Final Report for Atari Breakout: Candy Edition

Jason Eriksen (jce2148)
Xurxo Riesco (xr2154)

Spring 2023

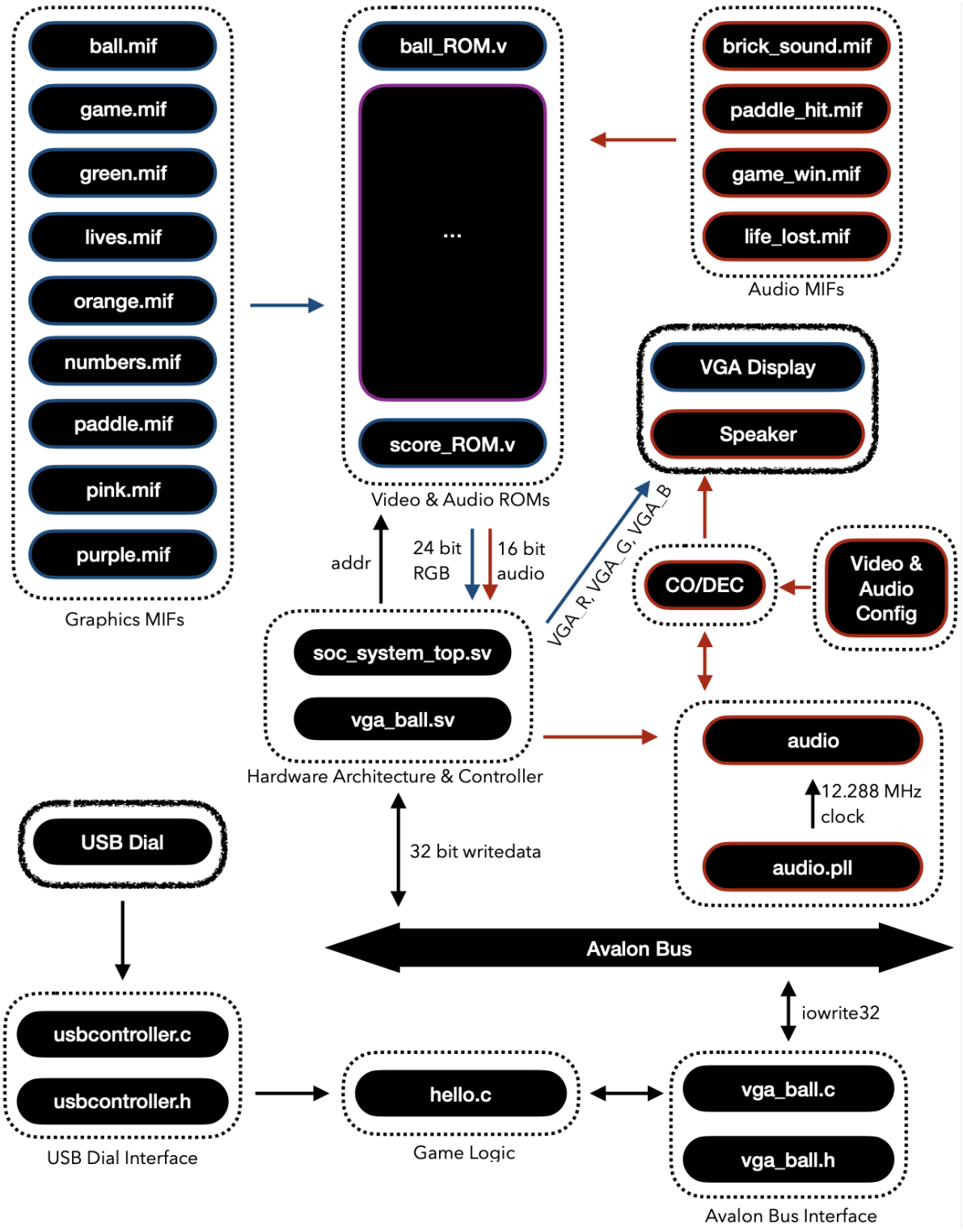
Table of Contents

1 Introduction

For our version of Atari Breakout, we implemented a variety of components. Several components are modified versions of those completed in the prior labs (i.e. vga_ball), but we also developed several completely new components, like audio and an interface via volume controller instead of keyboard. On the software side, we utilized a driver similar to that of Lab 3, but with updated logic to reflect the rules of Atari Breakout, such as the shattering of bricks, bounces on the paddle, and the changes in speed of the ball. Also, we included separate display modes for menu, victory, and defeat screens. Additionally, we developed the aforementioned user input driver that maps joystick movements to in game actions. This is all communicated to the hardware via the avalon bus. The hardware includes the video and audio logic as well as the associated Read Only Memory (ROM) for displaying the ball, paddle, bricks, and sound

effects.

2 System Block Diagram



The audio data path is highlighted in red and the video data path is highlighted in blue. The peripherals are denoted by the chalk-like outline.

3 Design

3.1 Software Design

3.1.1 Control Logic

To move the paddle, we will use a volume knob as the controller. The knob is connected to the board via USB. The functionality is very limited, only offering “turn right” and “turn left.”

The knob uses a very simple protocol, sending 2 byte packages, with the first byte being fixed at value 2 and the second byte representing the direction; value 234 for a left turn and value 233 for a right turn. In between each of the information packages, the knob sends a package where the second byte is set to 0. For the purposes of the game, that empty package just gets ignored, and we are only concerned with using the left and right turns as left and right movement for the paddle.

3.1.2 Game rules

The paddle is only allowed to be moved along the horizontal axis. Along the vertical axis, the rules are simple: if the top part of the ball touches the bottom part of a brick, the brick is destroyed, if the bottom part of the ball goes below the vertical position of the paddle while not making contact with it, the player loses a life. If the player is able to remove all the bricks before running out of lifes, the player wins the game. In contrast, if the player runs out of lives while there is one or more bricks on the screen, the player loses the game.

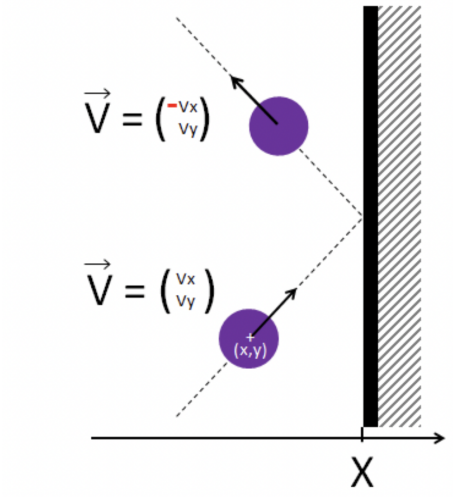
3.1.2 Bouncing Ball

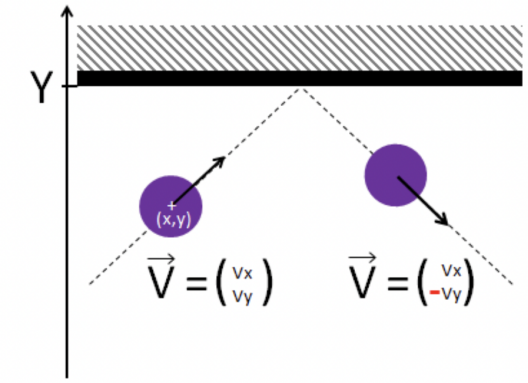
The movement of the ball is determined by the position of the ball along both axes, (x,y), and the velocity of the ball (v_x, v_y) in the following manner.

Position at t_0: (x, y)

Position at t_1: ($x+v_x$, $y+v_y$)

When the ball contacts the paddle or the bricks as defined in the Game Rules (3.1.1), the v_y component of the velocity is reversed via negation. Similarly, if the leftmost or rightmost point of the ball makes contact with the left or right wall, the v_x component of the velocity is reversed via negation. The contact with the right wall and brick are portrayed below, with contact with the left wall and the paddle being direct opposites.

