Electrical and Computer Engineering, Cal Poly Pomona
ECE3300
Digital Circuit Design Using Verilog
Final Project Report
Verilog Pong Game using the Nexys A7-100T

Group H
Section 3

11th December 2023

Professor:	Dr. Mohamed El Hadedy
Date Due:	11th December, 2023
Group Members:	Javier Eguia Chaire, Zaid Omar, Nourine Mahmoud & Andres Ruiz
Semester:	Fall 2023

Abstract

The project, titled "CPPONG," is an implementation of the classic Pong game on the Nexys A7-100T FPGA development board. The game features a graphical interface displayed through VGA, with a scoring system and lives counter. Additionally, the seven-segment displays showcase the ASCII conversion of keyboard inputs via PS2, providing real-time feedback. The LEDs on the board indicate the assigned keycodes for keyboard inputs, enhancing user interaction.

Control over the game extends beyond the keyboard, allowing manipulation through the board's buttons. The FPGA logic handles the game's state transitions, scoring mechanisms, and interface displays. Notably, the switches on the board are employed to adjust the music frequency and toggle the audio PWM output.

The implementation involves multiple modules, including a VGA controller, a text display unit, a graphics unit, and a timer unit. A finite state machine governs the game's states, such as "new game," "play," "new ball," and "game over." The logic ensures seamless gameplay, responsive scoring, and dynamic screen updates.

The project aligns with principles of digital design and FPGA programming, showcasing effective integration of hardware components for interactive game development. The combination of graphical and text-based outputs, coupled with diverse input methods, highlights the versatility of the Nexys A7-100T FPGA board for implementing interactive applications.

Contents

1 Introduction **4**

2 Music Implementation **4**
- 2.1 Music ROM 5
- 2.2 Tone Divider 7
- 2.3 Music Audio Generator 8

3 UART Implementation **9**
- 3.1 Debouncer 10
- 3.2 PS2 Receiver 11
- 3.3 UART Transmitter 13
- 3.4 UART Buffer Control 14
- 3.5 Binary to ASCII 15

4 Pong Game Implementation **16**
- 4.1 Timer 16
- 4.2 Counter 17
- 4.3 Graphics 19
- 4.4 Game Text 23
  - 4.4.1 Score Region 23
  - 4.4.2 Logo Region 23
  - 4.4.3 Rule Region 23
  - 4.4.4 Game Over Region 24
- 4.5 VGA Controller 27
- 4.6 Clock Divider 30
- 4.7 Seven Segment Display Controller 31

5 Top Implementation **35**

6 Simulation **41**

7 Demo **43**

8 Discussion **46**

9 Conclusions **47**

1 Introduction

The CPPONG project is a comprehensive exploration of FPGA (Field-Programmable Gate Array) design, focusing on the synthesis of digital logic and Verilog programming principles. Beyond the scope of a conventional Pong game for Nexys A7-100T, this project delves into the intricate integration of UART (Universal Asynchronous Receiver-Transmitter), audio PWM (Pulse Width Modulation), and VGA (Video Graphics Array) functionalities.

The inclusion of UART facilitates seamless communication between the FPGA and external devices, notably exemplified in the conversion of PS2 keyboard inputs to ASCII characters. This dynamic interaction expands the project's scope beyond a standalone game, showcasing the versatility of FPGA applications.

Furthermore, the integration of audio PWM introduces an auditory dimension to the project. By modulating pulse widths, the FPGA generates audio signals, enhancing the gaming experience. This not only highlights the FPGA's capabilities in multimedia applications but also underscores its potential in creating immersive, multi-sensory designs.

The VGA functionality forms the visual backbone of CPPONG. Leveraging dedicated modules such as a VGA controller, text and graphics units, the project orchestrates pixel-level graphics rendering. This intricate process underscores the FPGA's prowess in real-time visual output and serves as an academic illustration of advanced digital design principles.

In essence, CPPONG transcends the boundaries of traditional game development, offering a nuanced exploration of FPGA functionalities through the integration of UART, audio PWM, and VGA components. This project stands as a testament to the versatility and adaptability of FPGA technology in fostering interactive and multi-modal applications.

2 Music Implementation

In the music implementation of the CPPONG project, a sophisticated system is devised to generate musical tones through FPGA logic. The `music` module orchestrates this process by employing several interconnected modules. The `music_ROM` module fetches the full note values based on a continuously increasing tone counter. These full notes are then dissected into octave and note components using the `divide_by12` module. The module dynamically maps these note values to specific clock divider values, determining the frequency of the output tone. A counter mechanism regulates the duration of each note, and when the predefined duration is met, the speaker output is toggled, producing the desired musical sound. This implementation showcases the intricate integration of mathematical computations, sequential logic, and real-time control to achieve accurate and dynamic audio synthesis within the FPGA environment.

2.1 Music ROM

The `music_ROM` module serves as a crucial component in the CPPONG project, functioning as a lookup table for musical notes based on specific memory addresses. The sequential logic within the module, triggered by the system clock, facilitates the dynamic generation of musical notes by associating each address with a corresponding note value. This implementation showcases the FPGA's capability to handle sequential data and highlights the efficient use of memory resources for musical synthesis. The clear and concise structure of the code underscores the systematic approach to integrate audio elements into the larger FPGA-based design, contributing to the project's overall functionality.

Figure 1: Music ROM code

```
1
2
3  //////////////////////////////////////////////////
4  // Music ROM module
5  module music_ROM(
6      input clk,
7      input [7:0] address,
8      output reg [7:0] note
9  );
10
11 always @(posedge clk)
12     case(address)
13                 0, 1: note <= 8'd25;  // G
14                 2, 3: note <= 8'd30;  // C
15                 4, 5: note <= 8'd29;  // B
16                 6, 7: note <= 8'd30;  // C
17                 8, 9: note <= 8'd27;  // A
18                 10, 11: note <= 8'd27; // A
19                 12, 13: note <= 8'd27; // A
20                 14, 15: note <= 8'd27; // A
21                 16, 17: note <= 8'd27; // A
22                 18, 19: note <= 8'd27; // A
23                 20, 21: note <= 8'd30; // C
24                 22, 23: note <= 8'd27; // A
25                 24, 25: note <= 8'd25; // G
26                 26, 27: note <= 8'd25; // G
27                 28, 29: note <= 8'd25; // G
28                 30, 31: note <= 8'd25; // G
29                 32, 33: note <= 8'd25; // G
30                 34, 35: note <= 8'd25; // G
31                 36, 37: note <= 8'd30; // C
32                 38, 39: note <= 8'd29; // B
33                 40, 41: note <= 8'd30; // C
34                 42, 43: note <= 8'd27; // A
35                 44, 45: note <= 8'd27; // A
36                 46, 47: note <= 8'd23; // F
37                 48, 49: note <= 8'd27; // A
38                 50, 51: note <= 8'd27; // A
39                 52, 53: note <= 8'd25; // G
40                 54, 55: note <= 8'd25; // G
41                 56, 57: note <= 8'd27; // A
42                 58, 59: note <= 8'd25; // G
43                 60, 61: note <= 8'd23; // F
44                 62, 63: note <= 8'd23; // F
45                 64, 65: note <= 8'd22; // E
46                 66, 67: note <= 8'd22; // E
```

```
                68, 69: note <= 8'd22; // E
                70, 71: note <= 8'd22; // E
                72, 73: note <= 8'd22; // E
                74, 75: note <= 8'd22; // E
                76, 77: note <= 8'd25; // G
                78, 79: note <= 8'd30; // C
                80, 81: note <= 8'd29; // B
                82, 83: note <= 8'd30; // C
                84, 85: note <= 8'd27; // A
                86, 87: note <= 8'd27; // A
                88, 89: note <= 8'd27; // A
                90, 91: note <= 8'd27; // A
                92, 93: note <= 8'd27; // A
                94, 95: note <= 8'd27; // A
                96, 97: note <= 8'd30; // C
                98, 99: note <= 8'd27; // A
                100, 101: note <= 8'd25; // G
                102, 103: note <= 8'd25; // G
                104, 105: note <= 8'd25; // G
                106, 107: note <= 8'd25; // G
                108, 109: note <= 8'd25; // G
                110, 111: note <= 8'd25; // G
                112, 113: note <= 8'd30; // C
                114, 115: note <= 8'd29; // B
                116, 117: note <= 8'd30; // C
                118, 119: note <= 8'd27; // A
                120, 121: note <= 8'd27; // A
                122, 123: note <= 8'd23; // F
                124, 125: note <= 8'd27; // A
                126, 127: note <= 8'd27; // A
                128, 129: note <= 8'd25; // G
                130, 131: note <= 8'd25; // G
                132, 133: note <= 8'd27; // A
                134, 135: note <= 8'd25; // G
                136, 137: note <= 8'd23; // F
                138, 139: note <= 8'd23; // F
                140, 141: note <= 8'd22; // E
                142, 143: note <= 8'd22; // E
                144, 145: note <= 8'd22; // E
                146, 147: note <= 8'd22; // E
                148, 149: note <= 8'd22; // E
                150, 151: note <= 8'd22; // E
                //second part
                152,153: note <= 8'd34; // E;
                154,155: note <= 8'd34; // E;
                156,157: note <= 8'd34; // E;
                157,158: note <= 8'd32; // D;
                159,160: note <= 8'd30; // C;
                161,162: note <= 8'd30; // C;
                163, 164: note <= 8'd27; // A
                165, 166: note <= 8'd27; // A
                167,168: note <= 8'd30; // C;
                169,170: note <= 8'd30; // C;
                171,172: note <= 8'd32; // D;
                173,174: note <= 8'd32; // D;
                175,176: note <= 8'd32; // D;
                177,178: note <= 8'd30; // C;
                179, 180: note <= 8'd27; // A
                181, 182: note <= 8'd27; // A
                183, 184: note <= 8'd25; // G
                185, 186: note <= 8'd24; // F#
                187, 188: note <= 8'd25; // G
                189, 190: note <= 8'd25; // G
                191,192: note <= 8'd34; // E;
```

```
                193,194: note <= 8'd34; // E;
                195,196: note <= 8'd34; // E;
                197,198: note <= 8'd32; // D;
                199,200: note <= 8'd30; // C;
                201,202: note <= 8'd30; // C;
                203, 204: note <= 8'd27; // A
                205, 206: note <= 8'd27; // A
                207,208: note <= 8'd32; // D;
                209,210: note <= 8'd30; // C;
                211,212: note <= 8'd29; // B;
                213,214: note <= 8'd29; // B;
                215,216: note <= 8'd30; // C;
                217,218: note <= 8'd32; // D;
                219,220: note <= 8'd32; // D;
                221,222: note <= 8'd30; // C;
                223,224: note <= 8'd30; // C;
                225,226: note <= 8'd30; // C;
                227,228: note <= 8'd32; // D
                229,230: note <= 8'd34; // E;
                231,232: note <= 8'd35; // F;
                233,234: note <= 8'd37; // G;
                235,236: note <= 8'd39; // A;
                237,238: note <= 8'd41; // B;
                239,240: note <= 8'd42; // C;
                241,242: note <= 8'd42; // C;
                243,244: note <= 8'd42; // C;
                245,246: note <= 8'd42; // C;
                247,248: note <= 8'd42; // C;
                249,250: note <= 8'd42; // C;
                        default: note <= 8'd0;
    endcase
endmodule
```

