University of Southern Denmark
Department of Mechanical and Electrical Engineering

Advanced Programmable Electronics - Final Project Report
Master in Electronics

Spring 2022

FHD PONG game design in CMOD A7 FPGA

Alexandros Aslanidis
Exam no.: 498755
Date of birth: 18-11-1994

Supervisor: Anders Stengaard Sørensen

Deadline: 12. August 2022

Abstract

The purpose of this project is to design and implement a Full HD version of the famous pong game using the FPGA provided for this course - CMOD A7-15T.
For that purpose, at first the FPGA must be able to output a HDMI signal with the resolution of 1920x1080 utilizing the on-chip resources of the FPGA. Then, the game logic and graphics are designed in VHDL. To make the game playable, but also for debugging purposes, the receiver part of the UART is realized. The transmitter part of the UART is also realized, though not integral to the project.
Except from the CMOD A7, an additional board provided by the teacher is used to guide the HDMI signals at a proper HDMI output port and provide the A7 with a 24MHz oscillator.

Table of contents

1 Introduction

The goal of this project is to create an FPGA project that takes advantage of the on-chip resources of the FPGA that makes it suitable for high-speed designs. The requirement for the project as derived from the lectures and the description of this course could be :

- Take advantage of the on-chip resources of the FPGA such as MMCM and OSERDES.
- Take advantage of the High Range I/O peak performance of 1.2 Gbps.
- Utilize the UART connection and write simple bash scripts for PC-FPGA communication.
- Create a final functional project such as a simple game in VHDL.

In that spirit, a FHD signal is generated by the CMOD-A7 and a playable recreation of the famous pong game is realized in VHDL.

The project is divided into 3 parts: HDMI generation, Pong graphics design and gameplay design. Each part has a chapter describing the specific requirements and the design of that part.
All tests were made in a SAMSUNG L827C360HS screen.

2 HDMI signal generation

In this section the HDMI signal generated by the CMOD A7 will be discussed. This will include some theory on the HDMI protocol and A7 on-chip resources for the sake of completeness. The VHDL code used for the HDMI signal generation is partially inspired by the VHDL code of Colin Riley [1], which was also provided in the course. The maximum resolution achieved for this project with the provided FPGA is 1920x1080 with a refresh rate of 60Hz. The VHDL code for the whole project as well as the constraint file can be seen in appendix A. An overview of the elaborated design for the HDMI generation in Vivado can be seen in Figure 2.1.

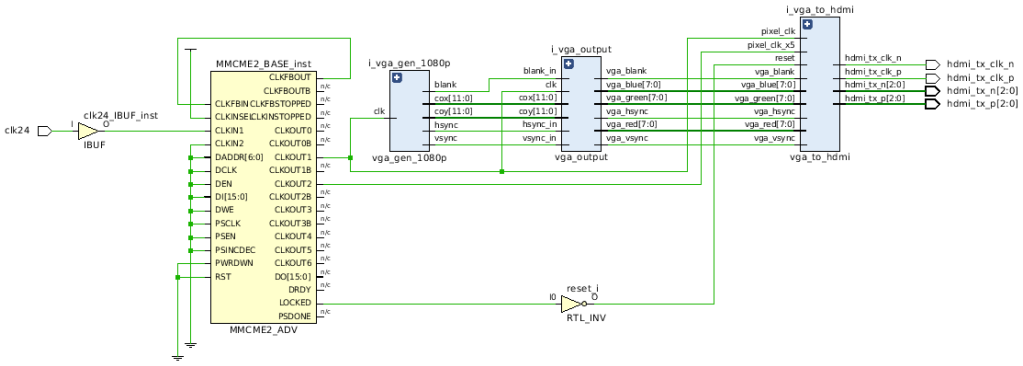

Figure 2.1: Overview of elaborated design in Vivado

The vga_gen module generates the control signals by progressively going through every pixel of each frame each time the pixel clock is rising. It outputs the control signal and the coordinates for each pixel. The vga_output module, paints the pixels with respect to their position and outputs the control signals and the color signals for each pixel. Finally, the vga_to_hdmi module, transforms the signals according to the TMDS protocol. All desired clocks are genereted by an instance of the MMCM primitive. The input clock is a external 24MHz clock provided in the CMOD A7 hat board which was provided in the lectures. All the above, will be discussed in detail in the upcoming subsections.

2.1 HDMI standard and TMDS

TMDS technology serves as the underlying protocol for the HDMI standard. TMDS technology is a differential signaling technique that ensures robust signal transmission with low noise by electromagnetic interference.

The TMDS transmitter receives the 24 bits of parallel data and then prepares the data for transmission by encoding and serializing it. For every RGB color, there are 8 bits. This means that the intensity of each color can vary from 0 to 255. An algorithm is incorporated in the transmitter that encodes these 8 bits into 10 with the intention of minimizing transitions and therefore minimizing radio frequency generation It also ensures a DC balanced frequency. The 9th bit, indicates if and XOR or an XNOR operation has been used to minimize transitions while the 10th indicates if the polarity of the bits has been reversed or not.

In more detail, in the first stage, 9-bits are produced as a representation of the 8-bits received.

The LSB is left as is. The second bit is XOR'ed and XNOR'ed with the LSB and the result of this operation is the 2nd output bit. The result of the previous operation is the XOR'ed and XNOR'ed respectively with the 3rd LSB and the output of this operation is the 3rd LSB in the final representation and so on. After this process is over, one of the two product representations of the initial 8-bit (one for XNOR and one for XOR operations) is chosen as the representation that leads to minimal transitions. The 9th bit is set to 1 if the XOR representation was used and to one if the XNOR one was used [2]. An example of this procedure using the XOR operand can be seen in Figure 2.2. This example would be the worst case scenario, with 7 transitions for 8 bits. The algorithm reduces the number of transitions to 3, which is the best possible result for this case.
The second stage tries to maintain the DC-balance of the data stream. Practically that means, to keep the number of 1s and 0s that are being transmitted by the line as equal as possible to avoid any charge build up [2]. Therefore, if a lot of 0s have been transmitted so far and 0s are also the majority of the current 8-bits, the algorithm reverses the current representation and the 10th bit is equal to 1. If such a thing is not necessary, then the 10th bit is equal to 0.

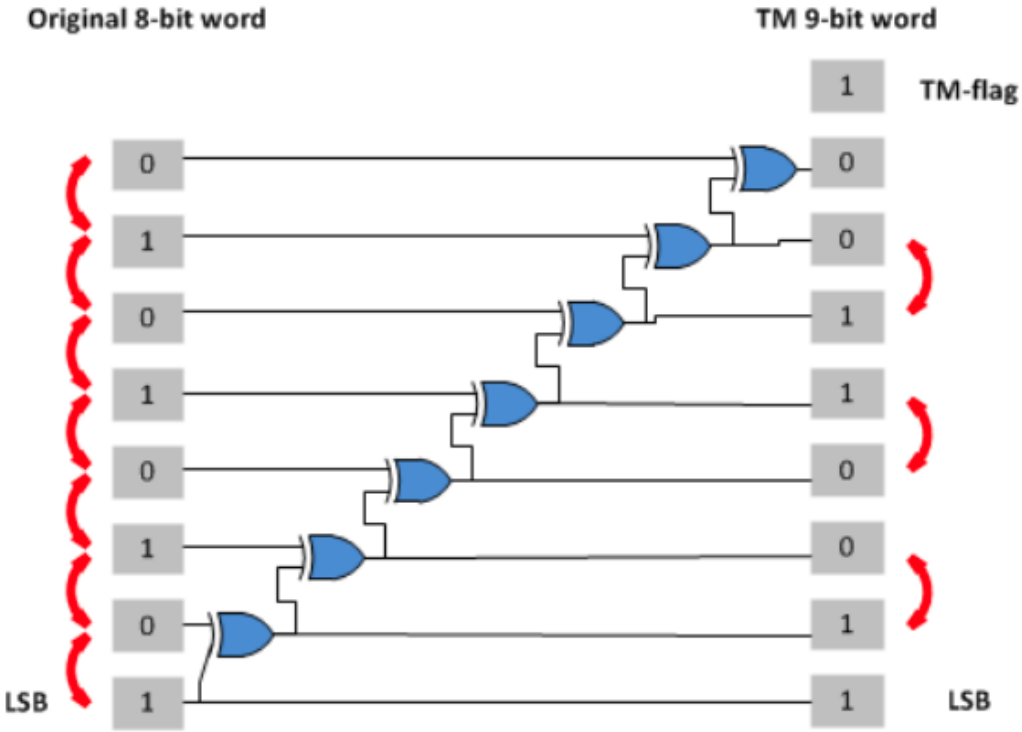

Figure 2.2: 8-bit to 9-bit representation using the XOR operand.

The VHDL code for the TMDS transmitter was used as is in [1].
When the transformation is completed, the data are serialized. For that purpose, in this project, the OSERDES module is used. Normally, a clock that is 10 times faster than the pixel clock is needed. OSERDES has the option for Double Data Rate (DDR) in which the data are delivered on both the rising and the falling edge of the clock. . That means, that a clock 5 times faster than the pixel clock will be sufficient. In fact, to have an parallel to serial converter 10:1, two OSERDES modules must be used and it is possible only in DDR mode [3]. The interconnection between the master OSERDES and the slave can be see in Figure 2.3 and is thoroughly discussed

in [3]. In total, four instances of the serializer are needed - three for each color channel and one for the clock.
This is not the complete HDMI protocol, but the DVID one which is electrically compatible with the HDMI. In true HDMI, during the blanking time some data packages - called Data Island - can be sent that are audio and meta data. In DVID, since there is no audio, the HSYNC, VSYNC are encoded in the blue channel bitstream during the blanking time while the output for the two other color channels is fixed as described in [2].

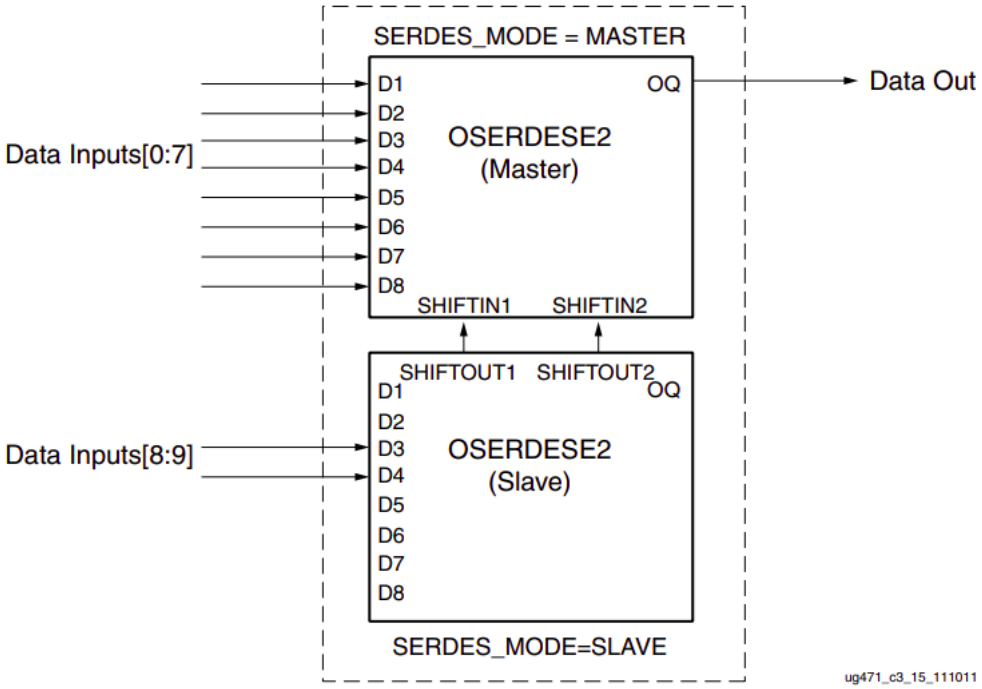

Figure 2.3: Block Diagram of OSERDESE2 Width Expansion. [3]

A simulation of the VHDL 10:1 parallel to serial converter can be seen in Figure 2.4. The module successfully serializes the 10-bit parallel data. The reset value is 0, when the main MMCM component is not locked.

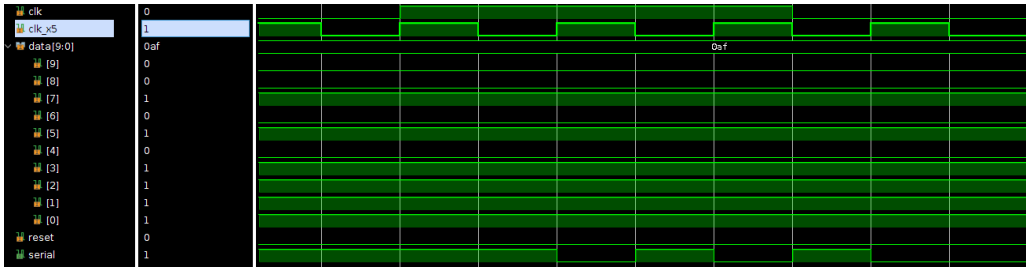

Figure 2.4: Simulation of the 10:1 parallel to serial converter

After the encoding and the serialization, each RGB color component and the clock are transmitted on separate channels in a process known as differential signaling. There are a total of four channels (differential pairs): three for RGB and the fourth for the clock [4]. Thankfully, since the HR I/O

components of the CMOD A7 supports the TMDS_3.3 protocol [3] , it is possible to implement the TMDS transmitter in the FPGA. The Differential Output Buffer primitive (OBUFDS) [5] is used to transform the single line signal into a differential one with the TMDS_3.3 standard selected . For each pair, pins that are close together on the CMOD A7 were selected based on an educated guess that they will have similar traces in the board.
The elaborated design of the module that handles the above can be seen in **??**

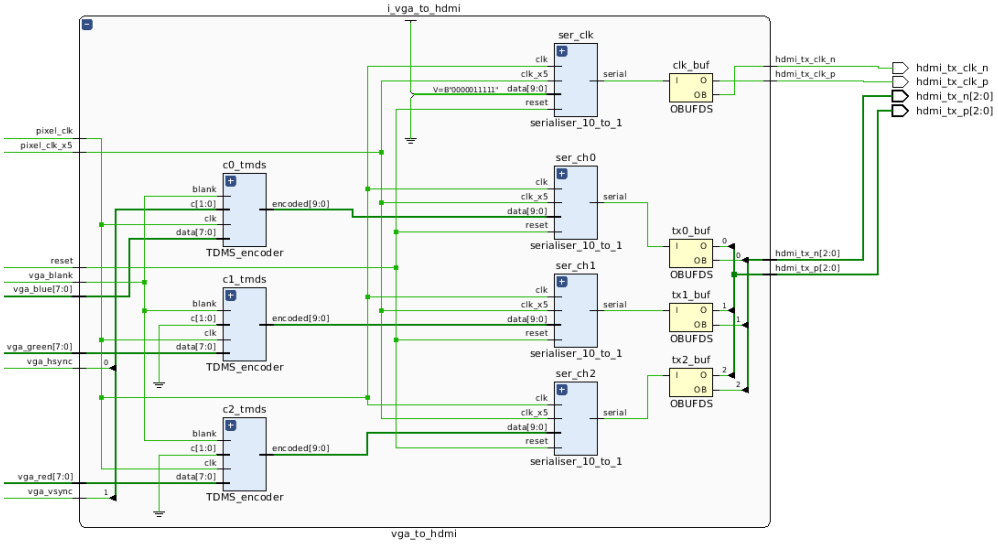

Figure 2.5: Vga to hdmi realization in Vivado.