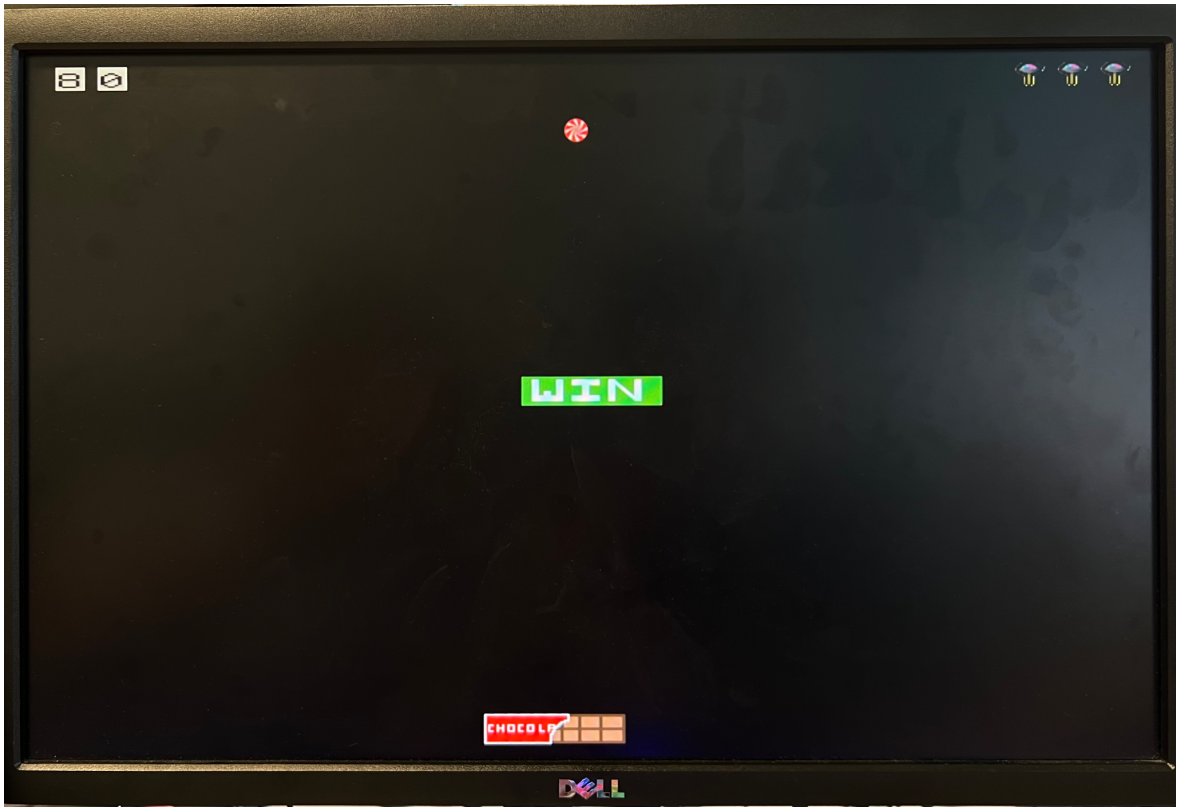


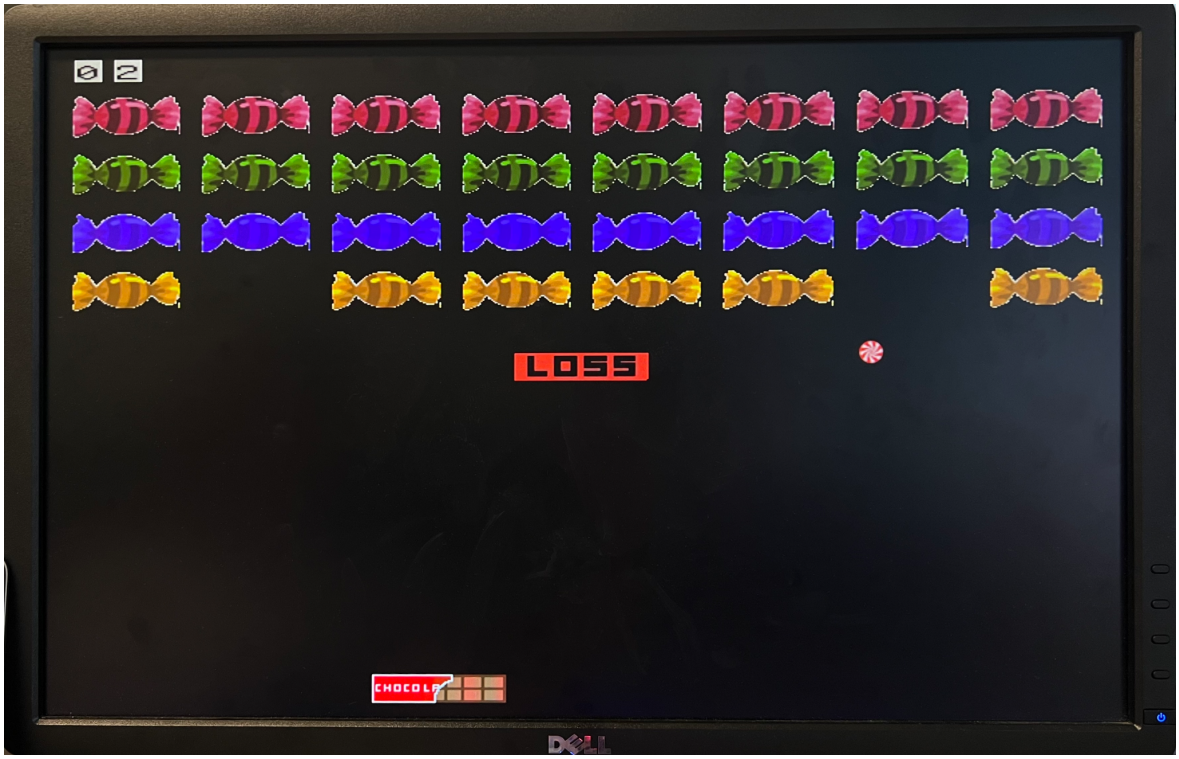


3.1 Hardware Design

3.1.1 Graphics

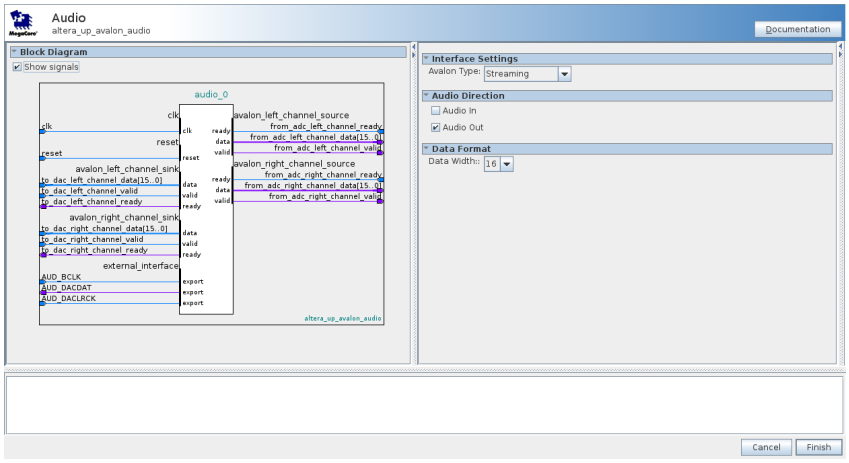
The hardware receives the center position of the ball as an (x,y) coordinate pair and center position of the paddle as the x coordinate, given that the y coordinate is fixed, with the software being responsible for all the calculations. In addition, the software receives a bitfield containing the state of the blocks per row and is then responsible for displaying said blocks in the screen, each with a fixed position, giving each of the blocks in a given row a particular design. The design for the blocks, the ball, and the paddle will be stored as sprites in the memory of the FPGA as well as the necessary sprites for game information, such as the lives and score. The hardware is then responsible for displaying the correct graphic based on the game logic outputs computed by software. As with the position of the ball and the paddle, the lives and scores are calculated from the software and the hardware is merely responsible for displaying them. The HW-SW interface is designed in such a way that facilitates the hardware task and leaves most of the calculations to the software. Further details can be found in that section. The win and loss logic is done in Hardware, if the player has more than one life and it has reached a score of 80, we consider it a win, and if the player has no lives and the score is less than 80, it is considered a loss, different banners are displayed in both of these scenarios.


80
WIN

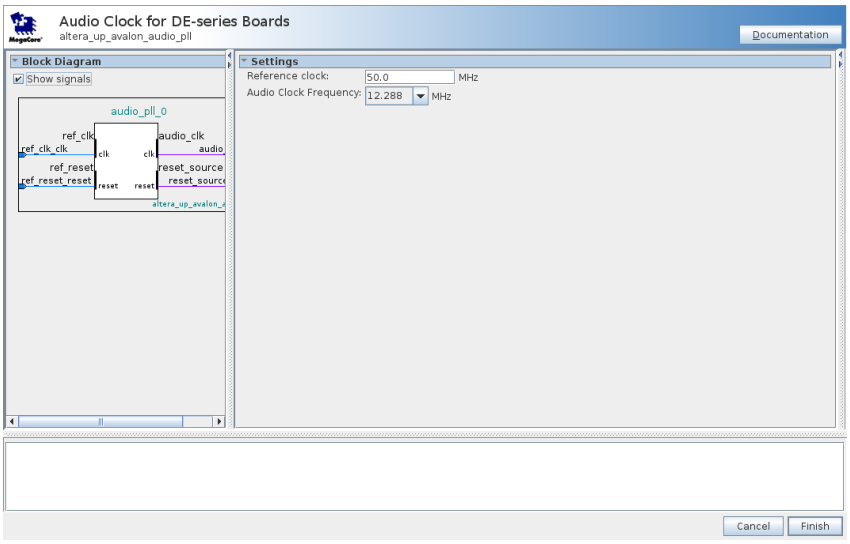

02
LOSS

3.1.2 Audio

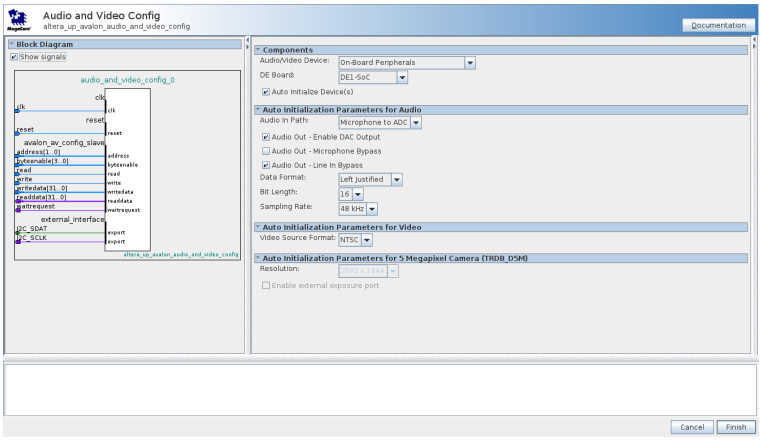
The game supports four different sounds that needed to be pre-processed prior to their storage in memory. All audio clips were sub-sampled at 8KHz, and converted to 16-bit mono audio. Then transformed into .mif files, each associated to a different one-port memory module. To actually play the audio stored in memory, several IPs were added to the board, and the main control module was then connected to the audio IPs in the following manner:



