num = 8 + rand() % (10 - 6 + 1);
		top_pipe_num = 2 + rand() % (pipe_num - 8 + 1);
		//every pipe has at least 3 with at least one top and one bottom.
		bottom_pipe_num = pipe_num - top_pipe_num;
		dx=-4;
	}

	// for (int i = 0; i < pipe_num; i++){
	//     sprites[16+i].id = 30;
	// }
	// add the refresh part.
	for (int i = pipe_index_start; i < pipe_index_start+13; i++) {
		sprites[i].x = 630;
		sprites[i].y = 470;
		sprites[i].dx = 0;
		sprites[i].dy = 0;
		sprites[i].id = 0;
		sprites[i].hit = 1;
		sprites[i].index = i;
	}

	for (int i = 0; i < top_pipe_num - 1; i++) {
		sprites[pipe_index_start+i].id = 30;
		sprites[pipe_index_start+i].x = 480 - 32;
		sprites[pipe_index_start+i].y = 32 * i;
		sprites[pipe_index_start+i].dx = dx;
		sprites[pipe_index_start+i].dy = 0;
		sprites[pipe_index_start+i].hit = 1;
		sprites[pipe_index_start+i].index = pipe_index_start+i;
	}

	//head.
	sprites[pipe_index_start+top_pipe_num-1].id = 29;
	sprites[pipe_index_start+top_pipe_num-1].x = 480 - 32;
```

```
	sprites[pipe_index_start+top_pipe_num-1].y = 32 * (top_pipe_num - 1);
	sprites[pipe_index_start+top_pipe_num-1].dx = dx;
	sprites[pipe_index_start+top_pipe_num-1].dy = 0;
	sprites[pipe_index_start+top_pipe_num-1].hit = 1;
	sprites[pipe_index_start+top_pipe_num-1].index =
pipe_index_start+top_pipe_num-1;
	printf("top pipe index: %d\n", pipe_index_start+top_pipe_num-1);
	printf("top pipe y is: %d\n", sprites[pipe_index_start+top_pipe_num-1].y);

	for (int i = 0; i < bottom_pipe_num-1; i++) {
		sprites[pipe_index_start+top_pipe_num+i].id = 30;
		sprites[pipe_index_start+top_pipe_num+i].x = 480 - 32;
		sprites[pipe_index_start+top_pipe_num+i].y = 480 - 32 - 32 * i;
		sprites[pipe_index_start+top_pipe_num+i].dx = dx;
		sprites[pipe_index_start+top_pipe_num+i].dy = 0;
		sprites[pipe_index_start+top_pipe_num+i].hit = 1;
		sprites[pipe_index_start+top_pipe_num+i].index =
pipe_index_start+top_pipe_num+i;
	}

	sprites[pipe_index_start+top_pipe_num+bottom_pipe_num-1].id = 29;
	sprites[pipe_index_start+top_pipe_num+bottom_pipe_num-1].x = 480 - 32;
	sprites[pipe_index_start+top_pipe_num+bottom_pipe_num-1].y = 480 - 32 - 32 *
(bottom_pipe_num - 1);
	sprites[pipe_index_start+top_pipe_num+bottom_pipe_num-1].dx = dx;
	sprites[pipe_index_start+top_pipe_num+bottom_pipe_num-1].dy = 0;
	sprites[pipe_index_start+top_pipe_num+bottom_pipe_num-1].hit = 1;
	sprites[pipe_index_start+top_pipe_num+bottom_pipe_num-1].index =
pipe_index_start+top_pipe_num+bottom_pipe_num-1;
	printf("bottom pipe index: %d\n",
pipe_index_start+top_pipe_num+bottom_pipe_num-1);
	printf("bottom pipe y is: %d\n",
sprites[pipe_index_start+top_pipe_num+bottom_pipe_num-1].y);
	pipe_info->pipe_num = pipe_num;
	pipe_info->top_pipe_max_index = pipe_index_start+top_pipe_num-1;
	pipe_info->bottom_pipe_max_index =
pipe_index_start+top_pipe_num+bottom_pipe_num-1;
}


void create_pipe_condition(sprite *sprites, vga_pipe_position_t *pipe_info_first,
vga_pipe_position_t *pipe_info_second, int difficulty_level) {
	int first_pipe_position_x = sprites[16].x;
	int second_pipe_position_x = sprites[29].x;
	if (first_pipe_position_x == 224) {
		printf("second pipe position x is created.\n");
		create_pipe(sprites, pipe_info_second, 29, difficulty_level);
	} else if (second_pipe_position_x == 224) {
		printf("first pipe position x is created.\n");
		create_pipe(sprites, pipe_info_first, 16, difficulty_level);
	}
}
void menu_setup(sprite *sprites){
	sprites[1].id = 20; //M 20
```