2.2 VGA signal generation

At first, the video signals are generated as VGA signals. VGA signal generation is trivial using FPGAs and since HDMI and VGA have similar data designs for video, it is a good start.

A screen begins a new line when it receives a horizontal sync and a new frame on a vertical sync. The sync signals are part of blanking intervals [6]. Blanking intervals allow time for the electron gun in cathode ray tubes (CRTs) to move to the following line (horizontal retrace) or top of the screen (vertical retrace). Modern digital displays have retained the blanking intervals and repurposed them to transmit audio and other data. For this project, the active image area needs to be 1920x1080 for FHD. The SMPTE 274M standard was used [7] in which there are 2200 pixels per line and 1125 total lines. A representation of the on-screen and off-screen area can be seen in Figure 2.6. That means that there is a need for a 2200x1125 (pixels/frame) x60 (frames/sec) = 148.5 MHz clock. This clock is referred as the pixel clock. As explained in the previous section, a clock 5 times faster than the pixel clock is needed for the serialization of the output data. This clock will be in this case 742.5 MHz. How to generate these clocks will be discussed in subsubsection 2.2.2. Since there are 10 bits per pixel, the total required bandwidth for each line is approximately 1.49 Gbps.

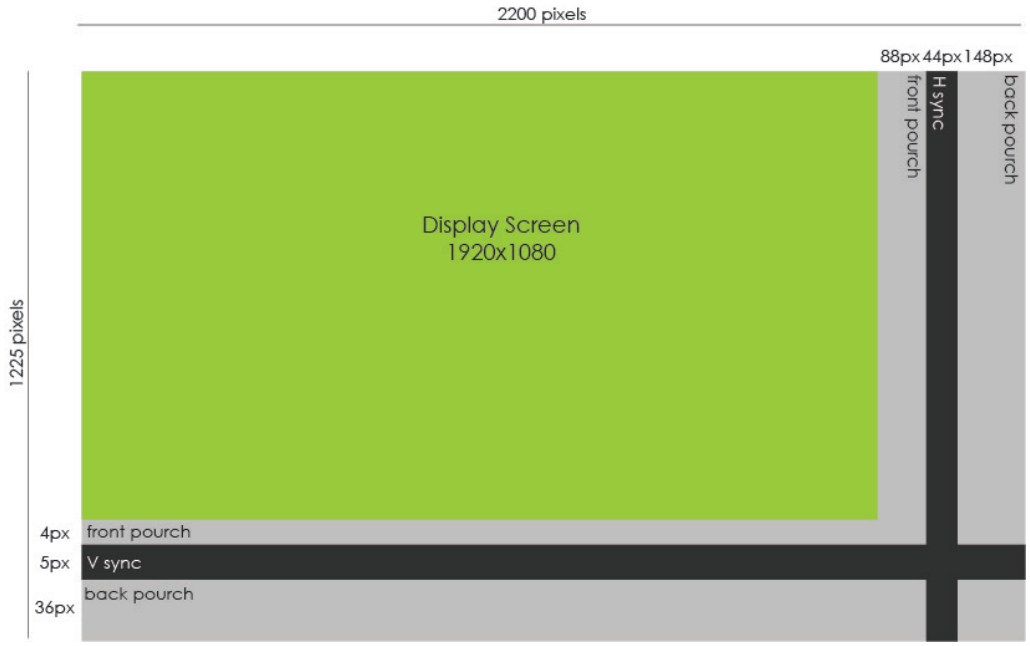

Figure 2.6: On-screen and off-screen area for 1080p VGA and pixel coordinates

In this project, the generation of the control signals and the coloring of the individual pixels are handled by two different VHDL components. The first one, generates the image progressively. For every on-screen pixel, the blank, VSYNC and HSYNC signals are set to 0. For every pixel outside of the on-screen are, blank is equal to 1 and HSYNC/VSYNC is set to either 1 or 0 respectively depending if the pixel falls inside the HSYNC/VSYNC area or not. The VHDL process inside of said module that sets the control signals can be seen in the Figure 2.7. Once the horizontal limit is reached, the next line is starting to be drawn and once the vertical limits of line is reached, the line counter resets to zero.

```
clk_proc: process(clk)
    begin
        if rising_edge(clk) then
            if x = 1920-1 then
                blank <= '1';
            elsif x = 2200-1 and (y < 1080-1 or y = 1125-1) then
                blank <= '0';
            end if;

            if x = 1920+88-1 then
                hsync <= '1';
            elsif x = 1920+88+44-1 then
                hsync <= '0';
            end if;

            if x = 2200-1 then
                x <= (others => '0');

                if y = 1080+4-1 then
                    vsync  <= '1';
                elsif y = 1080+4+5-1 then
                    vsync  <= '0';
                end if;

                if y = 1125-1 then
                    y <= (others => '0');
                else
                    y <= y +1;
                end if;
            else
                x <= x + 1;
            end if;
        cox<=x;
        coy <=y;
        end if;

    end process;
}
```

Figure 2.7: Setting of control signals with respect to pixel position

The x,y,cox,coy are 12-bit signals that represent the position of each pixel. They are 12-bit in order to be able to represent positive integers up to 2200. The first pixel of the active screen is in position (x,y) = (0,0) and the last one in position (x,y) = (1920,1080). The cox,coy signals are output of this module and are used as input to the vga_output module, in order for the active pixels to get properly colored. The next subsection describes the vga_output module.

2.2.1 VGA output

The elaborated design of the vga_output module can be seen in Figure 2.8

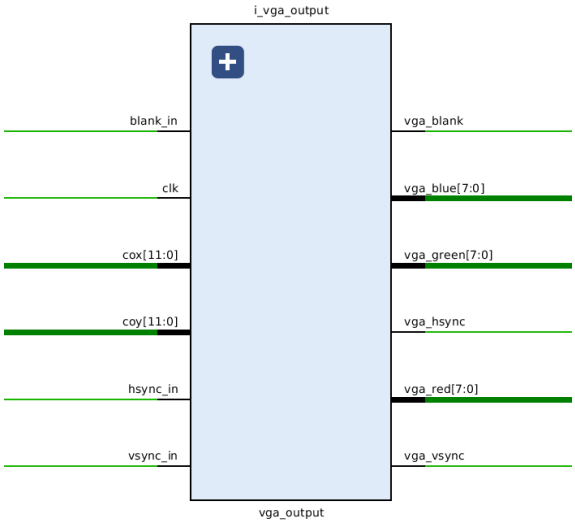

Figure 2.8: Vga_output module in Vivado

The output color signals are 8-bit words and correspond to the pixel intensity for the given color (0-255) in integer value. If blank is low, then the coordinates of the pixel falls inside the active area, and it will have the color values assigned inside the module. If blank is high, then all color bits are set to 0, since it is a pixel outside the active area.
The drawing of selected pixels or areas is achieved through if statements that check the position of the pixel and assign the color pixels accordingly. The mapping of the coordinates can be seen in **??**. To pick any desirable color in the RGB pallete, it is possible to use an online tool like [**pallete**], convert the integer values for the red,green and blue channel into binary and insert them into the VHDL code. For example, if the intention is to create a rectangular with its vertices in (x,y) = (150,15) , (150,50), (600,15), (600,50) and its color (R,G,B) = (250,5,5) while the rest of the screen has the color (R,G,B) = (5,5,250) the following code Figure 2.9 is inserted in the module. The resulted image on the screen as captured can be seen in Figure 2.10. Vivado gives a warning because the timing of the design is faster than the specs the manufacturer of the board gives but in reality the design works fine.

```
clk_proc: process(clk)
    begin
         if rising_edge(clk) then
            vga_hsync <= hsync_in;
            vga_vsync <= vsync_in;
            if blank_in = '0' then
            if cox > 149 and cox <601 and (coy >14 and coy<51) then
                vga_red <= "11111010";
                vga_green   <= "00000101";
                vga_blue  <= "00000101";      --RGB (250,5,5)
                vga_blank <= '0';
            else
               vga_blue <= "11111010";
                vga_red   <= "00000101";
                vga_green  <= "00000101"; -- RGB (5,5,250)
                vga_blank <= '0';

            end if;
            else
                vga_red    <= (others => '0');
                vga_green  <= (others => '0');
                vga_blue   <= (others => '0');
                vga_blank <= '1';
            end if;
        end if;

    end process;
}
```

Figure 2.9: VHDL code for generating the image Figure 2.10.

Figure 2.10: Image from CMOD A7 as captured from the HDMI output.

Since now it is possible to display any desirable image, in the section 3, the generation of the

graphics for the pong game are discussed. In the next subsection, the usage of the MMCM primitive is discussed for the generation of the desired clocks.

2.2.2 Clock generation

In 7 series FPGAs, the clock management tile (CMT) includes a mixed-mode clock manager (MMCM) and a phase-locked loop (PLL) [8]. PLL in 7 series is just a limited version of MMCM. MMCM primitive is used for all clock generations. A detailed block diagram for the MMCM can be seen in Figure 2.11.

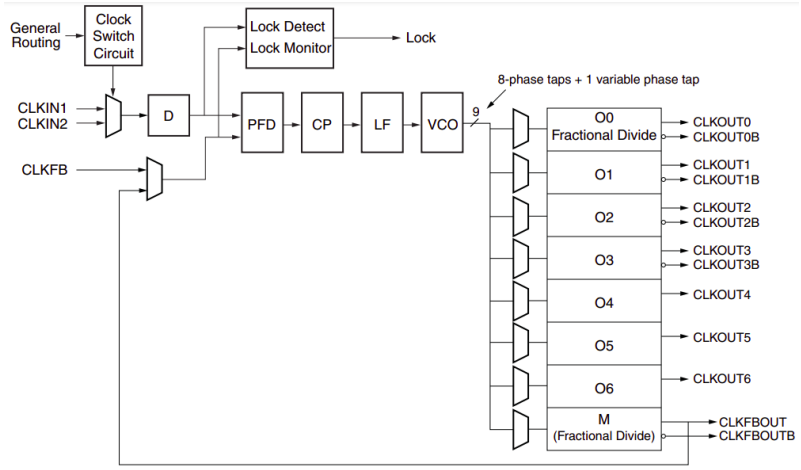

Figure 2.11: MMCM block diagram [8].

There are are two MMCM primitives - Base and Advanced. Base offers access to the most frequently used features such as clock deskew, frequency synthesis, coarse phase shifting, and duty cycle programming. This primitive is used in this project. A table with a description of the input/output ports can be seen in Figure 2.12

Description	Ports
Clock Input	CLKIN1, CLKFBIN
Control Inputs	RST
Clock Output	CLKOUT0 to CLKOUT6, CLKOUT0B to CLKOUT3B, CLKFBOUT, and CLKFBOUTB
Status and Data Outputs	LOCKED
Power Control	PWRDWN

Figure 2.12: Inputs/Output ports of the base MMCM primitive[8].

There are two settings that are applied for all output clocks. These are the CLKFBOUT_MULT_F value (M) and the DIVCLK_DIVIDE value (D). The resolution of M is 0.125. These two numbers decide the Voltage Controlled Oscillator (VCO) frequency which is the same for all output clocks. The equation for Fvco is :

$$Fvco = Fclkin \cdot M/D, \tag{2.1}$$

where Fclkin is the frequency of the input clock. Fvco must fall in a specified range for each FPGA. For the CMOD A7, that range is 600 Mhz - 1200 Mhz [9].

The frequency of each output clock is given by the equation:

$$Fclkout = Fvco/O, \tag{2.2}$$

where O is the value of the CLKOUT_DIVIDE setting for each output clock. For clock0, that number can also be functional with a resoluton of 0.125.
To achieve the clock frequency of exactly five times the pixel clock - 5*148.5 MHz = 742.5 MHz - with an input clock of 24 MHz, M = 61.875 and D = 2. The O value for this output clock is 1, while the O value for the pixel clock is 5. An excerpt of the MMCM base primitive instance can be seen in Figure 2.13.

```
MMCME2_BASE_inst : MMCME2_BASE
   generic map (
      BANDWIDTH => "OPTIMIZED",  -- Jitter programming (OPTIMIZED, HIGH, LOW)
      DIVCLK_DIVIDE   => 2,     -- Master division value (1-106)
      CLKFBOUT_MULT_F => 61.875,-- Multiply value for all CLKOUT (2.000-64.000).
      CLKFBOUT_PHASE => 0.0,
      -- Phase offset in degrees of CLKFB (-360.000-360.000).
      CLKIN1_PERIOD => 41.6666667,      -- Input clock period
      --in ns to ps resolution (i.e. 33.333 is 30 MHz).
      -- CLKOUT0_DIVIDE - CLKOUT6_DIVIDE: Divide amount for each CLKOUT (1-128)
      CLKOUT0_DIVIDE_F => 5.0,    -- Divide amount for CLKOUT0 (1.000-128.000).
      CLKOUT1_DIVIDE   => 5,
      CLKOUT2_DIVIDE   => 1,
      CLKOUT3_DIVIDE   => 1,
      CLKOUT4_DIVIDE   => 1,
      CLKOUT5_DIVIDE   => 1
}
```

Figure 2.13: MMCM base primitive instance excerpt.

3 Graphics design

This section discusses the generation of the graphics and the gameplay for the pong game.

Since now it is possible to generate any image via the vga_output module, in this subsection the design of the game graphical elements will be discussed in detail.

The concept graphics of the game can be seen in Figure 3.1 The game is happening in the 1400px * 700 px rectangular area. The winner is the one who scores first 3 points. If a bar score is filled with the color of the player's paddle, then a point has been scored.