2.2 Tone Divider

The `divide_by12` module is a fundamental building block in the CPPONG project, designed to efficiently divide a given 6-bit input (‘numerator‘) by 12. The module employs a case statement based on the four most significant bits of the input to determine both the quotient (‘quotient‘) and the remainder (‘remainder‘). This mechanism aligns with the musical concept of octaves, where each division by 12 represents a move to the next octave. By incorporating this module, the CPPONG project demonstrates how FPGA-based designs can seamlessly integrate mathematical operations, like division, to manipulate and control musical elements, showcasing the versatility of FPGA implementations beyond traditional digital logic applications.

Figure 2: Tone Divider code

```
module divide_by12(
        input [5:0] numerator,  // value to be divided by 12
        output reg [2:0] quotient,
        output [3:0] remainder
);

reg [1:0] remainder3to2;
```

```
always @(numerator[5:2])
case(numerator[5:2])
         0: begin quotient=0; remainder3to2=0; end
         1: begin quotient=0; remainder3to2=1; end
         2: begin quotient=0; remainder3to2=2; end
         3: begin quotient=1; remainder3to2=0; end
         4: begin quotient=1; remainder3to2=1; end
         5: begin quotient=1; remainder3to2=2; end
         6: begin quotient=2; remainder3to2=0; end
         7: begin quotient=2; remainder3to2=1; end
         8: begin quotient=2; remainder3to2=2; end
         9: begin quotient=3; remainder3to2=0; end
        10: begin quotient=3; remainder3to2=1; end
        11: begin quotient=3; remainder3to2=2; end
        12: begin quotient=4; remainder3to2=0; end
        13: begin quotient=4; remainder3to2=1; end
        14: begin quotient=4; remainder3to2=2; end
        15: begin quotient=5; remainder3to2=0; end
endcase

assign remainder[1:0] = numerator[1:0];  // the first 2 bits are copied
 ↪  through
assign remainder[3:2] = remainder3to2;  // and the last 2 bits come
 ↪  from the case statement
endmodule
```

2.3 Music Audio Generator

The `music` module in the CPPONG project serves as a crucial component for generating musical tones and controlling the speaker output. It employs a combination of modules to manage the frequency and duration of musical notes. The `music_ROM` module is utilized to fetch the full note value based on a continuously increasing tone counter, and the `divide_by12` module extracts the octave and note components from this full note value.

The note values obtained are then mapped to specific clock divider values in the `clkdivider` case statement, determining the frequency of the output tone. A counter mechanism (`counter_note` and `counter_octave`) controls the duration of each note, ensuring accurate timing. When a note is active and the predefined duration is met, the speaker output is toggled, producing sound. This dynamic interplay of frequency modulation and duration control showcases the integration of FPGA-based logic in creating a versatile music generation system. The `music` module, by combining mathematical computations and sequential logic, demonstrates the FPGA's capability to handle real-time audio synthesis and manipulation.

Figure 3: Tone Divider code

```


//////////////////////////////////////////////////
module music(
        input clk,
```

```
        output reg speaker
);

reg [30:0] tone;
always @(posedge clk) tone <= tone+31'd1;

wire [7:0] fullnote;
music_ROM get_fullnote(.clk(clk), .address(tone[29:22]),
 ↪  .note(fullnote));

wire [2:0] octave;
wire [3:0] note;
divide_by12 get_octave_and_note(.numerator(fullnote[5:0]),
 ↪  .quotient(octave), .remainder(note));

reg [8:0] clkdivider;
always @*
case(note)
         0: clkdivider = 9'd511;//A
         1: clkdivider = 9'd482;// A#/Bb
         2: clkdivider = 9'd455;//B
         3: clkdivider = 9'd430;//C
         4: clkdivider = 9'd405;// C#/Db
         5: clkdivider = 9'd383;//D
         6: clkdivider = 9'd361;// D#/Eb
         7: clkdivider = 9'd341;//E
         8: clkdivider = 9'd322;//F
         9: clkdivider = 9'd303;// F#/Gb
        10: clkdivider = 9'd286;//G
        11: clkdivider = 9'd270;// G#/Ab
        default: clkdivider = 9'd0;
endcase

reg [8:0] counter_note;
reg [7:0] counter_octave;
always @(posedge clk) counter_note <= counter_note==0 ? clkdivider :
 ↪  counter_note-9'd1;
always @(posedge clk) if(counter_note==0) counter_octave <=
 ↪  counter_octave==0 ? 8'd255 >> octave : counter_octave-8'd1;
always @(posedge clk) if(counter_note==0 && counter_octave==0 &&
 ↪  fullnote!=0 && tone[21:18]!=0) speaker <= ~speaker;
endmodule
```

3 UART Implementation

In the CPPONG project, the UART implementation is designed to facilitate communication between the FPGA board and external devices, potentially a computer or other systems. The UART functionality is orchestrated through several Verilog modules. Let's break down the key components of the UART implementation:

1. UART Transmitter (`uart_tx`):

- The `uart_tx` module is responsible for transmitting data over UART.
- It operates with a specified clock (`clk`) and takes an 8-bit data input (`tbus`), a start signal (`start`), and produces a transmit signal (`tx`) and a ready signal (`ready`).
- The transmitter uses a shift register (`shift`) to serialize the input data. It is triggered by the `start` signal and sends the data out as a serial bitstream.

2. UART Buffer Control (`uart_buf_con`):
   - The `uart_buf_con` module manages the control signals for UART transmission.
   - It takes the clock input (`clk`), byte count input (`bcount`), data input (`tbuf`), start signal (`start`), and produces signals such as readiness (`ready`), start of transmission (`tstart`), and data bus (`tbus`).
   - This module plays a crucial role in controlling when to start transmitting data and signaling the readiness of the UART system.
3. ASCII to Binary Conversion (`bin2ascii`):
   - The `bin2ascii` module converts binary data (`keycodev`) into ASCII format (`tbuf`), preparing it for UART transmission.
   - It includes logic to convert binary data into ASCII characters, ensuring compatibility with standard UART communication.
4. PS/2 Receiver (`PS2_Receiver`):
   - While not directly related to UART, the PS/2 receiver (`PS2_Receiver`) is an integral part of the system, handling the reception of data from the PS/2 keyboard.
   - The PS/2 receiver processes keyboard signals and produces a `keycode` output, indicating the key that was pressed or released.