```
	sprites[2].id = 18; //E 18
	sprites[3].id = 25; //N
	sprites[4].id = 19; //U

	for (int i = 1; i < 5; i++) {
		sprites[i].x = 108 + 32*(i-1);
		sprites[i].y = 120;
		sprites[i].dx = 0;
		sprites[i].dy = 0;
		sprites[i].hit = 1;
		sprites[i].index = i;
	}

	// START
	sprites[5].id = 22; //S
	sprites[6].id = 23; //T
	sprites[7].id = 33; //A
	sprites[8].id = 34; //R
	sprites[9].id = 23; //T

	for (int i = 5; i < 10; i++) {
		sprites[i].x = 140 + 32*(i-5);
		sprites[i].y = 180;
		sprites[i].dx = 0;
		sprites[i].dy = 0;
		sprites[i].hit = 1;
		sprites[i].index = i;
	}

	// SETTING
	sprites[10].id = 22; //S
	sprites[11].id = 18; //E
	sprites[12].id = 23; //T
	sprites[13].id = 23; //T
	sprites[14].id = 24; //I
	sprites[15].id = 25; //N
	sprites[16].id = 31; //G

	for (int i = 10; i < 17; i++) {
		sprites[i].x = 140 + 32*(i-10);
		sprites[i].y = 240;
		sprites[i].dx = 0;
		sprites[i].dy = 0;
		sprites[i].hit = 1;
		sprites[i].index = i;
	}

	//add a arrow
	// T I N G U
	//arrow index will always be 22
	// sprites[ARROW_INDEX].id = 28; // Arrow is bird now
	// sprites[ARROW_INDEX].x = 370;
	// sprites[ARROW_INDEX].y = 180;
	// sprites[ARROW_INDEX].dx = 0;
```

```
	// sprites[ARROW_INDEX].dy = 0;
	// sprites[ARROW_INDEX].hit = 1;
	// sprites[ARROW_INDEX].index = ARROW_INDEX;
}

void scorecombosetup(sprite *sprites) {
	//index 0 acts strangly
	//'SCORE'
	sprites[1].id = 17; //S
	sprites[2].id = 12; //C
	sprites[3].id = 15; //O
	sprites[4].id = 16; //R
	sprites[5].id = 13; //E
	for (int i = 1; i < 6; i++) {
		sprites[i].x = 480+32*(i-1);
		sprites[i].y = 40;
		sprites[i].dx = 0;
		sprites[i].dy = 0;
		sprites[i].hit = 1;
		sprites[i].index = i;
	}
	sprites[6].id = 10; //0
	sprites[7].id = 10; //0
	sprites[8].id = 10; //0
	for (int i = 6; i < 9; i++) {
		sprites[i].x = 480+32+32*(i-6);
		sprites[i].y = 90;
		sprites[i].dx = 0;
		sprites[i].dy = 0;
		sprites[i].hit = 1;
		sprites[i].index = i;
	}


	//'MAX'
	sprites[9].id = 14; //M
	sprites[10].id = 26; //A
	sprites[11].id = 27; //X
	for (int i = 9; i < 12; i++) {
		sprites[i].x = 480+32+32*(i-9);
		sprites[i].y = 240;
		sprites[i].dx = 0;
		sprites[i].dy = 0;
		sprites[i].hit = 1;
		sprites[i].index = i;
	}
	int max = read_max_from_file();
	if(max>0){
		update_max(sprites, max);
	}
	else{
	sprites[12].id = 10; //0
	sprites[13].id = 10; //0
	sprites[14].id = 10; //0
```

```
	}
	for (int i = 12; i < 15; i++) {
		sprites[i].x = 480+32+32*(i-12);
		sprites[i].y = 290;
		sprites[i].dx = 0;
		sprites[i].dy = 0;
		sprites[i].hit = 1;
		sprites[i].index = i;
	}
	sprites[15].id = BIRD_ID;
	sprites[15].x = 224;
	sprites[15].y = 240;
	sprites[15].dx = 0;
	sprites[15].dy = 0;
	sprites[15].hit = 1;
	sprites[15].index = 15;
}