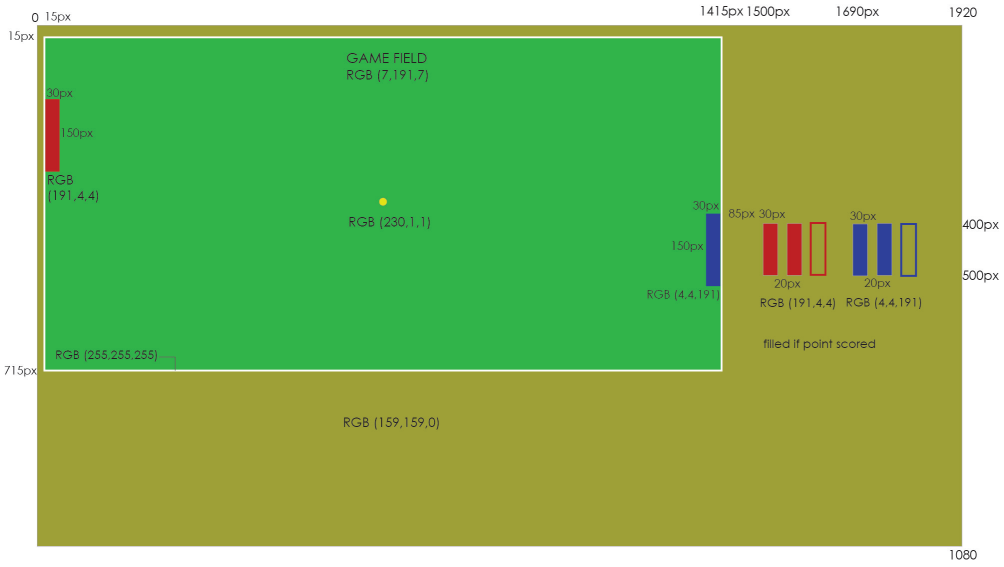

Figure 3.1: Concept art for pong game.

3.1 Game field and paddles

To avoid writing the same code over and over again, two new modules are created. The first one is called draw_rec and it serves a double purpose. An input signal p is used to select its operation. If this is equal to 0, the module's purpose is to create the outline of a rectangle with the position of its vertices being input signals (x_s,x_f,y_s,y_f), 12 bit wide each, in the module. Cox and coy are also inputs. The process inside the module checks to see if the cox,coy of the current pixel belong to the perimeter of the rectangle. If this is true, the output signal draw_rec is set to 1, otherwise it is 0. If p=1, the x_f,y_f inputs are irrelevant and the purpose of the instance of the module is to create the perimeter of the rectangular paddle. The logic behind the drawing is the same, but since the choice was for the paddles to have a fixed size of 30px * 150px only one point is necessary to draw it. If a given cox,coy belong to the perimeter of the paddle, then draw_rec=1, otherwise it is 0. The VHDL code for the module can be seen in Figure 3.2.

The second module is called draw_recfilled and it serves the same functionality as the one before with the difference that it draws the inside of the rectangle. That means that its output 1-bit signal is high only when the current pixel is in the inside are of the random rectangular or paddle.

The selection signal p serves the same functionality as before.

Using multiple instances of this module, it is possible to create now the game field and paddle. An excerpt of the VHDL instantiation of the draw_rect component as used in the project can be seen in Figure 3.3

```
begin
draw : process(cox)
begin
    if p = '0' then
    if (cox=x_s or cox=x_f ) and coy>y_s-1 and coy<y_f+1 then
                        draw_rec <= '1';
    elsif (coy=y_s or coy=y_f ) and cox>x_s and cox<x_f then
                        draw_rec <= '1';
    else
                        draw_rec <='0';
    end if;

    elsif p = '1' then
    if (cox=x_s or cox=x_s+30 ) and coy>y_s-1 and coy<y_s+149 then
                     draw_rec <= '1';
    elsif (coy=y_s or coy=y_s+150 ) and cox>x_s and cox<x_s+30 then

                     draw_rec <= '1';
    else
                     draw_rec <='0';


    end if;
    end if;
}
```

Figure 3.2: VHDL code for process inside of draw_rec module.

```
i_draw_rect: draw_rect port map (  -- draw game boundaries
            x_s => "000000001111",
            y_s => "000000001111",
            x_f => "010110000111",
            y_f => "001011001011",
            cox => cox,
            coy => coy,
            p => '0',
            draw_rec => rec

);
```

Figure 3.3: draw_rec component instantiation.

3.2 Ball design

Except from the paddles, a ball is also required for the pong game. To make the game more visually pleasing, the ball should be round. The concept design of the ball can be seen in Figure 3.4. A module that takes as input the signals cox,coy and Xo,Yo is created to draw the ball. The signals Xo,Yo incorporate the information about the current position of the ball which is periodically updated in the top module as explained in the next section. With if statements, it is possible to create the whole ball from just this point.

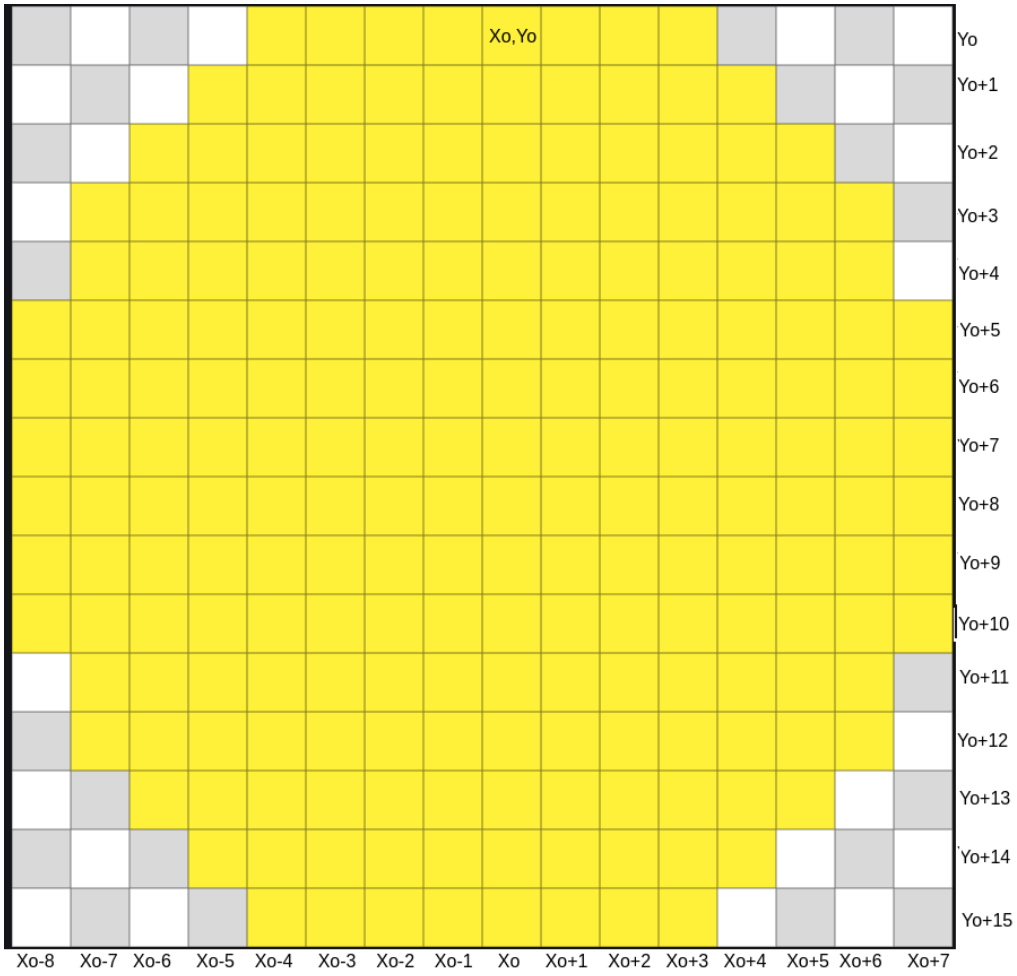

Figure 3.4: Ball design concept art. Only the coordinates of one point is needed to draw the ball.

If at this moment, cox and coy are inside the ball area, the output signal of the module is high. Otherwise, it is low. The VHDL code for the discussed module can be seen in Figure 3.5.

```
begin
draw : process(cox)
begin
    if   (cox=x or cox=x-1) and (coy>y-1 and coy<y+16) then
      draw_ball <= '1';

    elsif   (cox=x-3 or cox=x-2 or cox=x+2 or cox=x+1)
                          and ((coy>y-1 and coy<y+16)) then

      draw_ball <= '1';
    elsif   (cox=x-4 or cox=x-5 or cox=x+3 or cox=x+4)
                          and ((coy>y and coy<y+15)) then

      draw_ball <= '1';

    elsif   (cox=x-6 or cox=x+5 ) and ((coy>y+1 and
coy<y+14)) then

      draw_ball <= '1';
    elsif   (cox=x-7 or cox=x+6 ) and ((coy>y+2 and
coy<y+13)) then

      draw_ball <= '1';

    elsif   (cox=x-8 or cox=x+7 ) and (coy>y+4 and coy<y+11) then

      draw_ball <= '1';
    else
      draw_ball <= '0';
    end if;
end process;
```

Figure 3.5: Process inside the draw_ball module.

Since there is only one ball, only one instance of this module is needed.

3.3 Score board

The final touch in the graphics for the game is a score board. The winner is the player that scores first 3 points. The score board for each player is three parallel rectangles. If a point hasn't been scored yet then the inside color of these rectangles is the same as for the rest of the screen. If the point has been scored, the rectangle is filled with the same color as the color of the paddle of the player who scored the point. A module is created to handle the process. Two instances of this module are used since there is one score board for each player. The inputs of this module are cox, coy as usual, x - which is the first pixel of the drawing for the score board and score. Score is a two bit signal that keeps track of the score. The output of this module is one-bit signal named draw_sc and is high only when cox,coy are in the range of the score board range of pixels. The VHDL code for the process inside the module can be seen in Figure 3.6

```
begin
draw : process(cox)
begin
    if (cox=x or cox=x+30 or cox=x+50 or cox=x+80
       or cox = x+100 or cox = x+130) and (coy>400 and coy<500) then

                    draw_sc <= '1';

     elsif (coy=400 or coy=500 ) and ((cox>x+99 and cox<x+131)
       or (cox>x+49 and cox<x+81) or (cox>x-1 and cox<x+31)) then
                    draw_sc <= '1';

     elsif score = "01" and
               (coy> 400 and coy <500 and cox > x and cox < x+30) then
                    draw_sc <= '1';

     elsif score = "10" and (coy> 400 and coy <500 and
               ((cox > x and cox < x+30) or (cox>x+50 and cox<x+80))) then
                    draw_sc <= '1';

     elsif score = "11" and (coy> 400 and coy <500 and ((cox > x and cox < x+30)
              or (cox>x+50 and cox<x+80) or (cox>x+100 and cox<x+130))) then

                    draw_sc <= '1';


     else
                    draw_sc <='0';
   -- end if;
    end if;
end process;
end Behavioral;
```

Figure 3.6: Process inside the draw_score module.

3.4 Combining everything together

Since now all needed graphical elements can be drawn, using instances of their appropriate module, it is time to tie everything together. This happens in the vga_output module. The output signals of the instances are used to decide what should be draw on the current pixel. That is realized with if statements inside a process. For a example if the ball signal is high, the pixel is colored with a yellow shade color by setting its RGB signals correspondingly and blank is 0 - since it has to be in the active area.
The if statement must have the right hierarchy. For example, the process must check first if it is a ball pixel and then if it a field pixel in order for everything to appear properly. The whole VHDL process can be seen in Figure 3.4

```
    begin
        if rising_edge(clk) then
            vga_hsync <= hsync_in;
            vga_vsync <= vsync_in;
            if blank_in = '0' then
                if draw_bal='1' then -- ball flag
                    vga_red <= "11100110" ;
                    vga_green <= "11100110" ;
                    vga_blue <= "00000001" ;

                    vga_blank <= '0' ;

                elsif draw_sc(0) = '1' then -- score one flag
                    vga_red <= "10111111" ;
                    vga_green <= "00000111" ;
                    vga_blue <= "00000111";
                    vga_blank <= '0' ;
                    vga_blank <= '0' ;
                elsif draw_sc(1) = '1' then  -- score two flag
                    vga_red <= "00000100" ;
                    vga_green <= "00000100" ;
                    vga_blue <= "10111111";
                    vga_blank <= '0' ;
                    vga_blank <= '0' ;
                elsif paddle(0) = '1' or paddle(1)='1' then -- paddle_1 flag
                    vga_red <= "10111111" ;
                    vga_green <= "00000111" ;
                    vga_blue <= "00000111";
                    vga_blank <= '0' ;
                elsif paddle(2) = '1' or paddle(3)='1' then -- paddle_2 flag
                    vga_red <= "00000100" ;
                    vga_green <= "00000100" ;
                    vga_blue <= "10111111";
                    vga_blank <= '0' ;
                elsif rec = '1' then          -- boundaries flag
                    vga_red <= "10111111" ;
                    vga_green <= "10111111" ;
                    vga_blue <= "10111111";
                    vga_blank <= '0' ;
               elsif recfilled='1' then
-- field filling flag
                    vga_red <= "00010001" ;
                        vga_green <= "00101111"  ;
                        vga_blue <= "00000001" ;
                    vga_blank <= '0' ;
               else
                        vga_red    <= "10011111";
                        vga_green  <= "10011111";
                        vga_blue <= "00000001" ;
                        vga_blank <= '0' ;

             end if;
```

Figure 3.7: Process inside the vga_output module.

Two images captured from the HDMI output of the CMOD A7 can be seen in Figure 3.8 and in Figure 3.9. The first one is the initial screen of the game. In the second one, the blue player has won.

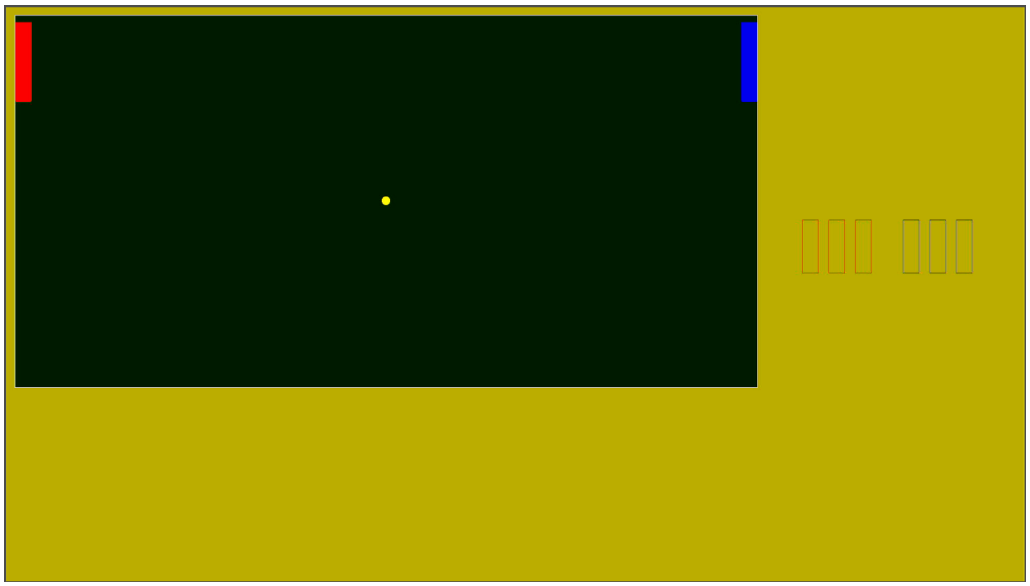

Figure 3.8: Initial screen of the game as captured from the HDMI output.