3.1 Debouncer

The `debouncer` module plays a crucial role in stabilizing input signals, ensuring reliable and noise-free operation within the CPPONG project. As part of the debouncing mechanism, the module takes input from a potentially noisy source, such as a button press, and produces a clean and stable output. It utilizes a simple yet effective algorithm to filter out rapid changes in the input signal.

Operating on a clock signal, the module compares the current input state (`I`) with its previous state (`Iv`). If the states match, it increments an internal counter (`count`). When this counter reaches a predefined maximum count (`COUNT_MAX`), the output (`O`) is updated to reflect the stable input state. In case of a mismatch

between the current and previous input states, the counter is reset to zero, and the previous input state is updated accordingly.

By implementing this debouncing mechanism, the `debouncer` module ensures that only valid and stable input signals are propagated, preventing unintended triggers caused by noise or bouncing in the physical input devices. This robust debouncing process contributes to the overall reliability and accuracy of the user input interface in the CPPONG project.

Figure 4: Debouncer code

```
`timescale 1ns / 1ps

module debouncer(
    input clk,
    input I,
    output reg O
    );
    parameter COUNT_MAX=255, COUNT_WIDTH=8;
    reg [COUNT_WIDTH-1:0] count;
    reg Iv=0;
    always@(posedge clk)
        if (I == Iv) begin
            if (count == COUNT_MAX)
                O <= I;
            else
                count <= count + 1'b1;
        end else begin
            count <= 'b0;
            Iv <= I;
        end

endmodule
```

3.2 PS2 Receiver

The `PS2_Receiver` module serves as a key component in the CPPONG project, responsible for decoding signals from a PS/2 keyboard into meaningful keycodes. It interfaces with the keyboard through clock (`kclk`) and data (`kdata`) lines, utilizing debouncing mechanisms (`db_clk` and `db_data`) to filter out noise and ensure the stability of input signals.

Operating on the falling edge of the keyboard clock (`kclkf`), the module captures the data bits transmitted by the keyboard during each clock cycle. The received bits are sequentially stored in `datacur`, representing a complete 8-bit keycode. A flag (`oflag`) is set high for a single clock cycle when a complete keycode is received, signaling the availability of valid data. The module efficiently handles the start bit and subsequent data bits, updating the internal state (`cnt`) accordingly.

The key innovation lies in the integration of a debouncing mechanism for both the clock and data lines, enhancing the reliability of signal interpretation. On each positive clock edge, the module updates the `keycode` output with the new

keycode, and the flag is set to indicate the presence of valid data. This robust PS/2 receiver module ensures accurate and stable communication between the FPGA and the PS/2 keyboard, forming a fundamental aspect of the user input interface in the CPPONG project.

Figure 5: PS2 Receiver code

```
`timescale 1ns / 1ps

module PS2_Receiver(
    input clk,
    input kclk,
    input kdata,
    output reg [15:0] keycode=0,
    output reg oflag
    );

    wire kclkf, kdataf;
    reg [7:0]datacur=0;
    reg [7:0]dataprev=0;
    reg [3:0]cnt=0;
    reg flag=0;

debouncer #(
    .COUNT_MAX(19),
    .COUNT_WIDTH(5)
) db_clk(
    .clk(clk),
    .I(kclk),
    .O(kclkf)
);
debouncer #(
   .COUNT_MAX(19),
   .COUNT_WIDTH(5)
) db_data(
    .clk(clk),
    .I(kdata),
    .O(kdataf)
);

always@(negedge(kclkf))begin
    case(cnt)
    0:;//Start bit
    1:datacur[0]<=kdataf;
    2:datacur[1]<=kdataf;
    3:datacur[2]<=kdataf;
    4:datacur[3]<=kdataf;
    5:datacur[4]<=kdataf;
    6:datacur[5]<=kdataf;
    7:datacur[6]<=kdataf;
    8:datacur[7]<=kdataf;
    9:flag<=1'b1;
    10:flag<=1'b0;

    endcase
        if(cnt<=9) cnt<=cnt+1;
        else if(cnt==10) cnt<=0;
end

reg pflag;
always@(posedge clk) begin
    if (flag == 1'b1 && pflag == 1'b0) begin
        keycode <= {dataprev, datacur};
```

```
        oflag <= 1'b1;
        dataprev <= datacur;
    end else
        oflag <= 'b0;
    pflag <= flag;
end

endmodule
```

3.3 UART Transmitter

The `uart_tx` module in the CPPONG project serves as the transmitter for UART communication, enabling the FPGA to receive input from an external source, such as a keyboard. Operating on a specified clock signal, the module efficiently manages the transmission of data. It employs a shift register mechanism, where the input data is serialized and transmitted bit by bit. The module includes a control logic that initiates the transmission upon receiving a start signal and manages the timing of the data bits. Additionally, the module incorporates a countdown mechanism to regulate the duration of the transmission, ensuring proper synchronization with the UART protocol. This implementation exemplifies an integral component for interfacing the FPGA with external devices and facilitates the reception of keyboard inputs for the CPPONG game.

Figure 6: UART Transmitter code

```
`timescale 1ns / 1ps

module uart_tx(
    input       clk   ,
    input [7:0] tbus  ,
    input       start,
    output      tx    ,
    output      ready
    );
    parameter CD_MAX=10416, CD_WIDTH=16;
    reg [CD_WIDTH-1:0] cd_count=0;
    reg [3:0] count=0;
    reg running=0;
    reg [10:0] shift=11'h7ff;
    always@(posedge clk) begin
        if (running == 1'b0) begin
            shift <= {2'b11, tbus, 1'b0};
            running <= start;
            cd_count <= 'b0;
            count <= 'b0;
        end else if (cd_count == CD_MAX) begin
            shift <= {1'b1, shift[10:1]};
            cd_count <= 'b0;
            if (count == 4'd10) begin
                running <= 1'b0;
                count <= 'b0;
            end
            else
                count <= count + 1'b1;
        end else
```

```
            cd_count <= cd_count + 1'b1;
    end
    assign tx = (running == 1'b1) ? shift[0] : 1'b1;
    assign ready = ((running == 1'b0 && start == 1'b0) || (cd_count ==
     ↪  CD_MAX && count == 4'd10)) ? 1'b1 : 1'b0;
endmodule
```

3.4 UART Buffer Control

The `uart_buf_con` module in the CPPONG project serves as a crucial bridge between the UART communication and the internal logic of the FPGA. It manages the transmission of data (`tbuf`) from the UART buffer to the internal system based on control signals such as `clk`, `bcount`, and `start`. Additionally, it handles the formatting of data for proper display.

Upon receiving a ready signal (`tready`), the module checks the state of the system. If in a running state, it decrements the selection counter (`sel`) and triggers the transmission start (`tstart`). Once the selection counter reaches a certain point, the transmission is concluded, and the system goes into a ready state.

When not in a running state and provided with a non-zero byte count (`bcount`), the module initializes the internal buffer (`pbuf`) with the incoming data and starts the transmission process. It sets the selection counter and triggers the transmission start accordingly.

The `tbus` output is dynamically assigned based on the selection counter and the internal buffer. It outputs different components of the data, including start and stop bits, ensuring the proper format for UART transmission.

This module acts as a vital interface, ensuring efficient communication between the external UART interface and the internal logic of the FPGA, facilitating the overall functionality of the CPPONG project.