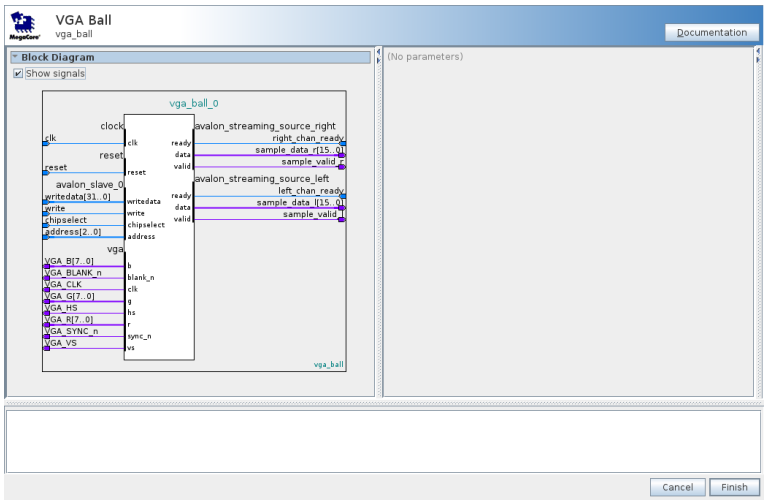
Audio IP Core



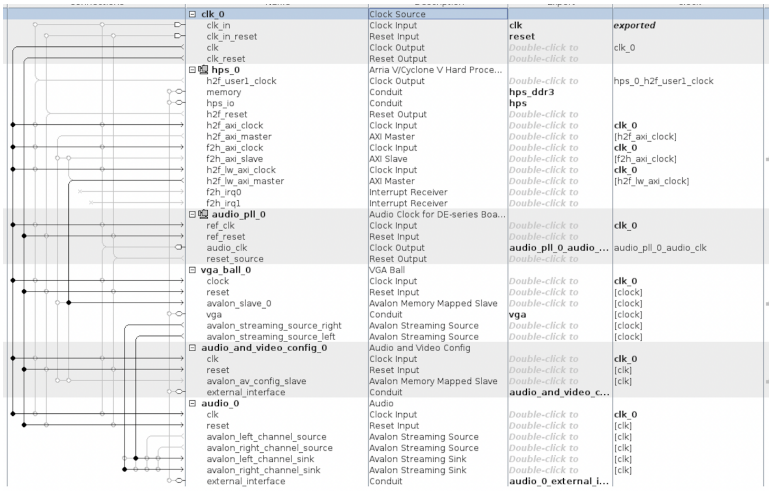
Audio Clock IP Core



Audio & Video Configuration IP Core

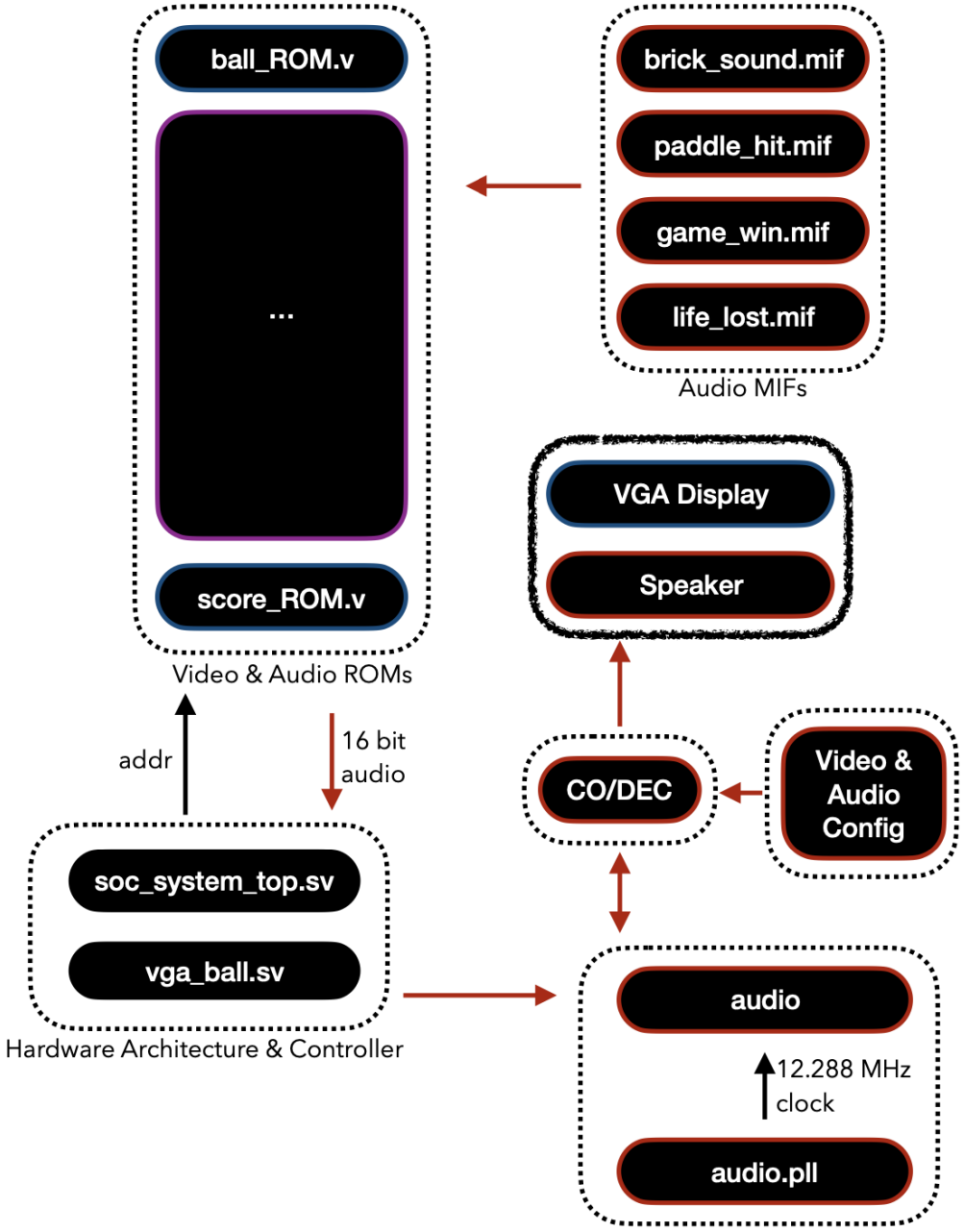


Main Module



Connections

The audio clock is set to 12.288 MHz, so the audio logic within the control module requires a frequency divider to properly operate. The audio module is configured to be left-justified and uses a 16 bit data-width to match the format of the pre-processed audio samples. The audio modules operate together and result in the main module receiving two ready signals, one for the left channel and one for the right channel, and to which it can respond with two signals for the data and two valid signals. The resulting architecture can be best summarized by the figure below, highlighted by the red path and outlines:



4 Resource Budget

4.1 Video

Image	Dimensions	# of Sprites	Size
Blocks	64 x 32	4	8,192
Ball	16 x 16	1	256
Paddle	80 x 20	1	1,600
Lives	16 x 16	1	256
Win/Loss	80 x 20	3	3,200
Score	16x16	10	2,560
		Total:	16,064

	Paddle Hit	Brick Hit	Life Lost	Game Win
time(s)	1.16	0.04	4.51	4.19
f_s(kHz)	8	8	8	8
Size	9,241	290	36,052	33,539
			Total:	79,122

4.3 Total bits

For the images, we need 24 bits of data, and for the audios we need 16 bits of data.

Therefore, 16,064 x 24 + 79,122 x 16 = 1,651,488 bits of total memory budget.

5 The Hardware/Software Interface

The total dimensions of the screen are 640 x 480. Therefore, 19 bits are needed to encode any given position. 10 bits for x component ($640 < 2^{10}$) and 9 bits for the y component ($480 < 2^{9}$). Score takes any value 0 to 80, so it could be represented in 7 bits, ($80 < 2^{7}$); however, since the digits are sent separately, we need 8 bits ($9 < 2^{4}$).