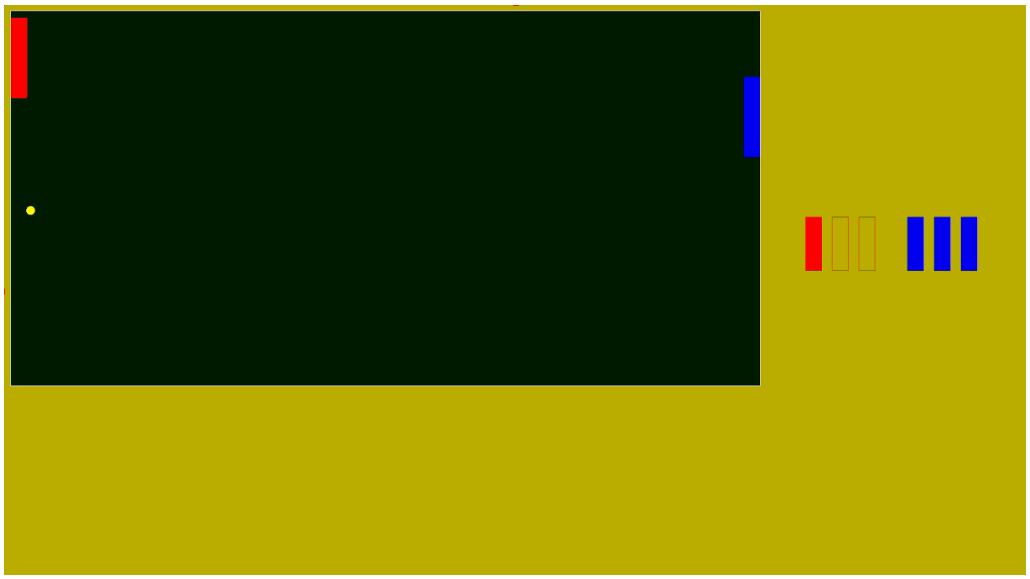

Figure 3.9: Blue player has won 3-1. Screen as captured from the HDMI output

4 Gameplay design

In this section, the gameplay design will be discussed. In the first subsection the ball movement will be discussed.
All the logic in this section is written into the top module of the code.

4.1 Ball movement

The ball movement has four different modes. This is represented by a 2-bit signal. The states and their 2-bit representation can be seen in Table 4.1

Ball Movement	**Binary Representation**	**Current position**	**Next position**
right & down	00	Xb,Yb	Xb-1,Yb+1
right & up	01	Xb,Yb	Xb-1,Yb-1
left & down	10	Xb,Yb	Xb+1,Yb+1
left & up	11	Xb,Yb	Xb+1,Yb-1

Table 4.1: Ball movement when no event occurs.

The position of the ball is encoded in two 12-bit signals, xbal and ybal. The coordinates of the position of the ball are updated with a selected frequency of 200 Hz. This allows for smooth ball movement on the screen and challenging gameplay. At first a 11 MHz clock is generated by the MMCM and then with a counter that counts up to 54999, the 200 Hz clock is obtained. When the game starts the ball is approximately in the center of the game field. Once the game is started, the ball is going right and down. The ball does not change state unless a synchronous event happens. The event is either the ball hitting the upper or low boundary of the game field or hitting one of the paddles or scoring a point. In Table 4.2, the list of events and the change of mode of ball movement can be seen. A signal named state is also present. If this signal is 0, then the ball is not moving at all.
If statements are used to decide if an event occurs. The dimensions of the graphical elements have to be taken into consideration. For example, to check if the ball hits the left paddle, the process checks if the x position of the ball is X(start of game field) plus width of the paddle. Then, if the y coordinate of the ball plus the height of the ball falls inside the y of the paddle plus its height, the ball is bounced back. Otherwise, the opponent scores a point. This example can be seen as written in VHDL code in Figure 4.1
Moving the paddles is achieved through the UART interface. This will be discussed in detail, in the upcoming section. When the proper byte is received from the UART, the y position of the paddle is updated. This update is either an increment or a decrement by 5 pixels of the y paddle position.

Ball Mode	Event	Next Ball Mode
right & down	hit low boundary	right & up
right & down	hit right paddle	left & down
right & up	hit upper boundary	right & down
right & up	hit right paddle	left & up
left & down	hit low boundary	left & up
left & down	hit left paddle	right & down
left & up	hit high boundary	right & up
left & up	hit left paddle	right and up
xxxxxxxxx	left player scored - no match point	left and down
xxxxxxxxx	right player scored - no match point	right and down
xxxxxxxxx	one player scored match point	ball freezes
ball freezed	game starts - ball is reset to field center	right and down

Table 4.2: How the ball movement mode changes at each event

```
    --excerpt
elsif xbal = 1415-7-30 then
            if ybal+15>yb_pad-1 and ybal<yb_pad+151 then
               if rise="01" then --ball hitting right paddle going up
                xbal <= xbal-1;
                ybal <= ybal-1;
                rise <= "11"; --ball is going right-up

               elsif rise ="00" then
                xbal <= xbal+1;
                ybal <= ybal-1;
                rise <= "10"; --ball is going right down
               end if;
            elsif sc_a ="00" or sc_a = "01" then
                sc_a <= std_logic_vector(unsigned(sc_a)+1);
                xbal <= "001011001011";
                ybal<= "000101100100";
                rise <="10";
            elsif sc_a ="10" then
                sc_a <= std_logic_vector(unsigned(sc_a)+1);

            end if;
```

Figure 4.1: The process checks if the ball hit the left paddle.

4.2 UART implementation - Receiver

To transfer data from the PC to the FPGA. the UART interface is implemented into the design. The PC serializes the data - a byte- and send it through the USB port into the FPGA. CMOD A7 has a dedicated port, for the receiving part of the UART which is connected to the micro USB input of the A7. This port is J17 and is included in the constraints file of the project [10]. In the FPGA, the byte is de-serialized. The UART can have a lot of different configurations of its settings. For this project, the baud rate is set to 3000000 with 1 stop bit, no parity bit and no flow control.

A new module is created to handle the process of receiving a byte throught the UART. The input

of this module is a clock and the receiving bits from the RX pin, while, the output is the 8-bit data and a flag signal. A finite state machine inside of said module, keeps track of the states and their transitions. The states are idle, start_bit, data_rec, stop_bit and cleanup. When idle, a constant stream of 1s is received. Once, a 0 is received, this is the start bit that let the UART receiver know that data will be sent. The next 8-bits are the data byte being sent from the PC starting with the least significant bit. The last bit is the stop bit and it is 1. After the stop bit is over, the output signal flag is driven high for one clock cycle to signify that the incoming byte is ready to be read. The FSM's state then goes to cleanup for one clock cycle to drive the flag low again and then to idle again. A graphical representation of the incoming bits can be seen in Figure 4.2 and of the FSM in Figure 4.3

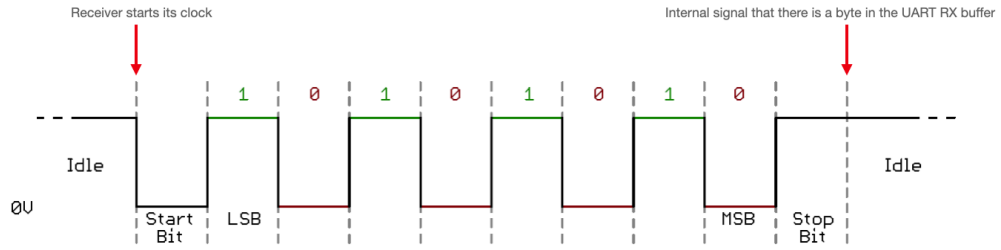

Figure 4.2: Graphical representation of the receiving bits [11].

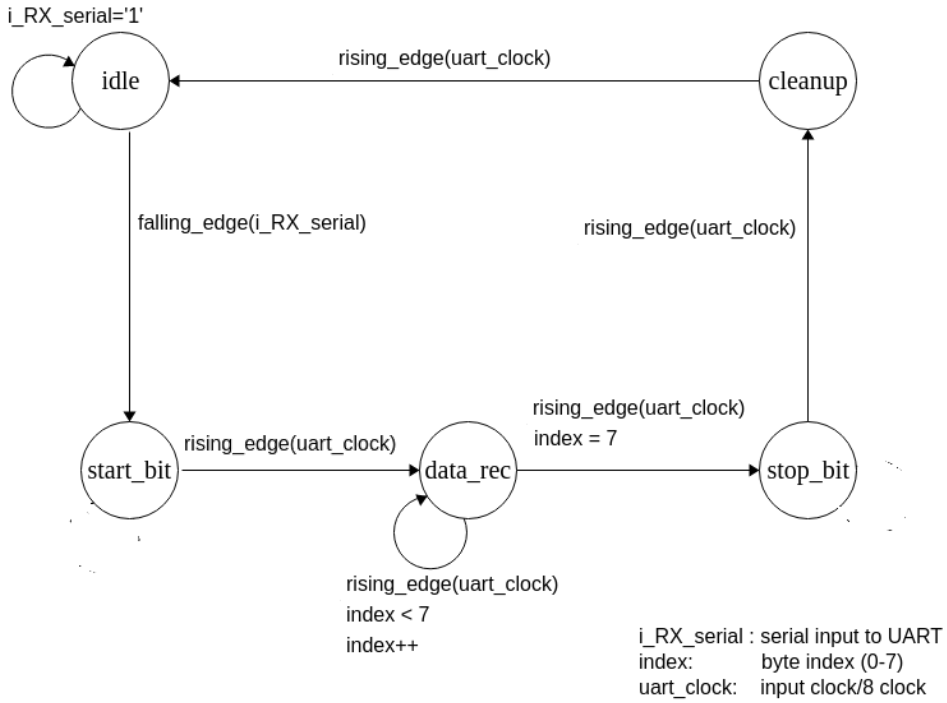

Figure 4.3: Graphical representation of the FSM.

To ensure that the data are not corrupted, the sampling of all bit happens at the middle of the

bit. For a baud rate of 3000000, a 3KHz sampling frequency is required. Using the 24 MHz clock as input to the module and a 8-step counter, the data are sampled after 4 steps on the counter. A simulation of the module can be seen in Figure 4.4

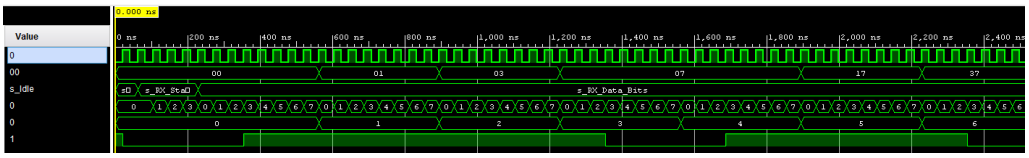

Figure 4.4: Simulation of the UART receiver module.

From the top level, an instance of the module is initiated. A new process is created. In its sensitivity list there is the flag output of the UART receiver module. Once its edge is rising, the data byte is read. Each byte corresponds to a different action. A table of the incoming data byte and its corresponding action can be seen in Table 4.3

Incoming Byte in Hex	**Action**
00	left paddle down
a0	left paddle up
80	right paddle down
90	right paddle up
ff	start game
0f	pause game
01	restart game

Table 4.3: UART commands

With the receiving part of the UART implemented, the game is now complete. Once the bitstream is generated and the FPGA connected to the PC via the micro USB cable, the UART can be tested.
In Linux environment, the communication can be established via the terminal. In order to do that, first the following command must be written, in which ttyUSB1 signifies in which port of the PC the FPGA is connected and 3000000 is the baud rate:

```
stty -F/dev/ttyUSB1 3000000
```

After that, data can be send with the following command:

```
echo -en '\x80' > /dev/ttyUSB1
```

In this format, the data are represented by 80 and x indicates that this is in hexadecimal representation. This moves the right paddle down. To write this command every time the player wants to move the paddle is tedious. A bash script is written that offers a simpler way of playing the game.

4.3 Bash script

Since it is a bash script, the first line is the shebang along with the relative directory of the bash shell on the PC used. After that, the connection is initiated as shown in the previous subsection. Then, with an echo command the player is informed about the game controls. Using an endless

while loop and a read command, the script reads the keystrokes of the player. If the key pressed corresponds to an action, the appropriate byte is send as shown in the previous section. To listen for keystrokes, the read command is used along with the parameters r,s,n1, which tell the program to expect a raw, one character long character without showing the character to the screen. The bash script can be seen in Figure 4.5

```
1
2 #! bin/bash
3 stty -F/dev/ttyUSB1 3000000
4
5 echo "Choose action -play(1) -pause(2) -restart(3) -leftpaddledown(a) -leftpaddleup
      (q) -rightpaddledown(s) -rightpaddleup(w)"
6
7
8 while true; do
9 read -rsn1 input
10 if [ "$input" = "s" ]; then
11     echo -en '\x80' > /dev/ttyUSB1
12 elif [ "$input" = "w" ]; then
13     echo -en '\x90' > /dev/ttyUSB1
14 elif [ "$input" = "q" ]; then
15     echo -en '\xa0' > /dev/ttyUSB1
16 elif [ "$input" = "a" ]; then
17     echo -en '\x00' > /dev/ttyUSB1
18 elif [ "$input" = "1" ]; then
19     echo -en '\xff' > /dev/ttyUSB1
20 elif [ "$input" = "2" ]; then
21     echo -en '\x0f' > /dev/ttyUSB1
22 elif [ "$input" = "3" ]; then
23     echo -en '\x01' > /dev/ttyUSB1
24 fi
25 done
```

Figure 4.5: Process inside the vga_output module.

4.4 UART - Transmitter

The transmitter part of the UART was also implemented, though it serves no functional purpose for the play-ability of the game. The pin for the transmitter on the CMOD A7 is J18 and must be included in the constraint file. A similar module as for the receiver is created for the transmitter. A FSM also handles the process. The states are exactly the same as for the receiver, except for the data_rec one which is now data_trans. The inputs of the module are the clock, the byte to transmit and a signal that driven high it takes the FSM from idle state to start_bit state, named i_TX. The output are the serial bit to be sent connected at the top level with the UART output pin, a signal that is high when the process of sending something is active and a flag signal that is driven high for one clock cycle, once the operation is completed. The baud rate is 3000000. The input clock is 24 MHz and it takes 8 clock periods to send one bit. The start bit is always zero, the stop bit 1 and when the uart is not transmitting data, it transmits a constant string of 1s. In the following Figure 4.6, a simulation of the module can be seen. The hex number 6f is being serialized and sent starting with the LSB.