Figure 7: UART Buffer Control code

```
`timescale 1ns / 1ps

module uart_buf_con(
    input             clk,
    input      [ 2:0] bcount,
    input      [31:0] tbuf,
    input             start,
    output            ready,
    output reg        tstart=0,
    input             tready,
    output reg [ 7:0] tbus=0
    );
    reg [2:0] sel=0;
    reg [31:0] pbuf=0;
    reg running=0;
    initial tstart <= 'b0;
    initial tbus <= 'b0;
```

```
18      always@(posedge clk)
19          if (tready == 1'b1) begin
20              if (running == 1'b1) begin
21                  if (sel == 4'd1) begin
22                      running <= 1'b0;
23                      sel <= bcount + 2'd2;
24                  end else begin
25                      sel <= sel - 1'b1;
26                      tstart <= 1'b1;
27                      running <= 1'b1;
28                  end
29              end else begin
30                  if (bcount != 2'b0) begin
31                      pbuf <= tbuf;
32                      tstart <= start;
33                      running <= start;
34                      sel <= bcount + 2'd2;
35                  end
36              end
37          end else
38              tstart <= 1'b0;
39      assign ready = ~running;
40      always@(sel, pbuf)
41          case (sel)
42          1: tbus <= 8'd13;
43          2: tbus <= 8'd10;
44          3: tbus <= pbuf[7:0];
45          4: tbus <= pbuf[15:8];
46          5: tbus <= 8'd32;
47          6: tbus <= pbuf[23:16];
48          7: tbus <= pbuf[31:24];
49          default: tbus <= 8'd0;
50          endcase
51  endmodule
```

3.5 Binary to ASCII

The `bin2ascii` module plays a crucial role in the CPPONG project by converting binary data into ASCII format, facilitating the display of relevant information on the seven-segment displays of the Nexys A7-100T FPGA board. With a parameterized number of bytes (NBYTES), this module efficiently transforms binary input data (I) into a corresponding ASCII representation (O).

Using a generate block, the module iterates through each nibble (4 bits) of the input data, mapping it to the appropriate ASCII character. For each nibble, if the value is in the range of 0 to 9, it is directly converted to the ASCII representation ('0' to '9'). Otherwise, if the value is in the range of 10 to 15, it is converted to the ASCII representation ('A' to 'F'). The resulting ASCII characters are concatenated in the output vector (O), providing a human-readable representation of the binary input.

This module is particularly significant for the project, as it enables the display of relevant information on the FPGA board's seven-segment displays, contributing to the overall user interface by presenting data in a format easily interpretable by

users. CPPONG game.

Figure 8: Binary to ASCII code

```
`timescale 1ns / 1ps

module bin2ascii(
    input [NBYTES*8-1:0] I,
    output reg [NBYTES*16-1:0] O=0
    );
    parameter NBYTES=2;
    genvar i;
    generate for (i=0; i<NBYTES*2; i=i+1)
        always@(I)
        if (I[4*i+3:4*i] >= 4'h0 && I[4*i+3:4*i] <= 4'h9)
            O[8*i+7:8*i] = 8'd48 + I[4*i+3:4*i];
        else
            O[8*i+7:8*i] = 8'd55 + I[4*i+3:4*i];
    endgenerate
endmodule
```

4 Pong Game Implementation

4.1 Timer

The timer module is a pivotal component in the game design, functioning as a reliable timekeeping mechanism. Its primary purpose is to measure and control time intervals critical for coordinating events within the game. Technically, the module features a 7-bit binary counter, denoted by the `timer_reg` signal, which effectively counts down with each clock cycle when initiated by the `timer_start` signal. The counter can also be decremented upon receiving the `timer_tick` signal, enabling external triggering of the timer.

Upon a system reset, the timer is initialized to its maximum value of 127, as dictated by the 7-bit width. The countdown commences only when the `timer_start` signal is asserted. The combinational logic block determines the next state of the timer (`timer_next`) based on specific conditions: resetting to the maximum value if `timer_start` is active, decrementing by 1 when both `timer_tick` is asserted and the current timer value is nonzero, or maintaining the current value otherwise.

The `timer_up` output signal serves as a crucial indicator, becoming active when the timer reaches zero. This signal plays a pivotal role in triggering predefined events or actions in the game, providing a robust mechanism for time-dependent functionalities. In the context of the game design, this timer module stands out as a versatile tool for managing temporal aspects.

Figure 9: Timer code

```
`timescale 1ns / 1ps

module timer(
    input clk,
    input reset,
    input timer_start, timer_tick,
    output timer_up
    );

    // signal declaration
    reg [6:0] timer_reg, timer_next;

    // register control
    always @(posedge clk or posedge reset)
        if(reset)
            timer_reg <= 7'b1111111;
        else
            timer_reg <= timer_next;

    // next state logic
    always @*
        if(timer_start)
            timer_next = 7'b1111111;
        else if((timer_tick) && (timer_reg != 0))
            timer_next = timer_reg - 1;
        else
            timer_next = timer_reg;

    // output
    assign timer_up = (timer_reg == 0);

endmodule
```

4.2 Counter

The m100_counter module serves as a two-digit decimal counter with individual control signals for incrementing (`d_inc`) and clearing (`d_clr`). It is designed to operate within a digital system, such as a display unit, where numerical values need to be counted or displayed. The module features a synchronous design, driven by the input clock (`clk`) and incorporates a reset mechanism for initialization.

The internal logic of the module is structured around two 4-bit registers (`r_dig0` and `r_dig1`) representing the two digits of the counter. These registers are updated on the rising edge of the clock or in response to a reset signal (`reset`). The next state logic block is responsible for determining the values of the next states (`dig0_next` and `dig1_next`) based on the control signals.

When the clear signal (`d_clr`) is asserted, both digit registers are set to zero, effectively resetting the counter. Conversely, if the increment signal (`d_inc`) is active, the counter behaves in a cascading manner. The least significant digit (`dig0`) increments by one with each clock cycle, and when it reaches 9, it resets to zero while simultaneously incrementing the most significant digit (`dig1`). This design allows the counter to cycle through values from 00 to 99, emulating a two-digit decimal counter.

The output ports (dig0 and dig1) provide the current values of the counter and can be used for further processing or display purposes within the broader digital system. Overall, the m100_counter module offers a versatile solution for implementing a decimal counter with precise control over incrementing and clearing actions.

Figure 10: Counter code

```
`timescale 1ns / 1ps

module m100_counter(
    input clk,
    input reset,
    input d_inc, d_clr,
    output [3:0] dig0, dig1
    );

    // signal declaration
    reg [3:0] r_dig0, r_dig1, dig0_next, dig1_next;

    // register control
    always @(posedge clk or posedge reset)
        if(reset) begin
            r_dig1 <= 0;
            r_dig0 <= 0;
        end

        else begin
            r_dig1 <= dig1_next;
            r_dig0 <= dig0_next;
        end

    // next state logic
    always @* begin
        dig0_next = r_dig0;
        dig1_next = r_dig1;

        if(d_clr) begin
            dig0_next <= 0;
            dig1_next <= 0;
        end

        else if(d_inc)
            if(r_dig0 == 9) begin
                dig0_next = 0;

                if(r_dig1 == 9)
                    dig1_next = 0;
                else
                    dig1_next = r_dig1 + 1;
            end

            else    // dig0 != 9
                dig0_next = r_dig0 + 1;
    end

    // output
    assign dig0 = r_dig0;
    assign dig1 = r_dig1;

endmodule
```

4.3 Graphics

The pong_graph module functions as a graphics controller for a Pong game, determining the positions and interactions of various graphical elements such as walls, paddles, and the ball within the display area. The module operates in sync with the input clock (clk) and incorporates mechanisms for game state control, including a reset signal (reset). It receives information about button presses, game state changes, and positional data from the game logic, influencing the graphical output accordingly.

Key components include the creation of a 60Hz refresh tick (refresh_tick) based on the vertical sync signal, which resets counters and controls the periodic update of the graphical elements. The module defines boundaries for walls, paddles, and the ball within the display area, utilizing parameters to specify their positions and dimensions. The paddle's position is controlled by button inputs (up and down), allowing the player to interact with the game.

The ball's movement is determined by velocity parameters and is responsive to collisions with the walls, paddle, and screen boundaries. The module employs a square ROM to represent the ball's shape, with logic to map pixel positions to ROM data, enabling the creation of a round ball on the screen. Collision detection is implemented to alter the ball's direction and trigger events such as hits or misses.

The module outputs signals (graph_on, hit, miss, graph_rgb) indicating the status of graphical elements within the display area. A multiplexing circuit determines the color of each pixel based on the current graphical element, creating a visually dynamic representation of the Pong game. The output is controlled by the video_on signal, ensuring that graphics are displayed only when the game is active. Overall, the pong_graph module serves as a vital component in creating a visually engaging and interactive Pong game on a display screen.