// void create_bird(sprite *sprites){
//     sprites[15].id = BIRD_ID;
//     sprites[15].x = 240;
//     sprites[15].y = 240;
//     sprites[15].dx = 0;
//     sprites[15].dy = 0;
//     sprites[15].hit = 1;
//     sprites[15].index = 15;
// }



void setting_setup(sprite *sprites){
	// we can change the index starting from 42
	// SETTING
		// SETTING
	sprites[1].id = 22; //S
	sprites[2].id = 18; //E
	sprites[3].id = 23; //T
	sprites[4].id = 23; //T
	sprites[5].id = 24; //I
	sprites[6].id = 25; //N
	sprites[7].id = 31; //G

	for (int i = 1; i < 8; i++) {
		sprites[i].x = 108 + 32*(i-1);
		sprites[i].y = 120;
		sprites[i].dx = 0;
		sprites[i].dy = 0;
		sprites[i].hit = 1;
		sprites[i].index = i;
	}

	// EASY
	sprites[8].id = 18;  //E
	sprites[9].id = 33;  //A
```

```
	sprites[10].id = 22; //S
	sprites[11].id = 35; //Y No this one

	for (int i = 8; i < 12; i++) {
		sprites[i].x = 140 + 32*(i-8);
		sprites[i].y = 180;
		sprites[i].dx = 0;
		sprites[i].dy = 0;
		sprites[i].hit = 1;
		sprites[i].index = i;
	}

	//MEDIUM
	sprites[12].id = 20; //M
	sprites[13].id = 18; //E
	sprites[14].id = 36; //D
	sprites[15].id = 24; //I
	sprites[16].id = 19; //U
	sprites[17].id = 20; //M

	for (int i = 12; i < 18; i++) {
		sprites[i].x = 140 + 32*(i-12);
		sprites[i].y = 240;
		sprites[i].dx = 0;
		sprites[i].dy = 0;
		sprites[i].hit = 1;
		sprites[i].index = i;
	}

	//HARD
	sprites[18].id = 37; //H
	sprites[19].id = 33; //A
	sprites[20].id = 34; //R
	sprites[21].id = 36; //D

	for (int i = 18; i < 22; i++) {
		sprites[i].x = 140 + 32*(i-18);
		sprites[i].y = 300;
		sprites[i].dx = 0;
		sprites[i].dy = 0;
		sprites[i].hit = 1;
		sprites[i].index = i;
	}

	// sprites[ARROW_INDEX].id = 28; // Arrow
	// sprites[ARROW_INDEX].x = 96;
	// sprites[ARROW_INDEX].y = 180;
	// sprites[ARROW_INDEX].dx = 0;
	// sprites[ARROW_INDEX].dy = 0;
	// sprites[ARROW_INDEX].hit = 1;
	// sprites[ARROW_INDEX].index = ARROW_INDEX;

}
```

```
void gameoversetup(sprite *sprites){
	//GAMEOVER
	sprites[1].id = 31; //G
	sprites[2].id = 33; //A
	sprites[3].id = 20; //M
	sprites[4].id = 18; //E
	sprites[5].id = 38; //O
	sprites[6].id = 21; //V
	sprites[7].id = 18; //E
	sprites[8].id = 34; //R
	for (int i = 1; i < 9; i++) {
		sprites[i].x = 120+32*(i-1);
		sprites[i].y = 120;
		sprites[i].dx = 0;
		sprites[i].dy = 0;
		sprites[i].hit = 1;
		sprites[i].index = i;
	}

	sprites[9].id = 32; //dead bird
	sprites[9].x = 224;
	sprites[9].y = 240;
	sprites[9].dx = 0;
	sprites[9].dy = 0;
	sprites[9].hit = 1;
	sprites[9].index = 9;
}

void update_score(sprite *sprites, const int score) {
	int huds = (int)score/100;
	int tens = (int)(score - huds*100)/10;
	int ones = score - huds*100 - tens*10;
	if (huds == 0) huds = 10;
	if (tens == 0) tens = 10;
	if (ones == 0) ones = 10;
	sprites[6].id = huds; //100s
	sprites[7].id = tens; //10s
	sprites[8].id = ones; //1s
	return;
}

void update_max(sprite *sprites, const int max) {
	int huds = (int)max/100;
	int tens = (int)(max - huds*100)/10;
	int ones = max - huds*100 - tens*10;
	if (huds == 0) huds = 10;
	if (tens == 0) tens = 10;
	if (ones == 0) ones = 10;
	sprites[12].id = huds; //100s
	sprites[13].id = tens; //10s
	sprites[14].id = ones; //1s
	return;
}
// dedicate all sprites below
```