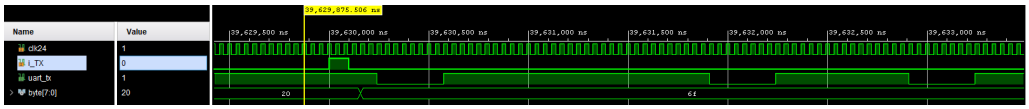

Figure 4.6: Simulation of the UART module. Hex 6f is serialized and sent to the module output.

Since the module was tested and worked fine, a purpose was given to it. When a player wins a game, the PC terminal should receive the appropriate bytes in order to display the message "game over!" , over and over again. A website like this [12] can be used to see which ASCII character correspond to which byte.

Another module, named UART_word, was created to handle this process. The inputs to this module is a signal called enable and the 24MHz clock. A FSM is written inside this module. It has two states - idle and run. In the first state, the UART instance does not sent any data and uart_tx = '1'. When the game is over, enable is driven high and causes the state change from idle to run. In this state, the bytes corresponding to each ASCII character are fed sequentially into the transmitter and the signal i_TX is driven high. i_RX must be driven high with a frequency of 279 or lower. In practice, a 200 Mhz clock was used, created by a 120 steps counter for convenience. An index keeps track of how many characters have been sent. The process stops when the game is restarted. The output in the PC terminal can be seen in Figure 4.7

Figure 4.7: Terminal output when the game is over.

5 Conclusion

This report describes the process of creating an HDMI signal and a playable version of the infamous pong game using CMOD A7-15T fpga. The project utilizes a lot of the on-chip resources of the A7.
No significant setbacks were faced during the development of the project. It was possible for the FPGA to drive a FHD HDMI signal at 60Hz. A lot of improvements can be made in the gameplay design like more complex ball movement but there is no difficulty to that, just more complex logic. All VHDL code can be seen in the appendices.

6 Bibliography

[1] Colin Riley. *HDMI over Pmod using the Arty Spartan 7 FPGA board*. URL: https://domipheus.com/blog/hdmi-over-pmod-using-the-arty-spartan-7-fpga-board/. (accessed: 11.08.2022).

[2] Digital Display Working Group. *Digital Visual Interface - DVI*. URL: https://glenwing.github.io/docs/DVI-1.0.pdf. (accessed: 11.08.2022).

[3] Xilinx. *7 Series FPGAs SelectIO Resources User Guide (UG471)*. URL: https://docs.xilinx.com/v/u/en-US/ug471_7Series_SelectIO. (accessed: 11.08.2022).

[4] Silicon Image. *Digital Visual Interface TMDS Extensions, WHITE PAPER*. URL: https://www.fpga4fun.com/files/WP_TMDS.pdf. (accessed: 11.08.2022).

[5] Xilinx. *UltraScale Architecture Libraries Guide (UG974)*. URL: https://docs.xilinx.com/r/2021.1-English/ug974-vivado-ultrascale-libraries/OBUFDS. (accessed: 11.08.2022).

[6] Nathan Ickes. *VGA Video*. URL: https://web.mit.edu/6.111/www/s2004/NEWKIT/vga.shtml. (accessed: 11.08.2021).

[7] SMPTE. *PROPOSED SMPTE 274M SMPTE STANDARD for Television —- 1920 × 1080 Scanning and Analog and Parallel Digital Interfaces for Multiple Picture Rates*. URL: http://car.france3.mars.free.fr/HD/INA-%2026%20jan%2006/SMPTE%20normes%20et%20confs/s274m.pdf. (accessed: 11.08.2022).

[8] Xilinx. *7 Series FPGAs Clocking Resources User Guide (UG472)*. URL: https://docs.xilinx.com/v/u/en-US/ug472_7Series_Clocking. (accessed: 11.08.2022).

[9] Xilinx. *Artix-7 FPGAs Data Sheet: DC and AC Switching Characteristics (DS181)*. URL: https://docs.xilinx.com/v/u/en-US/ds181_Artix_7_Data_Sheet. (accessed: 11.08.2022).

[10] Xilinx. *CMOD A7 schemantics*. URL: https://digilent.com/reference/_media/reference/programmable-logic/cmod-a7/cmod_a7_sch_rev_c0.pdf. (accessed: 11.08.2022).

[11] V. Hunter Adams. *Universal Asynchronous Receiver Transmitter (UART)*. URL: https://vanhunteradams.com/Protocols/UART/UART.html. (accessed: 11.08.2022).

[12] rapidtables.com. *Hex to ASCII Text String Converter*. URL: https://www.rapidtables.com/convert/number/hex-to-ascii.html. (accessed: 11.08.2022).

Appendices

A VHDL code

A.1 top.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

Library UNISIM;
use UNISIM.vcomponents.all;

entity top_level is
    Port (

        clk24    : in STD_LOGIC;
        uart_rx : in  std_logic;
        uart_tx: out std_logic;
        hdmi_tx_clk_p : out std_logic;
        hdmi_tx_clk_n : out std_logic;
        hdmi_tx_p     : out std_logic_vector(2 downto 0);
        hdmi_tx_n     : out std_logic_vector(2 downto 0)
);
end top_level;

architecture Behavioral of top_level is
    signal clk_pixel_x1  : std_logic;
    signal clk_pixel_x5  : std_logic;
    signal clk_b  : std_logic;
    signal xbal: unsigned(11 downto 0) := "001011001011";
    signal ybal: unsigned(11 downto 0):= "000101100100";
    signal ya_pad: unsigned(11 downto 0):= "000000011100";
    signal yb_pad: unsigned(11 downto 0):= "000000011100";
    signal blank : std_logic := '0';
    signal hsync : std_logic := '0';
    signal vsync : std_logic := '0';
    signal c: std_logic;
    signal state: std_logic:='0';
    signal clk_u: std_logic;
    signal w_RX_DV     : std_logic:='0';
    signal w_RX_Byte   : std_logic_vector(7 downto 0);
```

```
signal sc_a: std_logic_vector(1 downto 0):= "00" ; --score of player one
signal sc_b: std_logic_vector(1 downto 0):= "00" ; --score of player two

signal flag: std_logic:='0';

component vga_gen_1080p is
    port (
       clk        : in  std_logic;
       cox:out unsigned(11 downto 0);
       coy:out unsigned(11 downto 0);
       blank      : out std_logic;
       hsync      : out std_logic;
       vsync      : out std_logic
       );
end component;

component vga_output is
    Port ( clk : in STD_LOGIC;
           score_a: in std_logic_vector(1 downto 0);
           score_b: in std_logic_vector(1 downto 0);
           hsync_in : in STD_LOGIC;
           vsync_in : in STD_LOGIC;
           blank_in : in STD_LOGIC;
           cox:in unsigned(11 downto 0);
           coy:in unsigned(11 downto 0);
           xball:in unsigned(11 downto 0);
           yball:in unsigned(11 downto 0);
           ya_paddle: in unsigned(11 downto 0);
           yb_paddle: in unsigned(11 downto 0);

           vga_hsync : out std_logic;
           vga_vsync : out std_logic;
           vga_red   : out std_logic_vector(7 downto 0);
           vga_green : out std_logic_vector(7 downto 0);
           vga_blue  : out std_logic_vector(7 downto 0);
           vga_blank : out std_logic);
end component;

signal count         : std_logic_vector(20 downto 0);
signal cox: unsigned(11 downto 0);
signal coy: unsigned(11 downto 0);
signal rise: std_logic_vector(1 downto 0) := "00";
signal tx: std_logic:='0';
```

```
    signal vga_hsync      : std_logic;
    signal vga_vsync      : std_logic;
    signal vga_red        : std_logic_vector(7 downto 0);
    signal vga_green      : std_logic_vector(7 downto 0);
    signal vga_blue       : std_logic_vector(7 downto 0);
    signal vga_blank      : std_logic;
    signal reset          : std_logic;




    component vga_to_hdmi is
        port ( pixel_clk    : in std_logic;
               pixel_clk_x5 : in std_logic;
               reset  : in std_logic;

               vga_hsync : in std_logic;
               vga_vsync : in std_logic;
               vga_red   : in std_logic_vector(7 downto 0);
               vga_green : in std_logic_vector(7 downto 0);
               vga_blue  : in std_logic_vector(7 downto 0);
               vga_blank : in std_logic;

               hdmi_tx_clk_p : out   std_logic;
               hdmi_tx_clk_n : out   std_logic;
               hdmi_tx_p     : out   std_logic_vector(2 downto 0);
               hdmi_tx_n     : out   std_logic_vector(2 downto 0)
               );
    end component;

    signal locked         : std_logic; -- to check if mmcm has locked
    signal clkfb          : std_logic;
begin
    reset <= not locked;

MMCME2_BASE_inst : MMCME2_BASE
   generic map (
      BANDWIDTH => "OPTIMIZED",  --
                                --Jitter programming (OPTIMIZED, HIGH, LOW)
      DIVCLK_DIVIDE   => 2,        -- Master division value (1-106)
      CLKFBOUT_MULT_F => 61.875,
                     -- Multiply value for all CLKOUT (2.000-64.000).
```

```
      CLKFBOUT_PHASE => 0.0,
               -- Phase offset in degrees of CLKFB (-360.000-360.000).
      CLKIN1_PERIOD => 41.6666667,
              -- Input clock period in ns to ps resolution (i.e. 33.333 is 30 MHz).
      -- CLKOUT0_DIVIDE - CLKOUT6_DIVIDE: Divide amount for each CLKOUT (1-128)
      CLKOUT0_DIVIDE_F => 67.5,    -- Divide amount for CLKOUT0 (1.000-128.000).
      CLKOUT1_DIVIDE   => 5,
      CLKOUT2_DIVIDE   => 1,
      CLKOUT3_DIVIDE   => 1,
      CLKOUT4_DIVIDE   => 1,
      CLKOUT5_DIVIDE   => 1,
      CLKOUT6_DIVIDE   => 128,
      -- CLKOUT0_DUTY_CYCLE
      -- CLKOUT6_DUTY_CYCLE: Duty cycle for each CLKOUT (0.01-0.99).
      CLKOUT0_DUTY_CYCLE => 0.5,
      CLKOUT1_DUTY_CYCLE => 0.5,
      CLKOUT2_DUTY_CYCLE => 0.5,
      CLKOUT3_DUTY_CYCLE => 0.5,
      CLKOUT4_DUTY_CYCLE => 0.5,
      CLKOUT5_DUTY_CYCLE => 0.5,
      CLKOUT6_DUTY_CYCLE => 0.5,
      -- CLKOUT0_PHASE
      -- CLKOUT6_PHASE: Phase offset for each CLKOUT (-360.000-360.000).
      CLKOUT0_PHASE => 0.0,
      CLKOUT1_PHASE => 0.0,
      CLKOUT2_PHASE => 0.0,
      CLKOUT3_PHASE => 0.0,
      CLKOUT4_PHASE => 0.0,
      CLKOUT5_PHASE => 0.0,
      CLKOUT6_PHASE => 0.0,
      CLKOUT4_CASCADE => FALSE,
      -- Cascade CLKOUT4 counter with CLKOUT6 (FALSE, TRUE)
      REF_JITTER1 => 0.0,
      -- Reference input jitter in UI (0.000-0.999).
      STARTUP_WAIT => FALSE
      -- Delays DONE until MMCM is locked (FALSE, TRUE)
   )
   port map (
      -- Clock Outputs: 1-bit (each) output: User configurable clock outputs
      CLKOUT0   => clk_b,      -- 1-bit output: CLKOUT0
      CLKOUT0B  => open,          -- 1-bit output: Inverted CLKOUT0
      CLKOUT1   => clk_pixel_x1, -- 1-bit output: CLKOUT1
      CLKOUT1B  => open,          -- 1-bit output: Inverted CLKOUT1
```

```
      CLKOUT2   => clk_pixel_x5, -- 1-bit output: CLKOUT2
      CLKOUT2B  => open,         -- 1-bit output: Inverted CLKOUT2
      CLKOUT3   => open,         -- 1-bit output: CLKOUT3
      CLKOUT3B  => open,         -- 1-bit output: Inverted CLKOUT3
      CLKOUT4   => open,         -- 1-bit output: CLKOUT4
      CLKOUT5   => clk_u,         -- 1-bit output: CLKOUT5
      CLKOUT6   => open,         -- 1-bit output: CLKOUT6
      -- Feedback Clocks: 1-bit (each) output: Clock feedback ports
      CLKFBOUT  => clkfb,  -- 1-bit output: Feedback clock
      CLKFBOUTB => open,   -- 1-bit output: Inverted CLKFBOUT
      -- Status Ports: 1-bit (each) output: MMCM status ports
      LOCKED    => locked,   -- 1-bit output: LOCK
      -- Clock Inputs: 1-bit (each) input: Clock input
      CLKIN1    => clk24, -- 1-bit input: Clock
      -- Control Ports: 1-bit (each) input: MMCM control ports
      PWRDWN    => '0',    -- 1-bit input: Power-down
      RST       => '0',    -- 1-bit input: Reset
      -- Feedback Clocks: 1-bit (each) input: Clock feedback ports
      CLKFBIN   => clkfb   -- 1-bit input: Feedback clock
   );
```

```
i_vga_gen_1080p: vga_gen_1080p port map (
        clk         => clk_pixel_x1,
        blank       => blank,
        coy         => coy,
        cox         => cox,
        hsync       => hsync,
        vsync       => vsync
    );

  UART_RX_Inst : entity work.UART_RX
    generic map (
      g_CLKS_PER_BIT => 8)
    port map (
      i_Clk       => clk24,
      i_RX_Serial => uart_rx,
      o_RX_DV     => w_RX_DV,
      o_RX_Byte   => w_RX_Byte);
```

```
count_process: process(c)
    begin

if rising_edge(c) then

    if flag = '1' then
        xbal <= "001011001011";
        ybal <= "000101100100";
        rise <= "00";
        sc_a <= "00";
        sc_b <= "00";
        state <= '1';
        tx <= '0';


    elsif  ybal = 715-16 and rise = "00"   then
    --ball hitting bottom boundary coming from left
        xbal <= xbal+1;
        ybal <= ybal-1;
        rise <= "01"; --ball is going left-up


    elsif  ybal = 715-16 and rise = "10" then
    --ball hitting bottom boundary coming from right
        xbal <= xbal-1;
        ybal <= ybal-1;
        rise <= "11"; --ball is going right up

    elsif ybal = 15 and rise = "01" then
    --ball hitting top boundary coming from left
        xbal <= xbal+1;
        ybal <= ybal+1;
        rise <= "00";    -- ball is going left down

    elsif ybal = 15 and rise = "11" then
    --ball hitting top boundary coming from right
        xbal <= xbal+1;
        ybal <= ybal+1;
        rise <= "10";    -- ball is going left down