Figure 11: Graphics code

```
1  `timescale 1ns / 1ps
2
3  module pong_graph(
4      input clk,
5      input reset,
6      input [31:0] btn,          // btn[0] = up, btn[1] = down    UART
       ↪  enable W up and S down in the keyboard
7      input [1:0] btn_altern,
8      input gra_still,         // still graphics - newgame, game over
       ↪  states
9      input video_on,
10     input [9:0] x,
11     input [9:0] y,
12     output graph_on,
13     output reg hit, miss,   // ball hit or miss
14     output reg [11:0] graph_rgb
15     );
16
17     // maximum x, y values in display area
```

```
    parameter X_MAX = 639;
    parameter Y_MAX = 479;

    // create 60Hz refresh tick
    wire refresh_tick;
    assign refresh_tick = ((y == 481) && (x == 0)) ? 1 : 0; // start of vsync(vertical retrace)


    // WALLS
    // LEFT wall boundaries
    parameter L_WALL_L = 32;
    parameter L_WALL_R = 39;    // 8 pixels wide
    // TOP wall boundaries
    parameter T_WALL_T = 64;
    parameter T_WALL_B = 71;    // 8 pixels wide
    // BOTTOM wall boundaries
    parameter B_WALL_T = 472;
    parameter B_WALL_B = 479;    // 8 pixels wide



    // PADDLE
    // paddle horizontal boundaries
    parameter X_PAD_L = 600;
    parameter X_PAD_R = 603;    // 4 pixels wide
    // paddle vertical boundary signals
    wire [9:0] y_pad_t, y_pad_b;
    parameter PAD_HEIGHT = 160;  // 72 pixels high
    // register to track top boundary and buffer
    reg [9:0] y_pad_reg = 204;      // Paddle starting position
    reg [9:0] y_pad_next;
    // paddle moving velocity when a button is pressed
    parameter PAD_VELOCITY = 2;     // change to speed up or slow down paddle movement


    // BALL
    // square rom boundaries
    parameter BALL_SIZE = 8;
    // ball horizontal boundary signals
    wire [9:0] x_ball_l, x_ball_r;
    // ball vertical boundary signals
    wire [9:0] y_ball_t, y_ball_b;
    // register to track top left position
    reg [9:0] y_ball_reg, x_ball_reg;
    // signals for register buffer
    wire [9:0] y_ball_next, x_ball_next;
    // registers to track ball speed and buffers
    reg [9:0] x_delta_reg, x_delta_next;
    reg [9:0] y_delta_reg, y_delta_next;
    // positive or negative ball velocity
    parameter BALL_VELOCITY_POS = 2;    // ball speed positive pixel direction(down, right)
    parameter BALL_VELOCITY_NEG = -2;   // ball speed negative pixel direction(up, left)
    // round ball from square image
    wire [2:0] rom_addr, rom_col;   // 3-bit rom address and rom column
    reg [7:0] rom_data;             // data at current rom address
    wire rom_bit;                   // signify when rom data is 1 or 0 for ball rgb control


    // Register Control
    always @(posedge clk or posedge reset)
```

```
78          if(reset) begin
79              y_pad_reg <= 204;
80              x_ball_reg <= 0;
81              y_ball_reg <= 0;
82              x_delta_reg <= 10'h002;
83              y_delta_reg <= 10'h002;
84          end
85          else begin
86              y_pad_reg <= y_pad_next;
87              x_ball_reg <= x_ball_next;
88              y_ball_reg <= y_ball_next;
89              x_delta_reg <= x_delta_next;
90              y_delta_reg <= y_delta_next;
91          end
92
93
94      // ball rom
95      always @*
96          case(rom_addr)
97              3'b000 :    rom_data = 8'b00111100; //   ****
98              3'b001 :    rom_data = 8'b01111110; //  ******
99              3'b010 :    rom_data = 8'b11111111; // ********
100             3'b011 :    rom_data = 8'b11111111; // ********
101             3'b100 :    rom_data = 8'b11111111; // ********
102             3'b101 :    rom_data = 8'b11111111; // ********
103             3'b110 :    rom_data = 8'b01111110; //  ******
104             3'b111 :    rom_data = 8'b00111100; //   ****
105         endcase
106
107
108     // OBJECT STATUS SIGNALS
109     wire l_wall_on, t_wall_on, b_wall_on, pad_on, sq_ball_on, ball_on;
110     wire [11:0] wall_rgb, pad_rgb, ball_rgb, bg_rgb;
111
112
113     // pixel within wall boundaries
114     assign l_wall_on = ((L_WALL_L <= x) && (x <= L_WALL_R)) ? 1 : 0;
115     assign t_wall_on = ((T_WALL_T <= y) && (y <= T_WALL_B)) ? 1 : 0;
116     assign b_wall_on = ((B_WALL_T <= y) && (y <= B_WALL_B)) ? 1 : 0;
117
118
119     // assign object colors
120     assign wall_rgb   = 12'h006;    // navy blue walls
121     assign pad_rgb    = 12'h444;    // grey paddle
122     assign ball_rgb   = 12'hfff;    // white ball
123     assign bg_rgb     = 12'h262;    // dark green background
124
125
126     // paddle
127     assign y_pad_t = y_pad_reg;                             // paddle
        ↪ top position
128     assign y_pad_b = y_pad_t + PAD_HEIGHT - 1;              // paddle
        ↪ bottom position
129     assign pad_on = (X_PAD_L <= x) && (x <= X_PAD_R) &&     // pixel
        ↪ within paddle boundaries
130                 (y_pad_t <= y) && (y <= y_pad_b);
131
132
133     // Paddle Control
134     always @* begin
135         y_pad_next = y_pad_reg;     // no move
136
137         if(refresh_tick)
138             if((btn==32'h31443142 || btn_altern[1]) & (y_pad_b <
                ↪ (B_WALL_T - 1 - PAD_VELOCITY)))
```

```
139                 y_pad_next = y_pad_reg + PAD_VELOCITY;  // move down
140             else if((btn==32'h31423144 || btn_altern[0]) & (y_pad_t >
                 ↪  (T_WALL_B - 1 - PAD_VELOCITY)))
141                 y_pad_next = y_pad_reg - PAD_VELOCITY;  // move up
142     end
143
144
145     // rom data square boundaries
146     assign x_ball_l = x_ball_reg;
147     assign y_ball_t = y_ball_reg;
148     assign x_ball_r = x_ball_l + BALL_SIZE - 1;
149     assign y_ball_b = y_ball_t + BALL_SIZE - 1;
150     // pixel within rom square boundaries
151     assign sq_ball_on = (x_ball_l <= x) && (x <= x_ball_r) &&
152                         (y_ball_t <= y) && (y <= y_ball_b);
153     // map current pixel location to rom addr/col
154     assign rom_addr = y[2:0] - y_ball_t[2:0];   // 3-bit address
155     assign rom_col = x[2:0] - x_ball_l[2:0];    // 3-bit column index
156     assign rom_bit = rom_data[rom_col];         // 1-bit signal rom
         ↪  data by column
157     // pixel within round ball
158     assign ball_on = sq_ball_on & rom_bit;      // within square
         ↪  boundaries AND rom data bit == 1
159
160
161     // new ball position
162     assign x_ball_next = (gra_still) ? X_MAX / 2 :
163                          (refresh_tick) ? x_ball_reg + x_delta_reg :
                              ↪  x_ball_reg;
164     assign y_ball_next = (gra_still) ? Y_MAX / 2 :
165                          (refresh_tick) ? y_ball_reg + y_delta_reg :
                              ↪  y_ball_reg;
166
167     // change ball direction after collision
168     always @* begin
169         hit = 1'b0;
170         miss = 1'b0;
171         x_delta_next = x_delta_reg;
172         y_delta_next = y_delta_reg;
173
174         if(gra_still) begin
175             x_delta_next = BALL_VELOCITY_NEG;
176             y_delta_next = BALL_VELOCITY_POS;
177         end
178
179         else if(y_ball_t < T_WALL_B)                    // reach top
180             y_delta_next = BALL_VELOCITY_POS;   // move down
181
182         else if(y_ball_b > (B_WALL_T))          // reach bottom wall
183             y_delta_next = BALL_VELOCITY_NEG;   // move up
184
185         else if(x_ball_l <= L_WALL_R)           // reach left wall
186             x_delta_next = BALL_VELOCITY_POS;   // move right
187
188         else if((X_PAD_L <= x_ball_r) && (x_ball_r <= X_PAD_R) &&
189                 (y_pad_t <= y_ball_b) && (y_ball_t <= y_pad_b)) begin
190                 x_delta_next = BALL_VELOCITY_NEG;
191                 hit = 1'b1;
192         end
193
194         else if(x_ball_r > X_MAX)
195             miss = 1'b1;
196     end
197
198     // output status signal for graphics
```

```
199     assign graph_on = l_wall_on | t_wall_on | b_wall_on | pad_on |
        ↪ ball_on;
200
201
202     // rgb multiplexing circuit
203     always @*
204         if(~video_on)
205             graph_rgb = 12'h000;      // no value, blank
206         else
207             if(l_wall_on | t_wall_on | b_wall_on)
208                 graph_rgb = wall_rgb;     // wall color
209             else if(pad_on)
210                 graph_rgb = pad_rgb;      // paddle color
211             else if(ball_on)
212                 graph_rgb = ball_rgb;     // ball color
213             else
214                 graph_rgb = bg_rgb;       // background
215
216 endmodule
```

4.4 Game Text

The `pong_text` module serves as a text-based graphics controller for a Pong game, responsible for displaying various textual elements on the screen. This module utilizes an ASCII ROM (`ascii_rom`) to map characters to their corresponding ASCII codes, providing a versatile mechanism for generating text.