Register 1: Positions

1. Ball Position X (bx) - 10 bits
2. Ball Position Y (by) - 9 bits
3. Paddle Position X (px) - 10 bits

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
bx	**bx**	**bx**	**bx**	**bx**	**bx**	**bx**	**bx**	**bx**	**bx**	**by**	**by**	**by**	**by**	**by**	**by**
16	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*	*31*
by	**by**	**by**	**by**	**px**	**px**	**px**	**px**	**px**	**px**	**px**	**px**	**px**	**px**	**-**	**-**

Register 2: Bricks

1. Layer 1 Brick Status (b1) - 8 bits
2. Layer 2 Brick Status (b2) - 8 bits
3. Layer 3 Brick Status (b3) - 8 bits
4. Layer 4 Brick Status (b4) - 8 bits

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
b1	**b1**	**b1**	**b1**	**b1**	**b1**	**b1**	**b1**	**b2**	**b2**	**b2**	**b2**	**b2**	**b2**	**b2**	**b2**
16	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*	*31*
b3	**b3**	**b3**	**b3**	**b3**	**b3**	**b3**	**b3**	**b4**	**b4**	**b4**	**b4**	**b4**	**b4**	**b4**	**b4**

Register 3: Bricks

1. Lives, one-hot encoded (l) - 3 bits
2. Audio, one-hot encoded (a) - 4 bits
3. Score, upper digit (s1) - 4 bits
4. Score, upper digit (s2) - 4 bits
5. Start status (ss) - 1 bit

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
l	**l**	**l**	**a**	**a**	**a**	**a**	**s1**	**s1**	**s1**	**s1**	**s2**	**s2**	**s2**	**s2**	**ss**
16	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*	*31*
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

6 Division of Work and Learnings

Given our reduced team size, we were able to meet for all the development stages of the project, and worked collaboratively via pair programming. One of the struggles was the memory budget, despite leaving a reasonable amount of empty space during our planning, we still reached the memory budget while developing the audio, and had to update our design to use one less audio effect. To compensate for this, we decided to remove the “game lost” audio and let the “life lost” audio play even when the user goes from one to zero lives. Another struggle was getting the speakers to play the sounds, since there were many parts of the design involved, it was a non-trivial debugging process, which we overcame by adding visual clues to help test each aspect individually. In terms of advice for future groups, we would recommend sticking to smaller grouper sizes as this ensures easier collaboration and coordination. Additionally, this helps to minimize loss of understanding, as smaller groups force collaboration on greater portions of the project, rather than dividing up the work. Understanding of the complete project is vital even when working on its subparts.

7 References

1. https://www.101computing.net/bouncing-algorithm/
2. http://www.cs.columbia.edu/~sedwards/classes/2022/4840-spring/designs/Breakout.pdf
3. http://www.cs.columbia.edu/~sedwards/classes/2019/4840-spring/designs/BrickBreaker.pdf