--  elsif xbal = 1415-7-30 and rise="01" then
--ball hitting right boundary going up -- for debugging
    -- xbal <= xbal-1;
```

```
    -- ybal <= ybal-1;
    -- rise <= "11"; --ball is going right-up


--elsif xbal = 1415-7 and rise="00" then
--ball hitting right boundary going down -- for debugging
--      xbal <= xbal-1;
--      ybal <= ybal+1;
--      rise <= "10"; --ball is going righy down

    elsif xbal = 1415-7-30 then
        if ybal+15>yb_pad-1 and ybal<yb_pad+151 then
           if rise="01" then --ball hitting right paddle going up
               xbal <= xbal-1;
               ybal <= ybal-1;
               rise <= "11"; --ball is going right-up

           elsif rise ="00" then
               xbal <= xbal+1;
               ybal <= ybal-1;
               rise <= "10"; --ball is going righy down
               end if;

            elsif sc_a ="00" or sc_a = "01" then
                sc_a <= std_logic_vector(unsigned(sc_a)+1);
                xbal <= "001011001011";
                ybal<= "000101100100";
                rise <="10";
            elsif sc_a ="10" then
                sc_a <= std_logic_vector(unsigned(sc_a)+1);
                tx <= '1';

            end if;

            elsif xbal = 15+30+8 then
            if ybal+15>ya_pad-1 and ybal<ya_pad+151 then
               if rise="11" then --ball hitting left paddle going up
               xbal <= xbal+1;
               ybal <= ybal-1;
               rise <= "01"; --ball is going left-up
               elsif rise ="10" then
               xbal <= xbal+1;
               ybal <= ybal+1;
```

```
                rise <= "00"; --ball is going right down
                end if;

            elsif sc_b ="00" or sc_b = "01" then
                sc_b <= std_logic_vector(unsigned(sc_b)+1);
                xbal <= "001011001011";
                ybal<= "000101100100";
                rise <="00";
            elsif sc_b ="10" then
                sc_b <= std_logic_vector(unsigned(sc_b)+1);
                tx <='1';


            end if;


            elsif rise = "00" and state = '1' then --ball going right down
                xbal <= xbal+1;
                ybal <= ybal+1;
            elsif rise = "01" and state = '1' then -- ball going right up
                xbal <= xbal+1;
                ybal <= ybal-1;
             elsif rise = "10" and state = '1' then -- ball going left down
                xbal <= xbal-1;
                ybal <= ybal+1;
             elsif rise = "11" and state = '1' then  -- ball is going left up
                xbal <= xbal-1;
                ybal <= ybal-1;


            end if;
        end if;
    end process;


cou_process: process( clk_b)
    begin
        if rising_edge(clk_b) then
            if count = "00000110101101101010111" then
                c <= '1';
                count <= "00000000000000000000";
            else
                count <= std_logic_vector(unsigned(count)+1);
```

```
                c <= '0';
            end if;
        end if;
    end process;

--  send game over! if someone won
  UART_WORD : entity work.uart_word
   port map (
      clk24      => clk24,
      enable     => tx,
      uart_tx    => uart_tx
      );


led_out: process(w_RX_DV)
begin
if rising_edge(w_RX_DV) then
   -- counter <= w_RX_Byte;
   if w_RX_Byte = "00000000" then -- 00 lpdown
    if ya_pad+5+150<715 then
    ya_pad <= ya_pad + 5;
    end if;
   elsif w_RX_Byte = "10100000" then -- a0 lpdup
    if ya_pad-5 > 15 then
    ya_pad <= ya_pad -5 ;
    end if;
   elsif w_RX_Byte = "10000000" then -- 80 rpdown
    if yb_pad+5+150 < 715 then
    yb_pad <= yb_pad + 5;
    end if;
   elsif w_RX_Byte = "10010000" then -- 90 rpup
    if yb_pad-5 >15 then
    yb_pad <= yb_pad - 5;
    end if;
   elsif w_RX_Byte = "11111111" then --ff start game
    flag <= '0';
    state <= '1';
   elsif w_RX_Byte = "00001111" then --0f pause game
    state <= '0';
   elsif w_RX_Byte = "00000001" then --01 restart game
     flag <='1';
```

```
end if;
end if;
end process;

i_vga_output: vga_output Port map (
            clk       => clk_pixel_x1,
            hsync_in  => hsync,
            xball => xbal,
            yball => ybal,
            ya_paddle => ya_pad,
            yb_paddle => yb_pad,
            vsync_in  => vsync,
            blank_in  => blank,

            cox       => cox,
            coy       => coy,
            score_a => sc_a,
            score_b => sc_b,
            vga_hsync => vga_hsync,
            vga_vsync => vga_vsync,
            vga_red   => vga_red,
            vga_green => vga_green,
            vga_blue  => vga_blue,
            vga_blank => vga_blank
        );

 i_vga_to_hdmi: vga_to_hdmi port map (
        pixel_clk     => clk_pixel_x1,
        pixel_clk_x5  => clk_pixel_x5,
        reset         => reset,
        vga_hsync     => vga_hsync,
        vga_vsync     => vga_vsync,
        vga_red       => vga_red,
        vga_green     => vga_green,
        vga_blue      => vga_blue,
        vga_blank     => vga_blank,

        hdmi_tx_clk_p => hdmi_tx_clk_p,
        hdmi_tx_clk_n => hdmi_tx_clk_n,
        hdmi_tx_p     => hdmi_tx_p,
        hdmi_tx_n     => hdmi_tx_n
    );
```

```
end Behavioral;
```

A.2 vga_gen_1080p.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity vga_gen_1080p is
    Port ( clk : in STD_LOGIC;
           blank : out STD_LOGIC := '0';
           cox:out unsigned(11 downto 0);
           coy:out unsigned(11 downto 0);
           hsync : out STD_LOGIC := '0';
           vsync : out STD_LOGIC := '0');
end vga_gen_1080p;

architecture Behavioral of vga_gen_1080p is
    signal x : unsigned(11 downto 0) := (others => '0');
    signal y : unsigned(11 downto 0) := (others => '0');
begin

clk_proc: process(clk)
    begin
        if rising_edge(clk) then
            if x = 1920-1 then
                blank <= '1';
            elsif x = 2200-1 and (y < 1080-1 or y = 1125-1) then
                blank <= '0';
            end if;

            if x = 1920+88-1 then
                hsync <= '1';
            elsif x = 1920+88+44-1 then
                hsync <= '0';
            end if;

            if x = 2200-1 then
                x <= (others => '0');

                if y = 1080+4-1 then
                    vsync  <= '1';
```

```
                elsif y = 1080+4+5-1 then
                    vsync  <= '0';
                end if;

                if y = 1125-1 then
                    y <= (others => '0');
                else
                    y <= y +1;
                end if;
            else
                x <= x + 1;
            end if;
        cox<=x;
        coy <=y;
        end if;

    end process;
end Behavioral;
```

A.3 vga_output

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

Library UNISIM;
use UNISIM.vcomponents.all;

entity vga_output is
    Port ( clk : in STD_LOGIC;
           hsync_in : in STD_LOGIC;
           vsync_in : in STD_LOGIC;
           blank_in : in STD_LOGIC;
           cox:in unsigned(11 downto 0);
           coy:in unsigned(11 downto 0);
           xball:in unsigned(11 downto 0);
           yball:in unsigned(11 downto 0);
           score_a:in std_logic_vector(1 downto 0);
           score_b:in std_logic_vector(1 downto 0);
           ya_paddle: in unsigned(11 downto 0);
           yb_paddle: in unsigned(11 downto 0);
```

```
            vga_hsync : out std_logic;
            vga_vsync : out std_logic;
            vga_red   : out std_logic_vector(7 downto 0);
            vga_green : out std_logic_vector(7 downto 0);
            vga_blue  : out std_logic_vector(7 downto 0);
            vga_blank : out std_logic);
end vga_output;

architecture Behavioral of vga_output is


component draw_line is
    Port ( x : in unsigned(11 downto 0);
           y : in unsigned(11 downto 0);
           len : in unsigned(11 downto 0);
           cox:in unsigned(11 downto 0);
           coy:in unsigned(11 downto 0);
           orientation : in STD_LOGIC;
           draw_en : out std_logic
           );
end component;

component draw_ball is
       Port ( x : in unsigned(11 downto 0); --position of one pixel of the ball
           y : in unsigned(11 downto 0);

           cox:in unsigned(11 downto 0);
           coy:in unsigned(11 downto 0);


           draw_ball: out std_logic
           );
end component;

component draw_score is
    Port ( x:in unsigned(11 downto 0);
           cox:in unsigned(11 downto 0);
           coy:in unsigned(11 downto 0);
           score: in std_logic_vector(1 downto 0);
           draw_sc: out std_logic
           );
end component;
```

```
component draw_rect is
    Port ( x_s : in unsigned(11 downto 0); --position of one pixel of the ball
           y_s : in unsigned(11 downto 0);
           x_f: in unsigned(11 downto 0);
           y_f: in unsigned(11 downto 0);
           cox:in unsigned(11 downto 0);
           coy:in unsigned(11 downto 0);
           p: in std_logic;
           draw_rec: out std_logic
           );
end component;

component draw_rectfilled is
    Port ( x_s : in unsigned(11 downto 0); --position of one pixel of the ball
           y_s : in unsigned(11 downto 0);
           x_f: in unsigned(11 downto 0);
           y_f: in unsigned(11 downto 0);
           cox:in unsigned(11 downto 0);
           coy:in unsigned(11 downto 0);
           p: in std_logic;
           draw_rec: out std_logic
           );
end component;



signal bx: unsigned(11 downto 0);
signal by: unsigned(11 downto 0) ;

signal draw_en: std_logic_vector(1 downto 0);
signal draw_bal: std_logic;
signal ball_col:std_logic_vector(23 downto 0);
signal rec: std_logic;
signal recfilled: std_logic;
signal paddle: std_logic_vector(3 downto 0):= "0000";

signal draw_sc: std_logic_vector(1 downto 0):="00";

begin

o_draw_score: draw_score port map ( --draw score for player one
           x => "010111011100",
           cox => cox,
```

```
            coy => coy,
            score => score_a,
            draw_sc =>draw_sc(0)

);
oo_draw_score: draw_score port map ( -- draw score for player two
             x => "011010011010",
             cox => cox,
            coy => coy,
            score => score_b,
            draw_sc =>draw_sc(1)

);
i_draw_ball: draw_ball port map (         --draw ball

            x => xball,
            y => yball,
            cox => cox,
            coy => coy,

            draw_ball => draw_bal

    );

i_draw_rect: draw_rect port map (  -- draw game boundaries
            x_s => "000000001111",
            y_s => "000000001111",
            x_f => "010110000111",
            y_f => "001011001011",
            cox => cox,
            coy => coy,
            p => '0',
            draw_rec => rec

);


i_draw_rectfilled: draw_rectfilled port map ( -- draw game field
            x_s => "000000001111",
            y_s => "000000001111",
            x_f => "010110000111",
            y_f => "001011001011",
            cox => cox,
```

```
            coy => coy,
            p => '0',
            draw_rec => recfilled

);


pa_draw_rect: draw_rect port map ( -- draw left paddle layout
            x_s => "000000010000",
            y_s => ya_paddle,
            x_f => "000000000000",
            y_f => "000000000000",
            cox => cox,
            coy => coy,
            p => '1',
            draw_rec => paddle(0)

);


pa_draw_rectfilled: draw_rectfilled port map ( -- draw left paddle filling
            x_s => "000000010000",
            y_s => ya_paddle,
            x_f => "000000000000",
            y_f => "000000000000",
            cox => cox,
            coy => coy,
            p => '1',
            draw_rec => paddle(1)

);


pb_draw_rect: draw_rect port map ( -- draw right paddle layout
            x_s => "010101101000",
            y_s => yb_paddle,
            x_f => "000000000000",
            y_f => "000000000000",
            cox => cox,
            coy => coy,
            p => '1',
            draw_rec => paddle(2)
```

```
);


pb_draw_rectfilled: draw_rectfilled port map ( -- draw right paddle filling
            x_s => "010101101000",
            y_s => yb_paddle,
            x_f => "000000000000",
            y_f => "000000000000",
            cox => cox,
            coy => coy,
            p => '1',
            draw_rec => paddle(3)

);

bx <= "000101011110";
by <= "000101111110";
vga_buffer: process(clk)
    begin
        if rising_edge(clk) then
            vga_hsync <= hsync_in;
            vga_vsync <= vsync_in;
            if blank_in = '0' then
                if draw_bal='1' then -- ball flag
                    vga_red <= "11100110" ;
                    vga_green <= "11100110" ;
                    vga_blue <= "00000001" ;

                    vga_blank <= '0' ;

                elsif draw_sc(0) = '1' then -- score one flag
                    vga_red <= "10111111" ;
                    vga_green <= "00000111" ;
                    vga_blue <= "00000111";
                    vga_blank <= '0' ;
                    vga_blank <= '0' ;
                elsif draw_sc(1) = '1' then  -- score two flag
                    vga_red <= "00000100" ;
                    vga_green <= "00000100" ;
                    vga_blue <= "10111111";
                    vga_blank <= '0' ;
                    vga_blank <= '0' ;
                elsif paddle(0) = '1' or paddle(1)='1' then -- paddle_1 flag
```