The module is organized into distinct regions for displaying different types of text: the score region, logo region, rule region, and game over region. These regions are activated based on the current position of the cursor (`x` and `y` coordinates), allowing dynamic updates of the displayed text.

4.4.1 Score Region

The score region displays the player's score and the current ball number in a two-digit format. The ASCII characters for "Score: ", digits, " Ball: ", and the ball number are fetched from the ASCII ROM based on the x-coordinate, and the corresponding RGB color is set. The `score_on` signal activates this region.

4.4.2 Logo Region

The logo region displays the text "CPPONG" at the top center of the screen, serving as a background element. The ASCII characters for "CPPONG" are fetched from the ASCII ROM based on the x-coordinate, and the corresponding RGB color is set. The `logo_on` signal activates this region.

4.4.3 Rule Region

The rule region displays game rules in multiple rows. The ASCII characters for various rules are fetched from the ASCII ROM based on the y and x coordinates.

The corresponding RGB color is set, and the `rule_on` signal activates this region.

4.4.4 Game Over Region

The game over region displays the text "GAME OVER" at the center of the screen. The ASCII characters for "GAME OVER" are fetched from the ASCII ROM based on the x-coordinate, and the corresponding RGB color is set. The `over_on` signal activates this region.

The module outputs `text_on` to indicate which region is currently active and `text_rgb` to provide the RGB color for the displayed text. The `ascii_bit` signal is used to fetch the correct bit from the ASCII ROM, determining the pixel's color. Overall, the `pong_text` module plays a crucial role in rendering informative and visually appealing text elements during the Pong game.

Figure 12: Game Text code

```
`timescale 1ns / 1ps

module pong_text(
    input clk,
    input [1:0] ball,
    input [3:0] dig0, dig1,
    input [9:0] x, y,
    output [3:0] text_on,
    output reg [11:0] text_rgb
    );

    // signal declaration
    wire [10:0] rom_addr;
    reg [6:0] char_addr, char_addr_s, char_addr_l, char_addr_r,
     ↪  char_addr_o;
    reg [3:0] row_addr;
    wire [3:0] row_addr_s, row_addr_l, row_addr_r, row_addr_o;
    reg [2:0] bit_addr;
    wire [2:0] bit_addr_s, bit_addr_l, bit_addr_r, bit_addr_o;
    wire [7:0] ascii_word;
    wire ascii_bit, score_on, logo_on, rule_on, over_on;
    wire [7:0] rule_rom_addr;

   // instantiate ascii rom
   ascii_rom ascii_unit(.clk(clk), .addr(rom_addr), .data(ascii_word));

   //
    ↪  ----------------------------------------------------------------------
   // score region
   // - display two-digit score and ball # on top left
   // - scale to 16 by 32 text size
   // - line 1, 16 chars: "Score: dd Ball: d"
   //
    ↪  ----------------------------------------------------------------------
   assign score_on = (y >= 32) && (y < 64) && (x[9:4] < 16);
   //assign score_on = (y[9:5] == 0) && (x[9:4] < 16);
   assign row_addr_s = y[4:1];
   assign bit_addr_s = x[3:1];
   always @*
    case(x[7:4])
        4'h0 : char_addr_s = 7'h53;     // S 53 43 4F 52 45
        4'h1 : char_addr_s = 7'h43;     // C
```

```
        4'h2 : char_addr_s = 7'h4F;     // O
        4'h3 : char_addr_s = 7'h52;     // R
        4'h4 : char_addr_s = 7'h45;     // E
        4'h5 : char_addr_s = 7'h3A;     // :
        4'h6 : char_addr_s = {3'b011, dig1};    // tens digit
        4'h7 : char_addr_s = {3'b011, dig0};    // ones digit
        4'h8 : char_addr_s = 7'h00;     //
        4'h9 : char_addr_s = 7'h4C;     // L
        4'hA : char_addr_s = 7'h49;     // I
        4'hB : char_addr_s = 7'h56;     // V
        4'hC : char_addr_s = 7'h45;     // E
        4'hD : char_addr_s = 7'h53;     // S
        4'hE : char_addr_s = 7'h3A;     // :
        4'hF : char_addr_s = {5'b01100, ball};
    endcase

// logo region
// - display logo "CPPONG" at top center
// - used as background
// - scale to 64 by 128 text size
//
// ------------------------------------------------------------------------
assign logo_on = (y[9:7] == 2) && (1 <= x[9:6]) && (x[9:6] <= 8); // Adjusted x range
assign row_addr_l = y[6:3];
assign bit_addr_l = x[5:3];
always @*
    case(x[8:6])

        3'o1 : char_addr_l = 7'h00; //
        3'o2 : char_addr_l = 7'h43; // C
        3'o3 : char_addr_l = 7'h50; // P
        3'o4 : char_addr_l = 7'h50; // P
        3'o5 : char_addr_l = 7'h4F; // O
        3'o6 : char_addr_l = 7'h4E; // N
        3'o7 : char_addr_l = 7'h47; // G

        default : char_addr_l = 7'h00; // Empty space
    endcase

    //
    // ------------------------------------------------------------------------
    assign rule_on = (x[9:7] == 2) && (y[9:6] == 2);
    assign row_addr_r = y[3:0];
    assign bit_addr_r = x[2:0];
    assign rule_rom_addr = {y[5:4], x[6:3]};
    always @*
        case(rule_rom_addr)
            // row 1
             // row 1
        6'h00 : char_addr_r = 7'h45;    // E
        6'h01 : char_addr_r = 7'h43;    // C
        6'h02 : char_addr_r = 7'h45;    // E
        6'h03 : char_addr_r = 7'h33;    // 3
        6'h04 : char_addr_r = 7'h33;    // 3
        6'h05 : char_addr_r = 7'h30;    // 0
        6'h06 : char_addr_r = 7'h30;    // 0
        6'h07 : char_addr_r = 7'h00;    // Space
        6'h08 : char_addr_r = 7'h46;    // F
        6'h09 : char_addr_r = 7'h41;    // A
        6'h0A : char_addr_r = 7'h4C;    // L
        6'h0B : char_addr_r = 7'h4C;    // L
        6'h0C : char_addr_r = 7'h00;    //
        6'h0D : char_addr_r = 7'h32;    // 2
        6'h0E : char_addr_r = 7'h33;    // 3
```

```
        6'h0F : char_addr_r = 7'h00;    //
            // row 2
            6'h10 : char_addr_r = 7'h00;    //
            6'h11 : char_addr_r = 7'h46;    // F
            6'h12 : char_addr_r = 7'h49;    // I
            6'h13 : char_addr_r = 7'h4E;    // N
            6'h14 : char_addr_r = 7'h41;    // A
            6'h15 : char_addr_r = 7'h4C;    // L
            6'h16 : char_addr_r = 7'h00;    //
            6'h17 : char_addr_r = 7'h00;    //
            6'h18 : char_addr_r = 7'h50;    // P
            6'h19 : char_addr_r = 7'h52;    // R
            6'h1A : char_addr_r = 7'h4F;    // O
            6'h1B : char_addr_r = 7'h4A;    // J
            6'h1C : char_addr_r = 7'h45;    // E
            6'h1D : char_addr_r = 7'h43;    // C
            6'h1E : char_addr_r = 7'h54;    // T
            6'h1F : char_addr_r = 7'h00;    //
            // row 3
            6'h20 : char_addr_r = 7'h50;    // P
            6'h21 : char_addr_r = 7'h52;    // R
            6'h22 : char_addr_r = 7'h45;    // E
            6'h23 : char_addr_r = 7'h53;    // S
            6'h24 : char_addr_r = 7'h53;    // S
            6'h25 : char_addr_r = 7'h00;    //
            6'h26 : char_addr_r = 7'h45;    // E
            6'h27 : char_addr_r = 7'h4E;    // N
            6'h28 : char_addr_r = 7'h54;    // T
            6'h29 : char_addr_r = 7'h45;    // E
            6'h2A : char_addr_r = 7'h52;    // R
            6'h2B : char_addr_r = 7'h00;    //
            6'h2C : char_addr_r = 7'h00;    //
            6'h2D : char_addr_r = 7'h00;    //
            6'h2E : char_addr_r = 7'h00;    //
            6'h2F : char_addr_r = 7'h00;    //
            // row 4
            6'h30 : char_addr_r = 7'h31;    // 1
            6'h31 : char_addr_r = 7'h00;    //
            6'h32 : char_addr_r = 7'h48;    // H
            6'h33 : char_addr_r = 7'h49;    // I
            6'h34 : char_addr_r = 7'h54;    // T
            6'h35 : char_addr_r = 7'h00;    //
            6'h36 : char_addr_r = 7'h49;    // I
            6'h37 : char_addr_r = 7'h53;    // S
            6'h38 : char_addr_r = 7'h00;    //
            6'h39 : char_addr_r = 7'h33;    // 3
            6'h3A : char_addr_r = 7'h00;    //
            6'h3B : char_addr_r = 7'h50;    // P
            6'h3C : char_addr_r = 7'h54;    // T
            6'h3D : char_addr_r = 7'h53;    // S
            6'h3E : char_addr_r = 7'h00;    //
            6'h3F : char_addr_r = 7'h00;    //
        endcase
    //
     ↪  ------------------------------------------------------------------------
    // game over region
    // - display "GAME OVER" at center
    // - scale to 32 by 64 text size
    //
     ↪  ------------------------------------------------------------------------
    assign over_on = (y[9:6] == 3) && (5 <= x[9:5]) && (x[9:5] <= 13);
    assign row_addr_o = y[5:2];
    assign bit_addr_o = x[4:2];
    always @*
```

```
164         case(x[8:5])
165             4'h5 : char_addr_o = 7'h47;      // G
166             4'h6 : char_addr_o = 7'h41;      // A
167             4'h7 : char_addr_o = 7'h4D;      // M
168             4'h8 : char_addr_o = 7'h45;      // E
169             4'h9 : char_addr_o = 7'h00;      //
170             4'hA : char_addr_o = 7'h4F;      // O
171             4'hB : char_addr_o = 7'h56;      // V
172             4'hC : char_addr_o = 7'h45;      // E
173             default : char_addr_o = 7'h52;   // R
174         endcase
175 
176     // mux for ascii ROM addresses and rgb
177     always @* begin
178         text_rgb = 12'h020;     // green CPP background
179 
180         if(score_on) begin
181             char_addr = char_addr_s;
182             row_addr = row_addr_s;
183             bit_addr = bit_addr_s;
184             if(ascii_bit)
185                 text_rgb = 12'hFF0; // red
186         end
187 
188         else if(rule_on) begin
189             char_addr = char_addr_r;
190             row_addr = row_addr_r;
191             bit_addr = bit_addr_r;
192             if(ascii_bit)
193                 text_rgb = 12'hFF0; // red
194         end
195 
196         else if(logo_on) begin
197             char_addr = char_addr_l;
198             row_addr = row_addr_l;
199             bit_addr = bit_addr_l;
200             if(ascii_bit)
201                 text_rgb = 12'hFF0;// yellow
202         end
203 
204         else begin // game over
205             char_addr = char_addr_o;
206             row_addr = row_addr_o;
207             bit_addr = bit_addr_o;
208             if(ascii_bit)
209                 text_rgb = 12'hF00; // red
210         end
211     end
212 
213     assign text_on = {score_on, logo_on, rule_on, over_on};
214 
215     // ascii ROM interface
216     assign rom_addr = {char_addr, row_addr};
217     assign ascii_bit = ascii_word[~bit_addr];
218 
219 endmodule
```