8 Code Listings

8.1 Hardware

*** vga_ball.sv ***

```
/*
* Avalon memory-mapped peripheral that generates VGA
*
* Stephen A. Edwards
* Columbia University
*/

module vga_ball(
  input logic          clk,
  input logic          reset,
  input logic [31:0]   writedata,
  input logic          write,
  input                chipselect,
  input logic [2:0]    address,
  input logic          left_chan_ready,
  input logic          right_chan_ready,

  output logic [15:0]  sample_data_l,
  output logic [15:0]  sample_data_r,
  output logic         sample_valid_l,
  output logic         sample_valid_r,
  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
);

  logic [31:0]         blocks, next_blocks;

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

  logic [9:0]          paddle_pos, next_paddle_pos;
  logic [9:0]          ball_hpos, next_ball_hpos;
  logic [8:0]          ball_vpos, next_ball_vpos;

  logic [7:0]          background_r, background_g, background_b;

  logic [7:0]          score, next_score;
  logic [3:0]          audio, next_audio;
  logic [2:0]          lives, next_lives;

  logic                start, next_start;

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

  always_comb begin
     next_ball_hpos = ball_hpos;
     next_ball_vpos = ball_vpos;
     next_paddle_pos = paddle_pos;
     next_lives = lives;
     next_blocks = blocks;
     next_audio = audio;
     next_score = score;
     next_start = start;
     case (address)
        3'h1: begin
```

```
         next_ball_hpos = writedata[9:0];
         next_ball_vpos = writedata[18:10];
         next_paddle_pos = writedata[28:19];
      end
      3'h2: begin
         next_blocks = writedata;
      end
      3'h3: begin
         next_lives = writedata[2:0];
         next_audio = writedata[6:3];
         next_score = writedata[14:7];
         next_start = writedata[15];
      end
   endcase
end


always_ff @(posedge clk) begin
   if (reset) begin
      background_r <= 8'h0;
      background_g <= 8'h0;
      background_b <= 8'h80;
      blocks <= 32'b0;
      ball_hpos <= 10'd300;
      ball_vpos <= 9'd220;
      paddle_pos <= 10'd300;
      score <= 8'b0;
      audio <= 5'b0;
      lives <= 3'b111;
   end else if (chipselect && write)
      case (address)
         3'h0 : begin
            background_r <= writedata[7:0];
            background_g <= writedata[15:8];
            background_b <= writedata[23:16];
         end
      endcase
   ball_hpos <= next_ball_hpos;
   ball_vpos <= next_ball_vpos;
   paddle_pos <= next_paddle_pos;
   blocks <= next_blocks;
   lives <= next_lives;
   audio <= next_audio;
   score <= next_score;
   start <= next_start;
end



// ============ AUDIO ============ //

logic [13:0] counter;

// Brick Hit
logic [8:0]  bhs_addr;
logic [15:0] bhs_output;
logic bhs_en;
brick_sound_ROM bhs(.address(bhs_addr), .clock(clk), .q(bhs_output));

// Paddle Hit
logic [13:0] phs_addr;
logic [15:0] phs_output;
logic phs_en;
```

```
coin_sound_ROM phs(.address(phs_addr), .clock(clk), .q(phs_output));

// Game Win
logic [15:0] gws_addr;
logic [15:0] gws_output;
logic gws_en;
game_win_ROM gws(.address(gws_addr), .clock(clk), .q(gws_output));

// Life Lost
logic [15:0] lls_addr;
logic [15:0] lls_output;
logic lls_en;
life_lost_ROM lls(.address(lls_addr), .clock(clk), .q(lls_output));


// Sound Select Logic
always_ff @(posedge clk) begin
   if (reset) begin
      counter <= 0;
      sample_valid_l <= 0;
      sample_valid_r <= 0;
   end
   else if(left_chan_ready == 1 && right_chan_ready == 1 && counter < 6250) begin
      counter <= counter + 1;
      sample_valid_l <= 0;
      sample_valid_r <= 0;
   end
   else if(left_chan_ready == 1 && right_chan_ready == 1 && counter == 6250) begin
      counter <= 0;
      sample_valid_l <= 1;
      sample_valid_r <= 1;
      if (audio[0] || bhs_en == 1'b0) begin
         if (bhs_addr < 9'd290) begin
            bhs_addr <= bhs_addr+1;
            bhs_en <= 1'b0;
         end
         else begin
            bhs_addr <= 0;
            bhs_en <= 1'b1;
         end
         sample_data_l <= bhs_output;
         sample_data_r <= bhs_output;
      end else if (audio[1] || phs_en == 1'b0) begin
         if (phs_addr < 14'd9241) begin
            phs_addr <= phs_addr+1;
            phs_en <= 1'b0;
         end
         else begin
            phs_addr <= 0;
            phs_en <= 1'b1;
         end
         sample_data_l <= phs_output;
         sample_data_r <= phs_output;
      end else if (audio[2] || gws_en == 1'b0) begin
         if (gws_addr < 16'd33539) begin
            gws_addr <= gws_addr+1;
            gws_en <= 1'b0;
         end
         else begin
            gws_addr <= 0;
            gws_en <= 1'b1;
         end
         sample_data_l <= gws_output;
```

```
         sample_data_r <= gws_output;
      end else if (audio[3] || lls_en == 1'b0) begin
         if (lls_addr < 16'd36052) begin
            lls_addr <= lls_addr+1;
            lls_en <= 1'b0;
         end
         else begin
            lls_addr <= 0;
            lls_en <= 1'b1;
         end
         sample_data_l <= lls_output;
         sample_data_r <= lls_output;
      end
      else begin
         sample_data_l <= 0;
         sample_data_r <= 0;
      end
   end else begin
      sample_valid_l <= 0;
      sample_valid_r <= 0;
   end
end



// ============ VIDEO ============ //

// Ball Logic
logic [7:0] ball_addr;
logic [23:0] ball_output;
logic ball_en;
ball_ROM ball(.address(ball_addr),.clock(clk),.q(ball_output));
always_ff @(posedge clk) begin
   if (hcount[10:1] >= ball_hpos-8 && hcount[10:1] < ball_hpos+8 && vcount[9:0] >= ball_vpos-8 && vcount[9:0]<ball_vpos+8)
begin
      ball_en <= 1;
      ball_addr <= (hcount[10:1] - (ball_hpos-8)) + (vcount[9:0] - (ball_vpos-8)) * 16;
   end else begin
      ball_en <= 0;
      ball_addr <= 0;
   end
end

// Lives Logic
logic [7:0] lives_addr;
logic [23:0] lives_output;
logic lives_en;
lives_ROM life(.address(lives_addr),.clock(clk),.q(lives_output));
always_ff @(posedge clk) begin
   if (hcount[10:1] >= 608 && hcount[10:1] <= 624 && vcount[9:0] >= 8 && vcount[9:0] < 24 && lives[0]) begin
      lives_en <= 1;
      lives_addr <= (hcount[10:1] - 608) + (vcount[9:0] - 8) * 16;
   end else if (hcount[10:1] >= 584 && hcount[10:1] <= 600 && vcount[9:0] >= 8 && vcount[9:0] < 24 && lives[1]) begin
      lives_en <= 1;
      lives_addr <= (hcount[10:1] - 584) + (vcount[9:0] - 8) * 16;
   end else if (hcount[10:1] >= 560 && hcount[10:1] <= 576 && vcount[9:0] >= 8 && vcount[9:0] < 24 && lives[2]) begin
      lives_en <= 1;
      lives_addr <= (hcount[10:1] - 560) + (vcount[9:0] - 8) * 16;
   end else begin
      lives_en <= 0;
      lives_addr <= 0;
   end
end
```

```
// Game Logic
logic [11:0] game_addr;
logic [23:0] game_output;
logic game_en;
game_ROM game(.address(game_addr),.clock(clk),.q(game_output));
always_ff @(posedge clk) begin
   if (hcount[10:1] >= 280 && hcount[10:1] <= 360 && vcount[9:0] >= 220 && vcount[9:0] < 240 && score[7:4] == 8) begin
      game_en <= 1;
      game_addr <= ((hcount[10:1] - 280) + (vcount[9:0] - 220) * 80);
   end else if (hcount[10:1] >= 280 && hcount[10:1] <= 360 && vcount[9:0] >= 220 && vcount[9:0] < 240 && lives == 0) begin
      game_en <= 1;
      game_addr <= (hcount[10:1] - 280) + (vcount[9:0] - 220) * 80 + 1600;
   end else begin
      game_en <= 0;
      game_addr <= 0;
   end
end

// Paddle Logic
logic [10:0] paddle_addr;
logic [23:0] paddle_output;
logic paddle_en;
paddle_ROM paddle(.address(paddle_addr),.clock(clk),.q(paddle_output));
always_ff @(posedge clk) begin
   if (hcount[10:1] >= paddle_pos-40 && hcount[10:1] < paddle_pos+40 && vcount[9:0] >= 450 && vcount[9:0] <= 470) begin
      paddle_en <= 1;
      paddle_addr <= (hcount[10:1] - (paddle_pos-40)) + (vcount[9:0] - 450) * 80;
   end else begin
      paddle_en <= 0;
      paddle_addr <= 0;
   end
end

// Score Logic
logic [11:0] score_addr;
logic [23:0] score_output;
logic score_en;
score_ROM score_mem(.address(score_addr),.clock(clk),.q(score_output));
always_ff @(posedge clk) begin
   if (hcount[10:1] >= 16 && hcount[10:1] <= 32 && vcount[9:0] >= 8 && vcount[9:0] < 24) begin
      score_en <= 1;
      score_addr <= ((hcount[10:1] - 16) + (vcount[9:0] - 8) * 16) + (256 * score[7:4]);
   end else if (hcount[10:1] >= 40 && hcount[10:1] <= 56 && vcount[9:0] >= 8 && vcount[9:0] < 24) begin
      score_en <= 1;
      score_addr <= ((hcount[10:1] - 40) + (vcount[9:0] - 8) * 16) + (256 * score[3:0]);
   end else begin
      score_en <= 0;
      score_addr <= 0;
   end
end


// First Brick Row
logic [10:0] pink_addr;
logic [23:0] pink_output;
logic pink_en;
pink_ROM pink(.address(pink_addr),.clock(clk),.q(pink_output));
always_ff @(posedge clk) begin
   if (blocks[0] == 1 && hcount[10:1] >= 15 && hcount[10:1] <= 79 && vcount[9:0] >= 32 && vcount[9:0] < 64) begin