```
                    vga_red <= "10111111" ;
                    vga_green <= "00000111" ;
                    vga_blue <= "00000111";
                    vga_blank <= '0' ;
                elsif paddle(2) = '1' or paddle(3)='1' then -- paddle_2 flag
                    vga_red <= "00000100" ;
                    vga_green <= "00000100" ;
                    vga_blue <= "10111111";
                    vga_blank <= '0' ;
                elsif rec = '1' then          -- boundaries flag
                    vga_red <= "10111111" ;
                    vga_green <= "10111111" ;
                    vga_blue <= "10111111";
                    vga_blank <= '0' ;
              elsif recfilled='1' then            -- field filling flag
                    vga_red <= "00010001" ;
                        vga_green <= "00101111"  ;
                        vga_blue <= "00000001" ;
                    vga_blank <= '0' ;
              else
                        vga_red    <= "10011111";
                        vga_green  <= "10011111";
                        vga_blue <= "00000001" ;
                        vga_blank <= '0' ;

            end if;
            else
               vga_red    <= (others => '0');
               vga_green  <= (others => '0');
               vga_blue   <= (others => '0');
               vga_blank <= '1';
        END IF;
        end if;
    end process;


end Behavioral;
```

A.4 draw_score.vhd

[style=VHDL]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity draw_score is
    Port ( x: in unsigned(11 downto 0);
           cox:in unsigned(11 downto 0);
           coy:in unsigned(11 downto 0);
           score: in std_logic_vector(1 downto 0);
           draw_sc: out std_logic
           );
end draw_score;

architecture Behavioral of draw_score is

begin
draw : process(cox)
begin
    if (cox=x or cox=x+30 or cox=x+50 or cox=x+80 or cox = x+100 or cox = x+130)
          and (coy>400 and coy<500) then  --boundaries of the game
                   draw_sc <= '1';
     elsif (coy=400 or coy=500 ) and ((cox>x+99 and cox<x+131)
          or (cox>x+49 and cox<x+81) or (cox>x-1 and cox<x+31)) then
          --boundaries of the game
                   draw_sc <= '1';

     elsif score = "01"
                   and (coy> 400 and coy <500 and cox > x and cox < x+30) then
                   draw_sc <= '1';

     elsif score = "10" and (coy> 400 and coy <500 and ((cox > x and cox < x+30)
                            or (cox>x+50 and cox<x+80))) then
                   draw_sc <= '1';

     elsif score = "11" and (coy> 400 and coy <500 and ((cox > x and cox < x+30) or
               (cox>x+50 and cox<x+80) or (cox>x+100 and cox<x+130))) then
```

```
                    draw_sc <= '1';


     else
                    draw_sc <='0';
   -- end if;
    end if;
end process;
end Behavioral;
```

A.5 draw_ball.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity draw_ball is
    Port ( x : in unsigned(11 downto 0); --position of one pixel of the ball
           y : in unsigned(11 downto 0);

           cox:in unsigned(11 downto 0);
           coy:in unsigned(11 downto 0);


           draw_ball: out std_logic
           );
end draw_ball;

architecture Behavioral of draw_ball is

begin
draw : process(cox)
begin
    if   (cox=x or cox=x-1)
        and (coy>y-1 and coy<y+16) then --ball


                    draw_ball <= '1';

    elsif   (cox=x-3 or cox=x-2 or cox=x+2 or cox=x+1)
        and ((coy>y-1 and coy<y+16)) then

                    draw_ball <= '1';
```

```
                elsif   (cox=x-4 or cox=x-5 or cox=x+3 or cox=x+4)
                 and ((coy>y and coy<y+15)) then

                     draw_ball <= '1';

                elsif   (cox=x-6 or cox=x+5 )
                 and ((coy>y+1 and  coy<y+14)) then

                     draw_ball <= '1';
                elsif   (cox=x-7 or cox=x+6 )
                 and ((coy>y+2 and  coy<y+13)) then

                     draw_ball <= '1';

                elsif   (cox=x-8 or cox=x+7 )
                 and (coy>y+4 and coy<y+11) then

                     draw_ball <= '1';
               else
                     draw_ball <= '0';
             end if;
end process;
end Behavioral;
```

A.6 draw_rec.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity draw_rect is
    Port ( x_s : in unsigned(11 downto 0); --position of one pixel of the ball
           y_s : in unsigned(11 downto 0);
           x_f: in unsigned(11 downto 0);
           y_f: in unsigned(11 downto 0);
           cox:in unsigned(11 downto 0);
           coy:in unsigned(11 downto 0);
           p: in std_logic;
           draw_rec: out std_logic
           );
end draw_rect;

architecture Behavioral of draw_rect is
```

```
begin
draw : process(cox)
begin
    if p = '0' then
    if (cox=x_s or cox=x_f )
        and coy>y_s-1 and coy<y_f+1 then
                    draw_rec <= '1';
    elsif (coy=y_s or coy=y_f )
        and cox>x_s and cox<x_f then
                    draw_rec <= '1';
    else
                    draw_rec <='0';
    end if;

    elsif p = '1' then
    if (cox=x_s or cox=x_s+30 )
        and coy>y_s-1 and coy<y_s+149 then
                draw_rec <= '1';
    elsif (coy=y_s or coy=y_s+150 )
        and cox>x_s and cox<x_s+30 then

                draw_rec <= '1';
    else
                draw_rec <='0';


    end if;
    end if;
end process;
end Behavioral;
```

A.7 draw_recfilled

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity draw_rectfilled is
    Port ( x_s : in unsigned(11 downto 0);
           y_s : in unsigned(11 downto 0);
           x_f: in unsigned(11 downto 0);
```

```
            y_f: in unsigned(11 downto 0);
            cox:in unsigned(11 downto 0);
            coy:in unsigned(11 downto 0);
            p : in std_logic;
            draw_rec: out std_logic
            );
end draw_rectfilled;

architecture Behavioral of draw_rectfilled is

begin
draw : process(cox)
begin
    if p = '0' then
    if (cox>x_s and cox<x_f )
     and coy>y_s and coy<y_f then

                    draw_rec <= '1';
    elsif (coy=y_s or coy=y_f )
        and cox>x_s and cox<x_f then

                    draw_rec <= '1';
    else
                    draw_rec <='0';
    end if;
    elsif p = '1' then
    if (cox>x_s and cox<x_s+30 )    --paddlewidth 30px
        and coy>y_s and coy<y_s+150 then  --paddleheight 150px

                    draw_rec <= '1';
    elsif (coy=y_s or coy=y_s+150 )
        and cox>x_s and cox<x_s+30 then

                    draw_rec <= '1';
    else
                    draw_rec <='0';
    end if;
    end if;
end process;
end Behavioral;
```

A.8 vga_to_hdmi.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity vga_to_hdmi is
    port ( pixel_clk : in std_logic;
           pixel_clk_x5 : in std_logic;
           reset  : in std_logic;

           vga_hsync : in std_logic;
           vga_vsync : in std_logic;
           vga_red   : in std_logic_vector(7 downto 0);
           vga_green : in std_logic_vector(7 downto 0);
           vga_blue  : in std_logic_vector(7 downto 0);
           vga_blank : in std_logic;
           hdmi_tx_clk_p : out   std_logic;
           hdmi_tx_clk_n : out   std_logic;
           hdmi_tx_p     : out   std_logic_vector(2 downto 0);
           hdmi_tx_n     : out   std_logic_vector(2 downto 0)
       );
end vga_to_hdmi;

architecture Behavioral of vga_to_hdmi is
   component TDMS_encoder is
   Port ( clk     : in  STD_LOGIC;
          data    : in  STD_LOGIC_VECTOR (7 downto 0);
          c       : in  STD_LOGIC_VECTOR (1 downto 0);
          blank   : in  STD_LOGIC;
          encoded : out  STD_LOGIC_VECTOR (9 downto 0));
    end component;

    component serialiser_10_to_1 is
        Port ( clk : in STD_LOGIC;
               clk_x5 : in STD_LOGIC;
               reset  : in std_logic;
               data : in STD_LOGIC_VECTOR (9 downto 0);
               serial : out STD_LOGIC);
    end component;

    signal serial_clk : std_logic;
    signal serial_ch0 : std_logic;
```

```
    signal serial_ch1 : std_logic;
    signal serial_ch2 : std_logic;

    signal c0_tmds_symbol     : std_logic_vector(9 downto 0);
    signal c1_tmds_symbol     : std_logic_vector(9 downto 0);
    signal c2_tmds_symbol     : std_logic_vector(9 downto 0);
    signal pixel_clk_buffered : std_logic;
begin


c0_tmds: TDMS_encoder port map (
            clk     => pixel_clk,
            data    => vga_blue,
            c(1)    => vga_vsync,
            c(0)    => vga_hsync,
            blank   => vga_blank,
            encoded => c0_tmds_symbol);

c1_tmds: TDMS_encoder port map (
            clk     => pixel_clk,
            data    => vga_green,
            c       => (others => '0'),
            blank   => vga_blank,
            encoded => c1_tmds_symbol);

c2_tmds: TDMS_encoder port map (
            clk     => pixel_clk,
            data    => vga_red,
            c       => (others => '0'),
            blank   => vga_blank,
            encoded => c2_tmds_symbol);

--i_BUFR : BUFIO port map (
--         O => pixel_clk_io,
-- 1-bit output: Clock output (connect to I/O clock loads).
--         CE => '1',
--         CLR => '0',
--         I => pixel_clk_io_raw
-- 1-bit input: Clock input (connect to an IBUF or BUFMR).
--);
--i_BUFIO : BUFIO port map (
--         O => pixel_clk_x5,
-- 1-bit output: Clock output (connect to I/O clock loads).
```

```
--         I => pixel_clk_x5_raw
-- 1-bit input: Clock input (connect to an IBUF or BUFMR).
--);

ser_ch0: serialiser_10_to_1 Port map (
        clk    => pixel_clk,
        clk_x5 => pixel_clk_x5,
        reset  => reset,
        data   => c0_tmds_symbol,
        serial => serial_ch0);

ser_ch1: serialiser_10_to_1 port map (
        clk    => pixel_clk,
        clk_x5 => pixel_clk_x5,
        reset  => reset,
        data   => c1_tmds_symbol,
        serial => serial_ch1);

ser_ch2: serialiser_10_to_1 port map (
        clk    => pixel_clk,
        clk_x5 => pixel_clk_x5,
        reset  => reset,
        data   => c2_tmds_symbol,
        serial => serial_ch2);

ser_clk: serialiser_10_to_1 Port map (
        clk    => pixel_clk,
        clk_x5 => pixel_clk_x5,
        reset  => reset,
        data   => "0000011111",
        serial => serial_clk);

clk_buf: OBUFDS generic map ( IOSTANDARD => "TMDS_33",  SLEW => "FAST")
    port map ( O  => hdmi_tx_clk_p, OB => hdmi_tx_clk_n, I => serial_clk);

tx0_buf: OBUFDS generic map ( IOSTANDARD => "TMDS_33",  SLEW => "FAST")
    port map ( O  => hdmi_tx_p(0), OB => hdmi_tx_n(0), I  => serial_ch0);

tx1_buf: OBUFDS generic map ( IOSTANDARD => "TMDS_33",  SLEW => "FAST")
    port map ( O  => hdmi_tx_p(1), OB => hdmi_tx_n(1), I  => serial_ch1);

tx2_buf: OBUFDS generic map ( IOSTANDARD => "TMDS_33",  SLEW => "FAST")
    port map ( O  => hdmi_tx_p(2), OB => hdmi_tx_n(2), I  => serial_ch2);
```

```
end Behavioral;
```

A.9 TMDS_encoder.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity TDMS_encoder is
    Port ( clk     : in  STD_LOGIC;
           data    : in  STD_LOGIC_VECTOR (7 downto 0);
           c       : in  STD_LOGIC_VECTOR (1 downto 0);
           blank   : in  STD_LOGIC;
           encoded : out  STD_LOGIC_VECTOR (9 downto 0));
end TDMS_encoder;

architecture Behavioral of TDMS_encoder is
   signal xored  : STD_LOGIC_VECTOR (8 downto 0);
   signal xnored : STD_LOGIC_VECTOR (8 downto 0);

   signal ones                : STD_LOGIC_VECTOR (3 downto 0);
   signal data_word           : STD_LOGIC_VECTOR (8 downto 0);
   signal data_word_inv       : STD_LOGIC_VECTOR (8 downto 0);
   signal data_word_disparity : STD_LOGIC_VECTOR (3 downto 0);
   signal dc_bias             : STD_LOGIC_VECTOR (3 downto 0) := (others => '0');
begin
   -- Work our the two different encodings for the byte
   xored(0) <= data(0);
   xored(1) <= data(1) xor xored(0);
   xored(2) <= data(2) xor xored(1);
   xored(3) <= data(3) xor xored(2);
   xored(4) <= data(4) xor xored(3);
   xored(5) <= data(5) xor xored(4);
   xored(6) <= data(6) xor xored(5);
   xored(7) <= data(7) xor xored(6);
   xored(8) <= '1';

   xnored(0) <= data(0);
   xnored(1) <= data(1) xnor xnored(0);
   xnored(2) <= data(2) xnor xnored(1);
   xnored(3) <= data(3) xnor xnored(2);
   xnored(4) <= data(4) xnor xnored(3);
```

```
xnored(5) <= data(5) xnor xnored(4);
xnored(6) <= data(6) xnor xnored(5);
xnored(7) <= data(7) xnor xnored(6);
xnored(8) <= '0';

-- Count how many ones are set in data
ones <= "0000" + data(0) + data(1) + data(2) + data(3)
               + data(4) + data(5) + data(6) + data(7);

-- Decide which encoding to use
process(ones, data(0), xnored, xored)
begin
   if ones > 4 or (ones = 4 and data(0) = '0') then
      data_word     <= xnored;
      data_word_inv <= NOT(xnored);
   else
      data_word     <= xored;
      data_word_inv <= NOT(xored);
   end if;
end process;

-- Work out the DC bias of the dataword;
data_word_disparity  <= "1100" + data_word(0)
      + data_word(1) + data_word(2) + data_word(3)
      + data_word(4) + data_word(5) + data_word(6) + data_word(7);

-- Now work out what the output should be
process(clk)
begin
   if rising_edge(clk) then
      if blank = '1' then
         -- In the control periods, all values have and have balanced bit count
         case c is
            when "00"   => encoded <= "1101010100";
            when "01"   => encoded <= "0010101011";
            when "10"   => encoded <= "0101010100";
            when others => encoded <= "1010101011";
         end case;
         dc_bias <= (others => '0');
      else
         if dc_bias = "00000" or data_word_disparity = 0 then
            -- dataword has no disparity
            if data_word(8) = '1' then
```