4.5 VGA Controller

The `vga_controller` module plays a pivotal role in facilitating VGA display output on the Nexys A7-100T FPGA board. Operating within the constraints of a

100MHz system clock, this module intricately orchestrates the generation of a 25MHz pixel tick signal (`w_25MHz`). This pixel tick is essential for delineating the temporal aspects of the display. The module manages two distinct counters, one for horizontal (`h_count_reg`) and another for vertical (`v_count_reg`) positions, synchronizing them with the pixel tick. Through carefully crafted logic, these counters are reset to initiate a new scan line or frame when reaching the designated maximum values, corresponding to the horizontal and vertical dimensions of the display. The generation of horizontal and vertical sync signals (`hsync` and `vsync`) during their respective retrace periods is precisely controlled, adhering to VGA standards. The `video_on` signal indicates whether the current pixel counts fall within the active display area, a crucial factor for determining when to display pixel information. In essence, the `vga_controller` module serves as the temporal conductor, orchestrating the creation of a VGA-compatible signal and laying the groundwork for the graphical rendering capabilities of the connected display.

Figure 13: VGA Controller code

```
`timescale 1ns / 1ps


module vga_controller(
    input clk_100MHz,    // default clock for the Nexys A7-100T
    input reset,         // system reset
    output video_on,     // ON while pixel counts for x and y and within
     ↪  display area
    output hsync,        // horizontal sync
    output vsync,        // vertical sync
    output p_tick,       // the 25MHz pixel/second rate signal, pixel
     ↪  tick
    output [9:0] x,      // pixel count/position of pixel x, max 0-799
    output [9:0] y       // pixel count/position of pixel y, max 0-524
    );

    // Based on VGA standards found at vesa.org for 640x480 resolution
    // Total horizontal width of screen = 800 pixels, partitioned  into
     ↪  sections
    parameter HD = 640;             // horizontal display area width in
     ↪  pixels
    parameter HF = 48;              // horizontal front porch width in
     ↪  pixels
    parameter HB = 16;              // horizontal back porch width in
     ↪  pixels
    parameter HR = 96;              // horizontal retrace width in
     ↪  pixels
    parameter HMAX = HD+HF+HB+HR-1; // max value of horizontal counter
     ↪  = 799
    // Total vertical length of screen = 525 pixels, partitioned into
     ↪  sections
    parameter VD = 480;             // vertical display area length in
     ↪  pixels
    parameter VF = 10;              // vertical front porch length in
     ↪  pixels
    parameter VB = 33;              // vertical back porch length in
     ↪  pixels
    parameter VR = 2;               // vertical retrace length in
     ↪  pixels
    parameter VMAX = VD+VF+VB+VR-1; // max value of vertical counter =
     ↪  524
```

```
28
29      // *** Generate 25MHz from 100MHz
        ↪  ********************************************************
30          reg  [1:0] r_25MHz;
31          wire w_25MHz;
32
33          always @(posedge clk_100MHz or posedge reset)
34                  if(reset)
35                    r_25MHz <= 0;
36                  else
37                    r_25MHz <= r_25MHz + 1;
38
39          assign w_25MHz = (r_25MHz == 0) ? 1 : 0; // assert tick 1/4 of
             ↪  the time
40      //
        ↪  ************************************************************************************
41
42      // Counter Registers, two each for buffering to avoid glitches
43      reg [9:0] h_count_reg, h_count_next;
44      reg [9:0] v_count_reg, v_count_next;
45
46      // Output Buffers
47      reg v_sync_reg, h_sync_reg;
48      wire v_sync_next, h_sync_next;
49
50      // Register Control
51      always @(posedge clk_100MHz or posedge reset)
52          if(reset) begin
53              v_count_reg <= 0;
54              h_count_reg <= 0;
55              v_sync_reg  <= 1'b0;
56              h_sync_reg  <= 1'b0;
57          end
58          else begin
59              v_count_reg <= v_count_next;
60              h_count_reg <= h_count_next;
61              v_sync_reg  <= v_sync_next;
62              h_sync_reg  <= h_sync_next;
63          end
64
65      //Logic for horizontal counter
66      always @(posedge w_25MHz or posedge reset)      // pixel tick
67          if(reset)
68              h_count_next = 0;
69          else
70              if(h_count_reg == HMAX)                 // end of
                 ↪  horizontal scan
71                  h_count_next = 0;
72              else
73                  h_count_next = h_count_reg + 1;
74
75      // Logic for vertical counter
76      always @(posedge w_25MHz or posedge reset)
77          if(reset)
78              v_count_next = 0;
79          else
80              if(h_count_reg == HMAX)                 // end of
                 ↪  horizontal scan
81                  if((v_count_reg == VMAX))           // end of vertical
                     ↪  scan
82                      v_count_next = 0;
83                  else
84                      v_count_next = v_count_reg + 1;
85
86      // h_sync_next asserted within the horizontal retrace area
```

```
87      assign h_sync_next = (h_count_reg >= (HD+HB) && h_count_reg <=
        ↪  (HD+HB+HR-1));
88
89      // v_sync_next asserted within the vertical retrace area
90      assign v_sync_next = (v_count_reg >= (VD+VB) && v_count_reg <=
        ↪  (VD+VB+VR-1));
91
92      // Video ON/OFF - only ON while pixel counts are within the display
        ↪  area
93      assign video_on = (h_count_reg < HD) && (v_count_reg < VD); //
        ↪  0-639 and 0-479 respectively
94
95      // Outputs
96      assign hsync  = h_sync_reg;
97      assign vsync  = v_sync_reg;
98      assign x      = h_count_reg;
99      assign y      = v_count_reg;
100     assign p_tick = w_25MHz;
101
102 endmodule
```

4.6 Clock Divider

The `clock_divider` module serves as a clock divider, taking an input clock signal (`clk_gen_fsys`) and producing an output clock signal (`clk_gen_out`) with a reduced frequency determined by the specified division factor (`clk_gen_factor`). The module includes a reset signal (`clk_gen_rst`) to initialize the clock divider. Internally, it employs a register (`clk_gen_temp`) that increments on each positive edge of the input clock. The division factor is applied by selecting the most significant bit of `clk_gen_temp` as the output clock signal, effectively dividing the frequency. This module is valuable in digital systems where different components operate at distinct clock frequencies or in separate clock domains. For example, setting `clk_gen_factor` to 5 would result in an output clock frequency one-fifth of the input clock frequency, facilitating synchronization in diverse parts of a digital system.