      pink_en <= 1;
      pink_addr <= (hcount[10:1]-15) + (vcount[9:0] - 32) * 64;
   end else if (blocks[1] == 1 && hcount[10:1] >= 93 && hcount[10:1] <= 157 && vcount[9:0] >= 32 && vcount[9:0] < 64) begin
```

```
      pink_en <= 1;
      pink_addr <= (hcount[10:1]-93) + (vcount[9:0] - 32) * 64;
   end else if (blocks[2] == 1 && hcount[10:1] >= 171 && hcount[10:1] <= 235 && vcount[9:0] >= 32 && vcount[9:0] < 64) begin
      pink_en <= 1;
      pink_addr <= (hcount[10:1]-171) + (vcount[9:0] - 32) * 64;
   end else if (blocks[3] == 1 && hcount[10:1] >= 249 && hcount[10:1] <= 313 && vcount[9:0] >= 32 && vcount[9:0] < 64) begin
      pink_en <= 1;
      pink_addr <= (hcount[10:1]-249) + (vcount[9:0] - 32) * 64;
   end else if (blocks[4] == 1 && hcount[10:1] >= 327 && hcount[10:1] <= 391 && vcount[9:0] >= 32 && vcount[9:0] < 64) begin
      pink_en <= 1;
      pink_addr <= (hcount[10:1]-327) + (vcount[9:0] - 32) * 64;
   end else if (blocks[5] == 1 && hcount[10:1] >= 405 && hcount[10:1] <= 469 && vcount[9:0] >= 32 && vcount[9:0] < 64) begin
      pink_en <= 1;
      pink_addr <= (hcount[10:1]-405) + (vcount[9:0] - 32) * 64;
   end else if (blocks[6] == 1 && hcount[10:1] >= 483 && hcount[10:1] <= 547 && vcount[9:0] >= 32 && vcount[9:0] < 64) begin
      pink_en <= 1;
      pink_addr <= (hcount[10:1]-483) + (vcount[9:0] - 32) * 64;
   end else if (blocks[7] == 1 && hcount[10:1] >= 561 && hcount[10:1] <= 625 && vcount[9:0] >= 32 && vcount[9:0] < 64) begin
      pink_en <= 1;
      pink_addr <= (hcount[10:1]-561) + (vcount[9:0] - 32) * 64;
   end else begin
      pink_en <= 0;
      pink_addr <= 0;
   end
end

// Second Brick Row
logic green_en;
logic [10:0] green_addr;
logic [23:0] green_output;
green_ROM green(.address(green_addr),.clock(clk),.q(green_output));
always_ff @(posedge clk) begin
   if (blocks[8] == 1 && hcount[10:1] >= 15 && hcount[10:1] <= 79 && vcount[9:0] >= 74 && vcount[9:0] < 106) begin
      green_en <= 1;
      green_addr <= (hcount[10:1]-15) + (vcount[9:0] - 74) * 64;
   end else if (blocks[9] == 1 && hcount[10:1] >= 93 && hcount[10:1] <= 157 && vcount[9:0] >= 74 && vcount[9:0] < 106) begin
      green_en <= 1;
      green_addr <= (hcount[10:1]-93) + (vcount[9:0] - 74) * 64;
   end else if (blocks[10] == 1 && hcount[10:1] >= 171 && hcount[10:1] <= 235 && vcount[9:0] >= 74 && vcount[9:0] < 106) begin
      green_en <= 1;
      green_addr <= (hcount[10:1]-171) + (vcount[9:0] - 74) * 64;
   end else if (blocks[11] == 1 && hcount[10:1] >= 249 && hcount[10:1] <= 313 && vcount[9:0] >= 74 && vcount[9:0] < 106) begin
      green_en <= 1;
      green_addr <= (hcount[10:1]-249) + (vcount[9:0] - 74) * 64;
   end else if (blocks[12] == 1 && hcount[10:1] >= 327 && hcount[10:1] <= 391 && vcount[9:0] >= 74 && vcount[9:0] < 106) begin
      green_en <= 1;
      green_addr <= (hcount[10:1]-327) + (vcount[9:0] - 74) * 64;
   end else if (blocks[13] == 1 && hcount[10:1] >= 405 && hcount[10:1] <= 469 && vcount[9:0] >= 74 && vcount[9:0] < 106) begin
      green_en <= 1;
      green_addr <= (hcount[10:1]-405) + (vcount[9:0] - 74) * 64;
   end else if (blocks[14] == 1 && hcount[10:1] >= 483 && hcount[10:1] <= 547 && vcount[9:0] >= 74 && vcount[9:0] < 106) begin
      green_en <= 1;
      green_addr <= (hcount[10:1]-483) + (vcount[9:0] - 74) * 64;
   end else if (blocks[15] == 1 && hcount[10:1] >= 561 && hcount[10:1] <= 625 && vcount[9:0] >= 74 && vcount[9:0] < 106) begin
      green_en <= 1;
      green_addr <= (hcount[10:1]-561) + (vcount[9:0] - 74) * 64;
   end else begin
      green_en <= 0;
      green_addr <= 0;
   end
end
```

```
// Third Brick Row
logic purple_en;
logic [10:0] purple_addr;
logic [23:0] purple_output;
purple_ROM purple(.address(purple_addr),.clock(clk),.q(purple_output));
always_ff @(posedge clk) begin
    if (blocks[16] == 1 && hcount[10:1] >= 15 && hcount[10:1] <= 79 && vcount[9:0] >= 116 && vcount[9:0] < 148) begin
      purple_en <= 1;
      purple_addr <= (hcount[10:1]-15) + (vcount[9:0] - 116) * 64;
   end else if (blocks[17] == 1 && hcount[10:1] >= 93 && hcount[10:1] <= 157 && vcount[9:0] >= 116 && vcount[9:0] < 148) begin
      purple_en <= 1;
      purple_addr <= (hcount[10:1]-93) + (vcount[9:0] - 116) * 64;
   end else if (blocks[18] == 1 && hcount[10:1] >= 171 && hcount[10:1] <= 235 && vcount[9:0] >= 116 && vcount[9:0] < 148) begin
      purple_en <= 1;
      purple_addr <= (hcount[10:1]-171) + (vcount[9:0] - 116) * 64;
   end else if (blocks[19] == 1 && hcount[10:1] >= 249 && hcount[10:1] <= 313 && vcount[9:0] >= 116 && vcount[9:0] < 148) begin
      purple_en <= 1;
      purple_addr <= (hcount[10:1]-249) + (vcount[9:0] - 116) * 64;
   end else if (blocks[20] == 1 && hcount[10:1] >= 327 && hcount[10:1] <= 391 && vcount[9:0] >= 116 && vcount[9:0] < 148) begin
      purple_en <= 1;
      purple_addr <= (hcount[10:1]-327) + (vcount[9:0] - 116) * 64;
   end else if (blocks[21] == 1 && hcount[10:1] >= 405 && hcount[10:1] <= 469 && vcount[9:0] >= 116 && vcount[9:0] < 148) begin
      purple_en <= 1;
      purple_addr <= (hcount[10:1]-405) + (vcount[9:0] - 116) * 64;
   end else if (blocks[22] == 1 && hcount[10:1] >= 483 && hcount[10:1] <= 547 && vcount[9:0] >= 116 && vcount[9:0] < 148) begin
      purple_en <= 1;
      purple_addr <= (hcount[10:1]-483) + (vcount[9:0] - 116) * 64;
   end else if (blocks[23] == 1 && hcount[10:1] >= 561 && hcount[10:1] <= 625 && vcount[9:0] >= 116 && vcount[9:0] < 148) begin
      purple_en <= 1;
      purple_addr <= (hcount[10:1]-561) + (vcount[9:0] - 116) * 64;
   end else begin
      purple_en <= 0;
      purple_addr <= 0;
   end
end

// Fourth Brick Row
logic orange_en;
logic [10:0] orange_addr;
logic [23:0] orange_output;
orange_ROM orange(.address(orange_addr),.clock(clk),.q(orange_output));
always_ff @(posedge clk) begin
    if (blocks[24] == 1 && hcount[10:1] >= 15 && hcount[10:1] <= 79 && vcount[9:0] >= 158 && vcount[9:0] < 190) begin
      orange_en <= 1;
      orange_addr <= (hcount[10:1]-15) + (vcount[9:0] - 158) * 64;
   end else if (blocks[25] == 1 && hcount[10:1] >= 93 && hcount[10:1] <= 157 && vcount[9:0] >= 158 && vcount[9:0] < 190) begin
      orange_en <= 1;
      orange_addr <= (hcount[10:1]-93) + (vcount[9:0] - 158) * 64;
   end else if (blocks[26] == 1 && hcount[10:1] >= 171 && hcount[10:1] <= 235 && vcount[9:0] >= 158 && vcount[9:0] < 190) begin
      orange_en <= 1;
      orange_addr <= (hcount[10:1]-171) + (vcount[9:0] - 158) * 64;
   end else if (blocks[27] == 1 && hcount[10:1] >= 249 && hcount[10:1] <= 313 && vcount[9:0] >= 158 && vcount[9:0] < 190) begin
      orange_en <= 1;
      orange_addr <= (hcount[10:1]-249) + (vcount[9:0] - 158) * 64;
   end else if (blocks[28] == 1 && hcount[10:1] >= 327 && hcount[10:1] <= 391 && vcount[9:0] >= 158 && vcount[9:0] < 190) begin
      orange_en <= 1;
      orange_addr <= (hcount[10:1]-327) + (vcount[9:0] - 158) * 64;
   end else if (blocks[29] == 1 && hcount[10:1] >= 405 && hcount[10:1] <= 469 && vcount[9:0] >= 158 && vcount[9:0] < 190) begin
      orange_en <= 1;
      orange_addr <= (hcount[10:1]-405) + (vcount[9:0] - 158) * 64;
   end else if (blocks[30] == 1 && hcount[10:1] >= 483 && hcount[10:1] <= 547 && vcount[9:0] >= 158 && vcount[9:0] < 190) begin
      orange_en <= 1;
      orange_addr <= (hcount[10:1]-483) + (vcount[9:0] - 158) * 64;
```

```
    end else if (blocks[31] == 1 && hcount[10:1] >= 561 && hcount[10:1] <= 625 && vcount[9:0] >= 158 && vcount[9:0] < 190) begin
       orange_en <= 1;
       orange_addr <= (hcount[10:1]-561) + (vcount[9:0] - 158) * 64;
    end else begin
       orange_en <= 0;
       orange_addr <= 0;
    end
  end

  always_comb begin
    {VGA_R, VGA_G, VGA_B} = {8'h0, 8'h0, 8'h0};
    if (VGA_BLANK_n)
       // Display Ball
       if (ball_en)
          {VGA_R, VGA_G, VGA_B} = ball_output;
       // Display Paddle
       else if (paddle_en)
          {VGA_R, VGA_G, VGA_B} = paddle_output;
       // Display First Row of Blocks
       else if (pink_en)
          {VGA_R, VGA_G, VGA_B} = pink_output;
       // Display Second Row of Blocks
       else if (green_en)
          {VGA_R, VGA_G, VGA_B} = green_output;
       // Display Third Row of Blocks
       else if (purple_en)
          {VGA_R, VGA_G, VGA_B} = purple_output;
       // Display Fourth Row of Blocks
       else if (orange_en)
          {VGA_R, VGA_G, VGA_B} = orange_output;
       // Display Lives
       else if (lives_en)
          {VGA_R, VGA_G, VGA_B} = lives_output;
       // Display Score
       else if (score_en)
          {VGA_R, VGA_G, VGA_B} = score_output;
       // Display Game Win/Loss
       else if (game_en)
          {VGA_R, VGA_G, VGA_B} = game_output;
       else
          {VGA_R, VGA_G, VGA_B} = {background_r, background_g, background_b};

       if (~start)
          {VGA_R, VGA_G, VGA_B} = {8'h0, 8'h0, 8'h0};
  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
```