```
                encoded <= "01" & data_word(7 downto 0);
                dc_bias <= dc_bias + data_word_disparity;
            else
                encoded <= "10" & data_word_inv(7 downto 0);
                dc_bias <= dc_bias - data_word_disparity;
            end if;
        elsif (dc_bias(3) = '0' and data_word_disparity(3) = '0') or
              (dc_bias(3) = '1' and data_word_disparity(3) = '1') then
            encoded <= '1' & data_word(8) & data_word_inv(7 downto 0);
            dc_bias <= dc_bias + data_word(8) - data_word_disparity;
        else
            encoded <= '0' & data_word;
            dc_bias <= dc_bias - data_word_inv(8) + data_word_disparity;
        end if;
    end if;
  end if;
 end process;
end Behavioral;
```

A.10 10_to_1_serializer.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library UNISIM;
use UNISIM.VComponents.all;

entity serialiser_10_to_1 is
    Port ( clk    : in STD_LOGIC;
           clk_x5 : in STD_LOGIC;
           data   : in STD_LOGIC_VECTOR (9 downto 0);
           reset  : in std_logic;
           serial : out STD_LOGIC);
end serialiser_10_to_1;

architecture Behavioral of serialiser_10_to_1 is
    signal shift1      : std_logic := '0';
    signal shift2      : std_logic := '0';
    signal ce_delay    : std_logic := '0';
    signal reset_delay : std_logic_vector(7 downto 0) := (others => '0');
begin

master_serdes : OSERDESE2
   generic map (
```

```
    DATA_RATE_OQ => "DDR",    -- DDR, SDR
    DATA_RATE_TQ => "DDR",    -- DDR, BUF, SDR
    DATA_WIDTH => 10,          -- Parallel data width (2-8,10,14)
    INIT_OQ => '1',           -- Initial value of OQ output (1'b0,1'b1)
    INIT_TQ => '1',           -- Initial value of TQ output (1'b0,1'b1)
    SERDES_MODE => "MASTER", -- MASTER, SLAVE
    SRVAL_OQ => '0',          -- OQ output value when SR is used (1'b0,1'b1)
    SRVAL_TQ => '0',          -- TQ output value when SR is used (1'b0,1'b1)
    TBYTE_CTL => "FALSE",     -- Enable tristate byte operation (FALSE, TRUE)
    TBYTE_SRC => "FALSE",     -- Tristate byte source (FALSE, TRUE)
    TRISTATE_WIDTH => 1       -- 3-state converter width (1,4)
)
port map (
    OFB       => open,             -- 1-bit output: Feedback path for data
    OQ        => serial,               -- 1-bit output: Data path output
    -- SHIFTOUT1 / SHIFTOUT2: 1-bit (each) output:
    --Data output expansion (1-bit each)
    SHIFTOUT1 => open,
    SHIFTOUT2 => open,
    TBYTEOUT  => open,   -- 1-bit output: Byte group tristate
    TFB       => open,             -- 1-bit output: 3-state control
    TQ        => open,               -- 1-bit output: 3-state control
    CLK       => clk_x5,             -- 1-bit input: High speed clock
    CLKDIV    => clk,       -- 1-bit input: Divided clock
    -- D1 - D8: 1-bit (each) input: Parallel data inputs (1-bit each)
    D1 => data(0),
    D2 => data(1),
    D3 => data(2),
    D4 => data(3),
    D5 => data(4),
    D6 => data(5),
    D7 => data(6),
    D8 => data(7),
    OCE => ce_delay,             -- 1-bit input: Output data clock enable
    RST => reset,              -- 1-bit input: Reset
    -- SHIFTIN1 / SHIFTIN2: 1-bit (each) input: Data input expansion (1-bit each)
    SHIFTIN1 => SHIFT1,
    SHIFTIN2 => SHIFT2,
    -- T1 - T4: 1-bit (each) input: Parallel 3-state inputs
    T1 => '0',
    T2 => '0',
    T3 => '0',
    T4 => '0',
```

```
      TBYTEIN => '0', -- 1-bit input: Byte group tristate
      TCE => '0'                    -- 1-bit input: 3-state clock enable
   );

slave_serdes : OSERDESE2
   generic map (
      DATA_RATE_OQ   => "DDR",   -- DDR, SDR
      DATA_RATE_TQ   => "DDR",   -- DDR, BUF, SDR
      DATA_WIDTH     => 10,      -- Parallel data width (2-8,10,14)
      INIT_OQ        => '1',     -- Initial value of OQ output (1'b0,1'b1)
      INIT_TQ        => '1',     -- Initial value of TQ output (1'b0,1'b1)
      SERDES_MODE    => "SLAVE", -- MASTER, SLAVE
      SRVAL_OQ       => '0',     -- OQ output value when SR is used (1'b0,1'b1)
      SRVAL_TQ       => '0',     -- TQ output value when SR is used (1'b0,1'b1)
      TBYTE_CTL      => "FALSE", -- Enable tristate byte operation (FALSE, TRUE)
      TBYTE_SRC      => "FALSE", -- Tristate byte source (FALSE, TRUE)
      TRISTATE_WIDTH => 1        -- 3-state converter width (1,4)
   )
   port map (
      OFB       => open,         -- 1-bit output: Feedback path for data
      OQ        => open,         -- 1-bit output: Data path output
      -- SHIFTOUT1 / SHIFTOUT2: 1-bit (each) output:
      --Data output expansion (1-bit each)
      SHIFTOUT1 => shift1,
      SHIFTOUT2 => shift2,

      TBYTEOUT  => open,    -- 1-bit output: Byte group tristate
      TFB       => open,    -- 1-bit output: 3-state control
      TQ        => open,    -- 1-bit output: 3-state control
      CLK       => clk_x5,  -- 1-bit input: High speed clock
      CLKDIV    => clk,     -- 1-bit input: Divided clock
      -- D1 - D8: 1-bit (each) input: Parallel data inputs (1-bit each)
      D1        => '0',
      D2        => '0',
      D3        => data(8),
      D4        => data(9),
      D5        => '0',
      D6        => '0',
      D7        => '0',
      D8        => '0',
      OCE       => ce_delay,     -- 1-bit input: Output data clock enable
      RST       => reset,     -- 1-bit input: Reset
      -- SHIFTIN1 / SHIFTIN2: 1-bit (each) input: Data input expansion (1-bit each)
```

```
      SHIFTIN1 => '0',
      SHIFTIN2 => '0',
      -- T1 - T4: 1-bit (each) input: Parallel 3-state inputs
      T1       => '0',
      T2       => '0',
      T3       => '0',
      T4       => '0',
      TBYTEIN  => '0',     -- 1-bit input: Byte group tristate
      TCE      => '0'      -- 1-bit input: 3-state clock enable
   );
delay_ce: process(clk)
    begin
        if rising_edge(clk) then
            ce_delay <= not reset;
        end if;
    end process;
end Behavioral;
```

A.11 uart_rx.vhd

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;

entity UART_RX is
  generic (
    g_CLKS_PER_BIT : integer := 8
    );
  port (
    i_Clk       : in  std_logic;
    i_RX_Serial : in  std_logic;
    o_RX_DV     : out std_logic;
    o_RX_Byte   : out std_logic_vector(7 downto 0)
    );
end UART_RX;


architecture RTL of UART_RX is

  type state is (idle,start_bit,data_rec,stop_bit,clean);
  signal r_state : state := idle;
```

```
  signal clk_count : integer range 0 to g_CLKS_PER_BIT-1 := 0;
  signal index : integer range 0 to 7 := 0;  -- 8 Bits Total
  signal data   : std_logic_vector(7 downto 0) := (others => '0');
  signal flag     : std_logic := '0';

begin

  -- Purpose: Control RX state machine
  p_UART_RX : process (i_Clk)
  begin
    if rising_edge(i_Clk) then

      case r_state is

        when idle =>
          flag     <= '0';
          clk_count <= 0;
          index <= 0;

          if i_RX_Serial = '0' then       -- Start bit detected
            r_state <=start_bit;
          else
            r_state <= idle;
          end if;


        -- Check middle of start bit to make sure it's still low
        when start_bit =>
          if clk_count = (g_CLKS_PER_BIT-2)/2 then
            if i_RX_Serial = '0' then
              clk_count <= 0;  -- reset counter since we found the middle
              r_state   <=data_rec;
            else
              r_state   <= idle;
            end if;
          else
            clk_count <= clk_count + 1;
            r_state   <=start_bit;
          end if;


        -- Wait g_CLKS_PER_BIT-1 clock cycles to sample serial data_rec
        when data_rec =>
```

```
          if clk_count < g_CLKS_PER_BIT-1 then
            clk_count <= clk_count + 1;
            r_state   <=data_rec;
          else
            clk_count             <= 0;
            data(index) <= i_RX_Serial;

            -- Check if we have sent out all bits
            if index < 7 then
              index <= index + 1;
              r_state   <=data_rec;
            else
              index <= 0;
              r_state   <= stop_bit;
            end if;
          end if;


        -- Receive Stop bit.  Stop bit = 1
        when stop_bit =>
          -- Wait g_CLKS_PER_BIT-1 clock cycles for Stop bit to finish
          if clk_count < g_CLKS_PER_BIT-1 then
            clk_count <= clk_count + 1;
            r_state   <= stop_bit;
          else
            flag     <= '1';
            clk_count <= 0;
            r_state   <=clean;
          end if;


        -- Stay here 1 clock
        when clean =>
          r_state <= idle;
          flag   <= '0';


        when others =>
          r_state <= idle;

      end case;
    end if;
  end process p_UART_RX;
```

```
  o_RX_DV   <= flag;
  o_RX_Byte <= data;


  end RTL;
```

A.12 uart_tx.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity UART_TX is
  generic (
    g_CLKS_PER_BIT : integer := 8
    );
  port (
    i_Clk       : in  std_logic;
    i_TX     : in  std_logic;
    i_TX_Byte   : in  std_logic_vector(7 downto 0);
    o_TX_Active : out std_logic;
    o_TX_Serial : out std_logic;
    o_TX_Done   : out std_logic
    );
end UART_TX;


architecture RTL of UART_TX is

  type s_state is (idle, start_bit,data_trans,
                   stop_bit,clean);
  signal state : s_state := idle;

  signal clk_count : integer range 0 to g_CLKS_PER_BIT-1 := 0;
  signal index : integer range 0 to 7 := 0;  -- 8 Bits Total
  signal data   : std_logic_vector(7 downto 0) := (others => '0');
  signal flag   : std_logic := '0';

begin


  p_UART_TX : process (i_Clk)
```

```
begin
  if rising_edge(i_Clk) then

    case state is

      when idle =>
        o_TX_Active <= '0';
        o_TX_Serial <= '1';          -- Drive Line High for Idle
        flag   <= '0';
        clk_count <= 0;
        index <= 0;

        if i_TX = '1' then
          data <= i_TX_Byte;
          state <= start_bit;
        else
          state <= idle;
        end if;


      -- Send out Start Bit. Start bit = 0
      when start_bit =>
        o_TX_Active <= '1';
        o_TX_Serial <= '0';

        -- Wait g_CLKS_PER_BIT-1 clock cycles for start bit to finish
        if clk_count < g_CLKS_PER_BIT-1 then
          clk_count <= clk_count + 1;
          state   <= start_bit;
        else
          clk_count <= 0;
          state   <=data_trans;
        end if;


      -- Wait g_CLKS_PER_BIT-1 clock cycles for data bits to finish
      when data_trans =>
        o_TX_Serial <= data(index);

        if clk_count < g_CLKS_PER_BIT-1 then
          clk_count <= clk_count + 1;
          state   <=data_trans;
        else
```

```
            clk_count <= 0;

            -- Check if we have sent out all bits
            if index < 7 then
              index <= index + 1;
              state   <=data_trans;
            else
              index<= 0;
              state   <= stop_bit;
            end if;
          end if;


        -- Send out Stop bit.  Stop bit = 1
        when stop_bit =>
          o_TX_Serial <= '1';

          -- Wait g_CLKS_PER_BIT-1 clock cycles for Stop bit to finish
          if clk_count < g_CLKS_PER_BIT-1 then
            clk_count <= clk_count + 1;
            state   <= stop_bit;
          else
            flag   <= '1';
            clk_count <= 0;
            state   <=clean;
          end if;


        -- Stay here 1 clock
        when clean =>
          o_TX_Active <= '0';
          flag   <= '1';
          state   <= idle;


        when others =>
          state <= idle;

      end case;
    end if;
  end process p_UART_TX;

  o_TX_Done <= flag;
```

```
end RTL;
```

A.13 uart_word.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

Library UNISIM;
use UNISIM.vcomponents.all;


entity uart_word is
  Port (clk24:in std_logic;
        enable: in std_logic;
  --      finish: out std_logic ;
        uart_tx:out std_logic);
end uart_word;

architecture rtl of uart_word is
  signal count          : std_logic_vector(6 downto 0):="0000000";


  signal cc:std_logic_vector(87 downto 0):=
  "001000000010000101110010011001010111011001111001000001100101011011010110000101100111";
  signal n : integer range 0 to 11 := 0;
  signal ready: std_logic:='0';
  signal active: std_logic:='0';
  signal serial: std_logic:='0';
  signal done: std_logic:='0';
  signal byte: std_logic_vector(7 downto 0):="00000000";
  signal flag : std_logic:='0';

  type s_state is (idle, run);
  signal state : s_state := idle;


BEGIN
```

```
tyy: process(clk24)
   begin

   case state is
     when idle =>
        ready <= '0';
        byte <= "00000000";
        n <=0;
        if enable = '1' then
            state <= run;
        end if;

      when run =>




   if rising_edge(clk24) then
            if count = "1111000" then
               count <= "0000000";
               n <= n+1;
            elsif count ="1110111" or count ="1110110" then
                ready <= '1';
                byte <= cc((8*n+7) downto (8*n)) ;
                count <= std_logic_vector(unsigned(count)+1);
            else
                if n<11 then
                count <= std_logic_vector(unsigned(count)+1);
                elsif n=11 then
                    state <= idle;
                    count <= (others=>'0');
                end if;
                ready <= '0';
            end if;
            end if;
    end case;
    end process;


  UART_TX_Infst : entity work.UART_TX
    generic map (
```

```
      g_CLKS_PER_BIT => 8)
   port map (
      i_CLK       => clk24,
      i_TX      => ready,
      i_TX_Byte    => byte,
      o_TX_Active => active,
      o_TX_Serial => serial,
      o_TX_Done    => open
      );

uart_tx <= serial when active = '1' else '1';
end rtl;
```

A.14 constraints file

```
set_property -dict { PACKAGE_PIN A16     IOSTANDARD LVCMOS33 } [get_ports { clk24 }];
    create_clock -add -name sys_clk_pin -period 41.6666667 [get_ports clk24]

set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { uart_rx
}];
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { uart_tx
}];

##HDMI out
set_property -dict {PACKAGE_PIN N1 IOSTANDARD TMDS_33} [get_ports hdmi_tx_clk_n]
set_property -dict {PACKAGE_PIN N2 IOSTANDARD TMDS_33} [get_ports hdmi_tx_clk_p]
set_property -dict {PACKAGE_PIN T3 IOSTANDARD TMDS_33} [get_ports {hdmi_tx_n[0]}]
set_property -dict {PACKAGE_PIN R3 IOSTANDARD TMDS_33} [get_ports {hdmi_tx_p[0]}]
set_property -dict {PACKAGE_PIN P3 IOSTANDARD TMDS_33} [get_ports {hdmi_tx_n[1]}]
set_property -dict {PACKAGE_PIN N3 IOSTANDARD TMDS_33} [get_ports {hdmi_tx_p[1]}]
set_property -dict {PACKAGE_PIN W2 IOSTANDARD TMDS_33} [get_ports {hdmi_tx_n[2]}]
set_property -dict {PACKAGE_PIN V2 IOSTANDARD TMDS_33} [get_ports {hdmi_tx_p[2]}]
```