```
void screen_refresh(sprite* sprites) {
   for (int i = 1; i < SIZE; i++) {
              sprites[i].x = 630;
              sprites[i].y = 470;
              sprites[i].dx = 0;
              sprites[i].dy = 0;
              sprites[i].id = 0;
              sprites[i].hit = 1;
              sprites[i].index = i;
       }
       return;
}

int check_receive_audio(int counter, float* sum_audio_data, int aud_fd, aud_mem_t*
amt) {
       amt->data = get_aud_data(aud_fd);
       //printf("amt=%d\n",amt->data);
       if ((counter%10)==0) {
              if (*sum_audio_data / 10 > 50000000){
                     printf("after ten counter, with hit, the mean is: %f\n",
(*sum_audio_data/10));
                     *sum_audio_data = 0;
                     return 1;
              }
              //printf("after ten counter, without hit, the mean is: %f\n",
(*sum_audio_data/10));
              *sum_audio_data = 0;
              return 0;
       } else{
              *sum_audio_data += abs(amt->data);
              return 0;
       }
       return 0;
}

int return_operation(int aud_fd) {
       int button_value = get_button_value(aud_fd);
       // value it 15
       if (button_value == NEXT){
              return NEXT;
       } else if (button_value == CONFIRM){
              return CONFIRM;
       } else {
              return 0;
       }
       return 0;
}

int get_arrow_position_y(sprite *sprites){
       return sprites[ARROW_INDEX].y;
}
```

```c
void add_arrow_position_y(sprite *sprites, int y){
	sprites[ARROW_INDEX].y += y;
	//printf("Arrow pos is: %d\n", sprites[ARROW_INDEX].y );
}

void send_to_hardware(sprite *sprites, int vga_bird_fd, vga_bird_data_t *vzdt) {
	for (int i = 0; i < SIZE; i++) {
		vzdt->data[i] = (sprites[i].index<<26) + (sprites[i].id<<20) +
(sprites[i].y<<10) + (sprites[i].x<<0);
	}
	send_sprite_positions(&vzdt, vga_bird_fd);
}

int set_difficulty_level(sprite *sprites) {
	if (get_arrow_position_y(sprites) == 180) {

		return EASY;
	} else if (get_arrow_position_y(sprites) == 240) {
		return MEDIUM;
	} else if (get_arrow_position_y(sprites) == 300) {
		return HARD;
	}
	return EASY;
}
int read_max_from_file(enum difficulty difficulty) {
    int max = 0;
    FILE *file = NULL;

    // Determine the file name based on the difficulty level
    switch (difficulty) {
        case EASY:
            file = fopen("max_score_easy.txt", "r");
            break;
        case MEDIUM:
            file = fopen("max_score_medium.txt", "r");
            break;
        case HARD:
            file = fopen("max_score_hard.txt", "r");
            break;
        default:
            return -1;  // Return error if difficulty is not recognized
    }

    // Check if the file was opened successfully
    if (file == NULL) {
        return -1;  // Return -1 or another error code if file can't be opened
    }

    // Read the max score from the file
    fscanf(file, "%d", &max);
    fclose(file);  // Close the file
    return max;
}
```

```
void save_max_to_file(int max, enum difficulty difficulty) {
       FILE *file = NULL;
    // Select the appropriate file based on the difficulty level
    switch (difficulty) {
        case EASY:
            file = fopen("max_score_easy.txt", "w");
            break;
        case MEDIUM:
            file = fopen("max_score_medium.txt", "w");
            break;
        case HARD:
            file = fopen("max_score_hard.txt", "w");
            break;
        default:
            printf("Invalid difficulty level!\n");
            return;
    }

    // Check if the file was opened successfully
    if (file == NULL) {
        printf("Error opening file!\n");
        return;
    }

    // Write the max score to the file
    fprintf(file, "%d", max);
    fclose(file);  // Close the file
}