8.2 Software

*** hello.c ***

```c
/*
* Userspace program that communicates with the vga_ball device driver
* through ioctls
*
* Stephen A. Edwards
* Columbia University
*/
#include "usbcontroller.h"
#include "vga_ball.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

#define BALL_RADIUS         8
#define BLOCK_HEIGHT        16
#define BLOCK_WIDTH         32
#define PADDLE_WIDTH        40
#define PADDLE_TOP          450
#define PADDLE_BOTTOM       470
#define PADDLE_ACCELERATION 10
#define PADDLE_DECELERATION 1

#define BRICK_HIT_SOUND     1
#define PADDLE_HIT_SOUND    2
#define GAME_WIN_SOUND      4
#define LIFE_LOST_SOUND     8

struct libusb_device_handle *controller;
struct usb_controller_packet packet;
uint8_t endpoint_address;
int transferred, vga_ball_fd;

/* Read and print the background color */
void print_background_color()
{
 vga_ball_arg_t vla;

 if (ioctl(vga_ball_fd, VGA_BALL_READ_BACKGROUND, &vla))
 {
   perror("ioctl(VGA_BALL_READ_BACKGROUND) failed");
   return;
 }
 printf("%02x %02x %02x\n",
        vla.background.red, vla.background.green, vla.background.blue);
}
```

```c
/* Set the background color */
void set_background_color(const vga_ball_color_t *c)
{
 vga_ball_arg_t vla;
 vla.background = *c;
 if (ioctl(vga_ball_fd, VGA_BALL_WRITE_BACKGROUND, &vla))
 {
   perror("ioctl(VGA_BALL_SET_BACKGROUND) failed");
   return;
 }
}

/* Read and print the position */
void print_position()
{
 vga_ball_arg_t vla;

 if (ioctl(vga_ball_fd, VGA_BALL_READ_POSITION, &vla))
 {
   perror("ioctl(VGA_BALL_READ_POSITION) failed");
   return;
 }
 int ball_x = vla.position.ball_x;
 int ball_y = vla.position.ball_y;
 int paddle = vla.position.paddle;
 // printf("Ball: %d %d Paddle: %d\n", ball_x, ball_y, paddle);
}

/* Set the ball position */
void set_position(const vga_positions_t *c)
{
 vga_ball_arg_t vla;
 vla.position = *c;
 if (ioctl(vga_ball_fd, VGA_BALL_WRITE_POSITION, &vla))
 {
   perror("ioctl(VGA_BALL_SET_POSITION) failed");
   return;
 }
}

/* Set the ball position */
void set_blocks(const vga_blocks_t *c)
{
 vga_ball_arg_t vla;
 vla.blocks = *c;
 if (ioctl(vga_ball_fd, VGA_BLOCKS_WRITE_POSITION, &vla))
 {
   perror("ioctl(VGA_BLOCKS_SET_POSITION) failed");
   return;
 }
}

void set_info(const vga_gameinfo_t *c)
{
```

```
 vga_ball_arg_t vla;
 vla.gameinfo = *c;
 if (ioctl(vga_ball_fd, VGA_BALL_WRITE_INFO, &vla))
 {
   perror("ioctl(VGA_BALL_WRITE_INFO) failed");
   return;
 }
}

vga_positions_t encode_position(int ball_x, int ball_y, int paddle)
{
   vga_positions_t position;
   position.ball_x = ball_x & 0x3FF;
   position.ball_y = ball_y & 0x1FF;
   position.paddle = paddle & 0x3FF;
   return position;
}

unsigned short get_score(int score)
{
   char score1 = score / 10;
   char score2 = score % 10;
   return (score1 << 4) | score2;
}

unsigned short get_lives(int lives)
{
   if (lives == 1)
      return 1;
   else if (lives == 2)
      return 3;
   else if (lives == 3)
      return 7;
   return 0;
}

vga_gameinfo_t encode_info(int lives, int audio, int score)
{
   vga_gameinfo_t info;
   info.lives = get_lives(lives) & 7;
   info.audio = audio & 0xF;
   info.score = get_score(score) & 0xFF;
   return info;
}

int brick_hit(game_info_t *game_info, int block_x, int block_y)
{
 // Hits bottom of block
 if (game_info->ball_y > block_y + BLOCK_HEIGHT - BALL_RADIUS/2 &&
     game_info->ball_y < block_y + BLOCK_HEIGHT + BALL_RADIUS &&
     game_info->ball_x > block_x - BLOCK_WIDTH - BALL_RADIUS &&
     game_info->ball_x < block_x + BLOCK_WIDTH + BALL_RADIUS &&
     game_info->ball_U == 0)
 {
```

```
  printf("bottom\n");
  game_info->ball_U = 1;
  return 1;
 }
 // Hits top of block
 else if (game_info->ball_y < block_y - BLOCK_HEIGHT + BALL_RADIUS/2 &&
          game_info->ball_y > block_y - BLOCK_HEIGHT - BALL_RADIUS &&
          game_info->ball_x > block_x - BLOCK_WIDTH - BALL_RADIUS &&
          game_info->ball_x < block_x + BLOCK_WIDTH + BALL_RADIUS &&
          game_info->ball_U == 1)
 {
  printf("top\n");
  game_info->ball_U = 0;
  return 1;
 }
 // Hits left side of block
 else if (game_info->ball_y > block_y - BLOCK_HEIGHT &&
          game_info->ball_y < block_y + BLOCK_HEIGHT &&
          game_info->ball_x > block_x - BLOCK_WIDTH - 2*BALL_RADIUS &&
          game_info->ball_x < block_x - BLOCK_WIDTH + 2*BALL_RADIUS &&
          game_info->ball_v > 0)
 {
  printf("left side\n");
  game_info->ball_v = -game_info->ball_v;
  return 1;
 }
 // Hits right side of block
 else if (game_info->ball_y > block_y - BLOCK_HEIGHT &&
          game_info->ball_y < block_y + BLOCK_HEIGHT &&
          game_info->ball_x > block_x + BLOCK_WIDTH - 2*BALL_RADIUS &&
          game_info->ball_x < block_x + BLOCK_WIDTH + 2*BALL_RADIUS &&
          game_info->ball_v < 0)
 {
  printf("right side\n");
  game_info->ball_v = -game_info->ball_v;
  return 1;
 }
 return 0;
}

void update_position(game_info_t *game_info)
{
  int audio = 0;
  for (uint32_t i = 0; i < 8; i++)
  {
     if (game_info->blocks & (1 << i) && brick_hit(game_info, 47 + i * 78, 48))
     {
        game_info->blocks &= ~(1 << i);
        game_info->score += 4;
        audio = BRICK_HIT_SOUND;
     }
     if (game_info->blocks & (1 << (i+8)) && brick_hit(game_info, 47 + i * 78, 90))
     {
        game_info->blocks &= ~(1 << (i + 8));
```

```
        game_info->score += 3;
        audio = BRICK_HIT_SOUND;
     }
     if (game_info->blocks & (1 << (i+16)) && brick_hit(game_info, 47 + i * 78, 132))
     {
        game_info->blocks &= ~(1 << (i + 16));
        game_info->score += 2;
        audio = BRICK_HIT_SOUND;
     }
     if (game_info->blocks & (1 << (i+24)) && brick_hit(game_info, 47 + i * 78, 174))
     {
        game_info->blocks &= ~(1 << (i + 24));
        game_info->score += 1;
        audio = BRICK_HIT_SOUND;
      }
  }
// Ball hits side of paddle
if ((game_info->ball_y >= PADDLE_TOP &&
     game_info->ball_x > game_info->paddle_x - 75 &&
     game_info->ball_x < game_info->paddle_x - 45) ||
    (game_info->ball_y >= PADDLE_TOP &&
     game_info->ball_x > game_info->paddle_x + 45 &&
     game_info->ball_x < game_info->paddle_x + 75))
  {
     // if paddle moving same direction as ball then speed up
     if ((game_info->ball_v > 0 && game_info->paddle_v > 0) ||
         (game_info->ball_v < 0 && game_info->paddle_v < 0))
     {
        game_info->ball_v += game_info->paddle_v / 2;
     }
     else
     {
        game_info->ball_v = -game_info->ball_v;
     }
     audio = PADDLE_HIT_SOUND;
  }
  // Ball hits side of screen
  if ((game_info->ball_x < -game_info->ball_v + BALL_RADIUS) ||
      (game_info->ball_x > 620 - game_info->ball_v + BALL_RADIUS))
  {
     game_info->ball_v = -game_info->ball_v;
  }
  if (game_info->ball_U)
  {
     game_info->ball_y += 6;
     // Ball hits top of paddle
     if ((game_info->ball_y > PADDLE_TOP - BALL_RADIUS*2 &&
          game_info->ball_y < PADDLE_TOP &&
          game_info->ball_x > game_info->paddle_x - PADDLE_WIDTH - BALL_RADIUS &&
          game_info->ball_x < game_info->paddle_x + PADDLE_WIDTH + BALL_RADIUS))
     {
        game_info->ball_U = 0;
        game_info->ball_v = game_info->last_ball_v + game_info->paddle_v / 2;
        audio = PADDLE_HIT_SOUND;
```

```
      }
      // Ball is out of bounds at bottom
      else if (game_info->ball_y > 460)
      {
         game_info->ball_y = 220;
         game_info->ball_U = 1;
         game_info->ball_v = 4;
         game_info->lives--;
         audio = LIFE_LOST_SOUND;
      }
   }
 // Ball hits top of screen
 else
 {
    game_info->ball_y -= 6;
    if (game_info->ball_y < 20)
       game_info->ball_U = 1;
 }

 game_info->ball_x += game_info->ball_v;
 game_info->last_ball_v = game_info->ball_v;

 vga_positions_t position = encode_position(game_info->ball_x, game_info->ball_y, game_info->paddle_x);
 vga_gameinfo_t info = encode_info(game_info->lives, audio, game_info->score);
 set_position(&position);
 set_info(&info);
 set_blocks(&game_info->blocks);
}

int main()
{
 vga_ball_arg_t vla;
 int i;
 static const char filename[] = "/dev/vga_ball";

 static const vga_ball_color_t colors[] = {
     {0xff, 0x00, 0x00}, /* Red */
     {0x00, 0xff, 0x00}, /* Green */
     {0x00, 0x00, 0xff}, /* Blue */
     {0xff, 0xff, 0x00}, /* Yellow */
     {0x00, 0xff, 0xff}, /* Cyan */
     {0xff, 0x00, 0xff}, /* Magenta */
     {0x80, 0x80, 0x80}, /* Gray */
     {0x00, 0x00, 0x00}, /* Black */
                         // { 0xff, 0xff, 0xff }  /* White */
 };

#define COLORS 8

 printf("VGA ball Userspace program started\n");

 if ((vga_ball_fd = open(filename, O_RDWR)) == -1)
 {
    fprintf(stderr, "could not open %s\n", filename);
```

```
    return -1;
 }

 if ((controller = open_controller(&endpoint_address)) == NULL)
 {
    fprintf(stderr, "Did not find a controller\n");
    exit(1);
 }

 printf("initial state: ");
 print_background_color();

 game_info_t game_info;
 game_info.ball_x = 300;
 game_info.ball_y = 220;
 game_info.ball_v = 4;
 game_info.last_ball_v = 4;
 game_info.ball_U = 1;
 game_info.paddle_x = 300;
 game_info.paddle_v = 0;
 game_info.blocks = 0xFFFFFFFF;
 game_info.lives = 3;
 game_info.score = 0;

 while (game_info.lives > 0 && game_info.blocks != 0)
 {
    int result = libusb_interrupt_transfer(controller, endpoint_address, (unsigned char *)&packet, sizeof(packet),
&transferred, 20);
    if (result == LIBUSB_SUCCESS && transferred == sizeof(packet))
    {
       if (packet.direction == USB_RIGHT)
       {
          game_info.paddle_v = PADDLE_ACCELERATION;
       }
       else if (packet.direction == USB_LEFT)
       {
          game_info.paddle_v = -PADDLE_ACCELERATION;
       }
    }
    else if (result == LIBUSB_ERROR_TIMEOUT) // Timeout occurred
    {
       if (game_info.paddle_v > 0)
       {
          game_info.paddle_v -= PADDLE_DECELERATION;
       }
       else if (game_info.paddle_v < 0)
       {
          game_info.paddle_v += PADDLE_DECELERATION;
       }
    }
    else
    {
       fprintf(stderr, "libusb_interrupt_transfer error: %s\n", libusb_error_name(result));
    }
```

```
    game_info.paddle_x += game_info.paddle_v;
    if (game_info.paddle_x > 640 - PADDLE_WIDTH)
        game_info.paddle_x = 640 - PADDLE_WIDTH;
    if (game_info.paddle_x < PADDLE_WIDTH)
        game_info.paddle_x = PADDLE_WIDTH;

    set_background_color(&colors[7]);
    update_position(&game_info);
    usleep(50000);
 }

 int audio = game_info.blocks == 0 ? GAME_WIN_SOUND : 0;
 vga_gameinfo_t info = encode_info(game_info.lives, audio, game_info.score);
 set_info(&info);
 usleep(50000);
 info = encode_info(game_info.lives, 0, game_info.score);
 set_info(&info);

 printf("VGA BALL Userspace program terminating\n");
 return 0;
}
```