// simple game of hitting random falling notes when they reach the green zone
int main()
{
       log_file = fopen("log.txt", "w");
       if (log_file == NULL) {
         printf("Failed to open the file.\n");
         return 1;
    }
       vga_bird_arg_t vzat;

       aud_arg_t aat;
       aud_mem_t amt;

       vga_pipe_position_t pipe_info_first, pipe_info_second;

       int signal;
       int pipe_num;

       float sum_audio_data;

       srand(time(NULL));

       static const char filename1[] = "/dev/vga_bird";
       static const char filename2[] = "/dev/aud";
```

```
	printf("VGA Bird test Userspace program started\n");
	printf("%d\n", sizeof(int));
	printf("%d\n", sizeof(short));

	if ((vga_bird_fd = open(filename1, O_RDWR)) == -1) {
		fprintf(stderr, "could not open %s\n", filename1);
		return -1;
	}
	if ((aud_fd = open(filename2, O_RDWR)) == -1) {
		fprintf(stderr, "could not open %s\n", filename2);
		return -1;
	}
	//FILE *fp = fopen("test.txt", "w");
	//if (fp == NULL)   return -1;
	vga_bird_data_t vzdt;
	sprite *sprites = NULL;
	sprites = calloc(SIZE, sizeof(*sprites));
	int max = 0;
	int arr_y=60;
	enum difficulty difficulty_level = EASY;
	int arrow_created;
program_start:
	// Just for testing.
	//  for (;;){
	//    screen_refresh(sprites);
	//    setting_setup(sprites);

	//    if (return_operation(aud_fd) == CONFIRM){
	//          break;
	//    }

	//    for (int i = 0; i < SIZE; i++) {
	//          vzdt.data[i] = (sprites[i].index<<26) + (sprites[i].id<<20) +
(sprites[i].y<<10) + (sprites[i].x<<0);
	//    }
	//    //send package to hardware
	//    send_sprite_positions(&vzdt, vga_bird_fd);
	//  }
	arrow_created = 0;
	screen_refresh(sprites);
	menu_setup(sprites);
	for (;;){
		if (!arrow_created){
			sprites[ARROW_INDEX].id = 28; // Arrow is bird now
			sprites[ARROW_INDEX].x = 370;
			sprites[ARROW_INDEX].y = 180;
			sprites[ARROW_INDEX].dx = 0;
			sprites[ARROW_INDEX].dy = 0;
			sprites[ARROW_INDEX].hit = 1;
			sprites[ARROW_INDEX].index = ARROW_INDEX;
		}
		arrow_created = 1;
		if (return_operation(aud_fd) == CONFIRM &&
get_arrow_position_y(sprites) == 180) {
```

```
                    printf("game start.\n");
                    break;
             } else if (return_operation(aud_fd) == CONFIRM &&
get_arrow_position_y(sprites) == 240) {
                    printf("setting start.\n");
                    screen_refresh(sprites);
                    setting_setup(sprites);
                    int setting_arrow_created;
                    setting_arrow_created = 0;
                    for (;;){
                          usleep(200000);
                          if (!setting_arrow_created){
                                sprites[ARROW_INDEX].id = 28; // Arrow is bird now
                                sprites[ARROW_INDEX].x = 370;
                                sprites[ARROW_INDEX].y = 180;
                                sprites[ARROW_INDEX].dx = 0;
                                sprites[ARROW_INDEX].dy = 0;
                                sprites[ARROW_INDEX].hit = 1;
                                sprites[ARROW_INDEX].index = ARROW_INDEX;
                          }
                          setting_arrow_created = 1;
                          if (return_operation(aud_fd) == CONFIRM) {
                                difficulty_level = set_difficulty_level(sprites);
                                screen_refresh(sprites);
                                menu_setup(sprites);
                                arrow_created = 0;
                                break;
                          } else if (return_operation(aud_fd) == NEXT) {
                                add_arrow_position_y(sprites, 60);
                                if (get_arrow_position_y(sprites) == 360) {
                                      add_arrow_position_y(sprites, -180);
                                }
                          }
                          for (int i = 0; i < SIZE; i++) {
                                vzdt.data[i] = (sprites[i].index<<26) +
(sprites[i].id<<20) + (sprites[i].y<<10) + (sprites[i].x<<0);
                          }
                          //send package to hardware
                          send_sprite_positions(&vzdt, vga_bird_fd);
                    }
             } else if (return_operation(aud_fd) == NEXT) {
                    printf("next.\n");
                    add_arrow_position_y(sprites, 60);
                    if (get_arrow_position_y(sprites) == 300) {
                          add_arrow_position_y(sprites, -120);
                    }
             }
             usleep(200000);
             for (int i = 0; i < SIZE; i++) {
                    vzdt.data[i] = (sprites[i].index<<26) + (sprites[i].id<<20) +
(sprites[i].y<<10) + (sprites[i].x<<0);
             }
             //send package to hardware
             send_sprite_positions(&vzdt, vga_bird_fd);
```