*** vga_ball.c ***

```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_ball.ko
*
* Check code style with
* checkpatch.pl --file --no-tree vga_ball.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_ball.h"

#define DRIVER_NAME "vga_ball"

/* Device registers */
#define BACKGROUND_COLORS(x)        (x)
#define BALL_PADDLE_POSITION(x)     ((x)+4)
#define BLOCK_STATUS(x)             ((x)+8)
#define GAME_INFO(x)                ((x)+12)


/*
* Information about our device
*/
struct vga_ball_dev {
   struct resource res; /* Resource: our registers */
   void __iomem *virtbase; /* Where registers can be accessed in memory */
   vga_ball_color_t background;
   vga_positions_t position;
   vga_blocks_t blocks;
   vga_gameinfo_t gameinfo;
} dev;

/*
* Write segments of a single digit
* Assumes digit is in range and the device information has been set up
*/
static void write_background(vga_ball_color_t *background)
```

```
{
   uint32_t background_colors = background->red | (background->green << 8) | (background->blue << 16);
   iowrite32(background_colors, BACKGROUND_COLORS(dev.virtbase));
   dev.background = *background;
}

static void write_position(vga_positions_t *positions)
{
   uint32_t position = positions->ball_x | (positions->ball_y << 10) | (positions->paddle << 19);
   iowrite32(position, BALL_PADDLE_POSITION(dev.virtbase));
   dev.position = *positions;
}

static void write_blocks(vga_blocks_t *blocks)
{
   iowrite32(*blocks, BLOCK_STATUS(dev.virtbase));
   dev.blocks = *blocks;
}

static void write_info(vga_gameinfo_t *gameinfo)
{
   uint32_t info = gameinfo->lives | (gameinfo->audio << 3) | (gameinfo->score << 7) | (1 << 15);
   iowrite32(info, GAME_INFO(dev.virtbase));
   dev.gameinfo = *gameinfo;
}

/*
* Handle ioctl() calls from userspace:
* Read or write the segments on single digits.
* Note extensive error checking of arguments
*/
static long vga_ball_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
   vga_ball_arg_t vla;

   switch (cmd) {
   case VGA_BALL_WRITE_BACKGROUND:
       if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                  sizeof(vga_ball_arg_t)))
           return -EACCES;
       write_background(&vla.background);
       break;

   case VGA_BALL_READ_BACKGROUND:
       vla.background = dev.background;
       if (copy_to_user((vga_ball_arg_t *) arg, &vla,
                sizeof(vga_ball_arg_t)))
           return -EACCES;
       break;

   case VGA_BALL_WRITE_POSITION:
       if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                  sizeof(vga_ball_arg_t)))
           return -EACCES;
       write_position(&vla.position);
       break;

   case VGA_BALL_READ_POSITION:
       vla.position = dev.position;
       if (copy_to_user((vga_ball_arg_t *) arg, &vla,
                sizeof(vga_ball_arg_t)))
           return -EACCES;
       break;
```

```
  case VGA_BLOCKS_WRITE_POSITION:
      if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                 sizeof(vga_ball_arg_t)))
          return -EACCES;
      write_blocks(&vla.blocks);
      break;

  case VGA_BLOCKS_READ_POSITION:
      vla.blocks = dev.blocks;
      if (copy_to_user((vga_ball_arg_t *) arg, &vla,
               sizeof(vga_ball_arg_t)))
          return -EACCES;
      break;

  case VGA_BALL_WRITE_INFO:
      if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                 sizeof(vga_ball_arg_t)))
          return -EACCES;
      write_info(&vla.gameinfo);
      break;

  case VGA_BALL_READ_INFO:
      vla.gameinfo = dev.gameinfo;
      if (copy_to_user((vga_ball_arg_t *) arg, &vla,
               sizeof(vga_ball_arg_t)))
          return -EACCES;
      break;

  default:
      return -EINVAL;
  }

  return 0;
}

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

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

/*
* Initialization code: get resources (registers) and display
* a welcome message
*/
static int __init vga_ball_probe(struct platform_device *pdev)
{
  vga_ball_color_t beige = { 0xf9, 0xe4, 0xb7 };
  int ret;

  /* Register ourselves as a misc device: creates /dev/vga_ball */
  ret = misc_register(&vga_ball_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;
   }

   /* Set an initial color */
       write_background(&beige);

   return 0;

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

/* Clean-up code: release resources */
static int vga_ball_remove(struct platform_device *pdev)
{
   iounmap(dev.virtbase);
   release_mem_region(dev.res.start, resource_size(&dev.res));
   misc_deregister(&vga_ball_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_ball_of_match[] = {
   { .compatible = "csee4840,vga_ball-1.0" },
   {},
};
MODULE_DEVICE_TABLE(of, vga_ball_of_match);
#endif

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

/* Called when the module is loaded: set things up */
static int __init vga_ball_init(void)
{
   pr_info(DRIVER_NAME ": init\n");
```

```
   return platform_driver_probe(&vga_ball_driver, vga_ball_probe);
}

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

module_init(vga_ball_init);
module_exit(vga_ball_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA ball driver");
```

*** vga_ball.h ***

```
#ifndef _VGA_BALL_H
#define _VGA_BALL_H

#include <linux/ioctl.h>

typedef struct {
   unsigned char red, green, blue;
} vga_ball_color_t;

typedef struct {
   unsigned short ball_x;
   unsigned short ball_y;
   unsigned short paddle;
} vga_positions_t;

typedef struct {
   unsigned short lives;
   unsigned short audio;
   unsigned short score;
} vga_gameinfo_t;

typedef uint32_t vga_blocks_t;

typedef struct {
   int lives;
   int score;
   int ball_x;
   int ball_y;
   int ball_v;
   int last_ball_v;
   int paddle_x;
   int paddle_v;
   unsigned char ball_U;
   uint32_t blocks;
} game_info_t;

typedef struct {
   vga_ball_color_t background;
   vga_positions_t position;
   vga_blocks_t blocks;
   vga_gameinfo_t gameinfo;
} vga_ball_arg_t;

#define VGA_BALL_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_BALL_WRITE_BACKGROUND   _IOW(VGA_BALL_MAGIC, 1, vga_ball_arg_t *)
#define VGA_BALL_READ_BACKGROUND    _IOR(VGA_BALL_MAGIC, 2, vga_ball_arg_t *)
#define VGA_BALL_WRITE_POSITION     _IOW(VGA_BALL_MAGIC, 3, vga_ball_arg_t *)
#define VGA_BALL_READ_POSITION      _IOR(VGA_BALL_MAGIC, 4, vga_ball_arg_t *)
#define VGA_BLOCKS_WRITE_POSITION   _IOW(VGA_BALL_MAGIC, 5, vga_ball_arg_t *)
#define VGA_BLOCKS_READ_POSITION    _IOR(VGA_BALL_MAGIC, 6, vga_ball_arg_t *)
#define VGA_BALL_WRITE_INFO         _IOW(VGA_BALL_MAGIC, 7, vga_ball_arg_t *)
#define VGA_BALL_READ_INFO          _IOR(VGA_BALL_MAGIC, 8, vga_ball_arg_t *)

#endif
```

*** usbcontroller.c ***

```c
#include "usbcontroller.h"

#include <stdio.h>
#include <stdlib.h>

struct libusb_device_handle *open_controller(uint8_t *endpoint_address)
{
   libusb_device **devs;
   struct libusb_device_handle *controller = NULL;
   struct libusb_device_descriptor desc;
   ssize_t num_devs, d;
   uint8_t i, k;

   if (libusb_init(NULL) < 0)
   {
       fprintf(stderr, "Error: libusb_init failed\n");
       exit(1);
   }

   if ((num_devs = libusb_get_device_list(NULL, &devs)) < 0)
   {
       fprintf(stderr, "Error: libusb_get_device_list failed\n");
       exit(1);
   }

   for (d = 0; d < num_devs; d++)
   {
       libusb_device *dev = devs[d];
       if (libusb_get_device_descriptor(dev, &desc) < 0)
       {
           fprintf(stderr, "Error: libusb_get_device_descriptor failed\n");
           exit(1);
       }

       if (desc.bDeviceClass == LIBUSB_CLASS_PER_INTERFACE)
       {
           struct libusb_config_descriptor *config;
           libusb_get_config_descriptor(dev, 0, &config);
           for (i = 0; i < config->bNumInterfaces; i++)
               for (k = 0; k < config->interface[i].num_altsetting; k++)
               {
                   const struct libusb_interface_descriptor *inter = config->interface[i].altsetting + k;
                   if (inter->bInterfaceClass == LIBUSB_CLASS_HID &&
                       inter->bInterfaceProtocol == USB_VOLUME_CONTROL_PROTOCOL)
                   {
                       int r;
                       if ((r = libusb_open(dev, &controller)) != 0)
                       {
                           fprintf(stderr, "Error: libusb_open failed: %d\n", r);
                           exit(1);
                       }
                       if (libusb_kernel_driver_active(controller, i))
                           libusb_detach_kernel_driver(controller, i);
                       libusb_set_auto_detach_kernel_driver(controller, i);
                       if ((r = libusb_claim_interface(controller, i)) != 0)
                       {
                           fprintf(stderr, "Error: libusb_claim_interface failed: %d\n", r);
                           exit(1);
                       }
                       *endpoint_address = inter->endpoint[0].bEndpointAddress;
                       goto found;
```

```
                    }
                }
            }
        }

found:
    libusb_free_device_list(devs, 1);

    return controller;
}
```

*** usbcontroller.h ***

```
#ifndef _USBCONTROLLER_H
#define _USBCONTROLLER_H

#include <libusb-1.0/libusb.h>

#define USB_VOLUME_CONTROL_PROTOCOL 0

/* Modifier bits */
#define USB_LEFT    234
#define USB_RIGHT   233

struct usb_controller_packet
{
   uint8_t reserved;
   uint8_t direction;
};

/* Find and open the controller.  Argument should point to
  space to store an endpoint address.  Returns NULL if no controller
  device was found. */
extern struct libusb_device_handle *open_controller(uint8_t *);

#endif
```