```
        }
       sum_audio_data = 0;
       screen_refresh(sprites);
       scorecombosetup(sprites);
       create_pipe(sprites, &pipe_info_first, 16, difficulty_level);

       //initialize pipe info second.
       pipe_info_second.pipe_num = 0;
       pipe_info_second.top_pipe_max_index = 0;
       pipe_info_second.bottom_pipe_max_index = 0;

       int score = 0;
       int combo = 0;
       //packet of sprite data to send
       int combo_flag = 1;
       int counter = 0;
       int gamecounter = 0;
    int validleft, validright;
    int hitcount = 0;
       int noteCount = 0;
       int MAX_NOTE_COUNT = 100;
       for (;;) {
       //printf("amt = %f\n", amt.data);
              //printf("game difficulty level is: %d\n", difficulty_level);
              if (check_receive_audio(counter, &sum_audio_data, aud_fd, &amt)) {
                     for (int j = 0; j < 20; j++){
                            sprites[15].dy = -1;
                            updateBall(&sprites[15]);

                            vzdt.data[15] = (sprites[15].index<<26) +
(sprites[15].id<<20) + (sprites[15].y<<10) + (sprites[15].x<<0);

                            //send package to hardware
                            send_sprite_positions(&vzdt, vga_bird_fd);
                     }
                     aud_mem_t c;
                     c.sound = 1;
                     printf("send bird flap sound\n");
                     send_sound(&c, aud_fd);
              } else{
                     sprites[15].dy = 1;
                     aud_mem_t c;
                     c.sound = 0;
                     send_sound(&c, aud_fd);
              }

              //update_score(sprites, amt.data);
              //update_combo(sprites, 1+(sprites[validleft].id-17)>>1);
              //update_score(sprites, score);
              //update_combo(sprites, combo);
              //update_max(sprites, max);

              //package the sprites together
              create_pipe_condition(sprites, &pipe_info_first, &pipe_info_second,
```

```
difficulty_level);

            int bird_dead;
            bird_dead = check_bird_position(sprites, &pipe_info_first,
&pipe_info_second);
            if (bird_dead) {
                  printf("bird dead.\n");
                  //reset the bird position.
                  screen_refresh(sprites);
                  gameoversetup(sprites);

                  for (int i = 0; i < SIZE; i++) {
                  vzdt.data[i] = (sprites[i].index<<26) + (sprites[i].id<<20) +
(sprites[i].y<<10) + (sprites[i].x<<0);
                  }
                  //send package to hardware
                  send_sprite_positions(&vzdt, vga_bird_fd);
                  usleep(3000000);
                  aud_mem_t c;
                  c.sound = 0;
                  printf("send sound\n");
                  send_sound(&c, aud_fd);
                  goto program_start;
            }
            int passed=check_bird_pass_pipe(sprites, &pipe_info_first,
&pipe_info_second);
            if(passed){
                  score += passed;
            }
            max=read_max_from_file(difficulty_level);
            if(score > max){
                  max = score;
                  save_max_to_file(max,difficulty_level);
            }
            update_score(sprites, score);
            update_max(sprites, max);
            for (int i = 0; i < SIZE; i++) {
                  vzdt.data[i] = (sprites[i].index<<26) + (sprites[i].id<<20) +
(sprites[i].y<<10) + (sprites[i].x<<0);
            }
            //send package to hardware
            send_sprite_positions(&vzdt, vga_bird_fd);

            updateBall(&sprites[15]);
            //update the pipe position;
            for (int i = 16; i < 42; i++) {
                  updateBall(&sprites[i]);
            }
            combo_flag = 1;
            //pause to let hardware catch up
            counter++;
            //signal = 0;
            // aud_mem_t c;
            // c.sound = 0;
```

```
            // printf("send sound\n");
            // send_sound(&c, aud_fd);
            usleep(20000);
        }
        free (sprites);

        return 0;
}
```