Figure 14: Clock Divider code

```
1   `timescale 1ns / 1ps
2
3
4   module clock_divider #(parameter SIZE = 32)
5   (
6       input clk_gen_fsys,
7       input clk_gen_rst,
8       input [4:0] clk_gen_factor,
9       output clk_gen_out
10      );
11
12      reg [SIZE-1:0] clk_gen_temp;
13
14      initial
15          begin
16              clk_gen_temp = 0;
```

```
        end
    always@(posedge clk_gen_fsys)
        begin:CLOCKGENERATOR
                    clk_gen_temp <= clk_gen_temp +1;
        end
    assign clk_gen_out = clk_gen_temp[clk_gen_factor-1];

endmodule
```

4.7 Seven Segment Display Controller

The `seg7_control` module is designed for controlling a 7-segment display to showcase numerical digits based on the input data provided for eight different digits (`digit1` to `digit8`). The module utilizes internal parameters to define segment patterns for the numerical digits 0 through 9. It consists of a digit select mechanism to cycle through the eight digits sequentially, and a digit timer to regulate the refresh rate of the display. The display refreshes every 4 milliseconds (ms), a duration determined by the 100 MHz clock on the Nexys A7 board.

The module features logic for driving the 8-bit anode output, which is responsible for activating each digit in turn. It utilizes a case statement to set the appropriate anode pattern based on the current digit select value.

Additionally, the module includes logic to control the segments based on the selected digit and the value of each digit. It employs nested case statements to assign the proper segment pattern for each digit, ensuring that the 7-segment display accurately represents the input numerical values. The module supports digits ranging from 0 to 9 and includes segment patterns for hexadecimal values A to F.

Overall, the `seg7_control` module provides a flexible and comprehensive solution for interfacing with a 7-segment display and displaying various numerical values on each digit of the display.

Figure 15: SSD Controller code

```
module seg7_control(
    input clk,
    input reset,
    input [3:0] digit1,  // Rename 'ones' to 'digit1'
    input [3:0] digit2,  // Rename 'tens' to 'digit2'
    input [3:0] digit3,  // Rename 'hundreds' to 'digit3'
    input [3:0] digit4, // Rename 'thousands' to 'digit4'
    input [3:0] digit5,
    input [3:0] digit6,
    input [3:0] digit7,
    input [3:0] digit8,
    output reg [0:6] seg,  // segment pattern 0-9
    output reg [7:0] digit  // digit select signals
);

    // Parameters for segment patterns
```

```
    parameter ZERO  = 7'b000_0001;  // 0
    parameter ONE   = 7'b100_1111;  // 1
    parameter TWO   = 7'b001_0010;  // 2
    parameter THREE = 7'b000_0110;  // 3
    parameter FOUR  = 7'b100_1100;  // 4
    parameter FIVE  = 7'b010_0100;  // 5
    parameter SIX   = 7'b010_0000;  // 6
    parameter SEVEN = 7'b000_1111;  // 7
    parameter EIGHT = 7'b000_0000;  // 8
    parameter NINE  = 7'b000_0100;  // 9

    // To select each digit in turn
    reg [2:0] digit_select;     // 3 bit counter for selecting each of 8 digits
    reg [23:0] digit_timer;     // counter for digit refresh

    // Logic for controlling digit select and digit timer
    always @(posedge clk or posedge reset) begin
        if(reset) begin
            digit_select <= 0;
            digit_timer <= 0;

        end
        else                                        // 1ms x 4 displays = 4ms refresh period
            if(digit_timer == 99_999) begin         // The period of 100MHz clock in NEXYS A7 is 10ns (1/100,000,000 seconds)
                digit_timer <= 0;                   // 10ns x 100,000 = 1ms
                digit_select <=  digit_select + 1;
            end
            else
                digit_timer <=  digit_timer + 1;
    end


    // Logic for driving the 8-bit anode output based on digit select
    always @(digit_select) begin
        case (digit_select)
            3'b000 : digit = 8'b1111_1110;   // Turn on digit1
            3'b001 : digit = 8'b1111_1101;   // Turn on digit2
            3'b010 : digit = 8'b1111_1011;   // Turn on digit3
            3'b011 : digit = 8'b1111_0111;   // Turn on digit4
            3'b100 : digit = 8'b1110_1111;   // Turn on digit5
            3'b101 : digit = 8'b1101_1111;   // Turn on digit6
            3'b110 : digit = 8'b1011_1111;   // Turn on digit7
            3'b111 : digit = 8'b0111_1111;   // Turn on digit8
        endcase
    end

    // Logic for driving segments based on which digit is selected and the value of each digit
    always @*
    case (digit_select)
        3'b000:begin // ONES DIGIT
            case (digit1)
                4'b0000: seg = ZERO;
                4'b0001: seg = ONE;
                4'b0010: seg = TWO;
                4'b0011: seg = THREE;
                4'b0100: seg = FOUR;
                4'b0101: seg = FIVE;
                4'b0110: seg = SIX;
                4'b0111: seg = SEVEN;
```

```
                4'b1000: seg = EIGHT;
                4'b1001: seg = NINE;
                4'hA:seg = 7'b0001000;
                4'hB:seg = 7'b0000011;
                4'hC:seg = 7'b1000110;
                4'hD:seg = 7'b0100001;
                4'hE:seg = 7'b0000110;
                4'hF:seg = 7'b0001110;
                default: seg = ZERO;
            endcase
            end
        3'b001:begin // TENS DIGIT
            case (digit2)
                4'b0000: seg = ZERO;
                4'b0001: seg = ONE;
                4'b0010: seg = TWO;
                4'b0011: seg = THREE;
                4'b0100: seg = FOUR;
                4'b0101: seg = FIVE;
                4'b0110: seg = SIX;
                4'b0111: seg = SEVEN;
                4'b1000: seg = EIGHT;
                4'b1001: seg = NINE;
                4'hA:seg = 7'b0001000;
                4'hB:seg = 7'b0000011;
                4'hC:seg = 7'b1000110;
                4'hD:seg = 7'b0100001;
                4'hE:seg = 7'b0000110;
                4'hF:seg = 7'b0001110;
                default: seg = ZERO;
            endcase
            end
        3'b010:begin // HUNDREDS DIGIT
            case (digit3)
                4'b0000: seg = ZERO;
                4'b0001: seg = ONE;
                4'b0010: seg = TWO;
                4'b0011: seg = THREE;
                4'b0100: seg = FOUR;
                4'b0101: seg = FIVE;
                4'b0110: seg = SIX;
                4'b0111: seg = SEVEN;
                4'b1000: seg = EIGHT;
                4'b1001: seg = NINE;
                4'hA:seg = 7'b0001000;
                4'hB:seg = 7'b0000011;
                4'hC:seg = 7'b1000110;
                4'hD:seg = 7'b0100001;
                4'hE:seg = 7'b0000110;
                4'hF:seg = 7'b0001110;
                default: seg = ZERO;
            endcase
            end
        3'b011:begin // THOUSANDS DIGIT
            case (digit4)
                4'b0000: seg = ZERO;
                4'b0001: seg = ONE;
                4'b0010: seg = TWO;
                4'b0011: seg = THREE;
                4'b0100: seg = FOUR;
                4'b0101: seg = FIVE;
                4'b0110: seg = SIX;
                4'b0111: seg = SEVEN;
                4'b1000: seg = EIGHT;
                4'b1001: seg = NINE;
                4'hA:seg = 7'b0001000;
```

```
                4'hB:seg = 7'b0000011;
                4'hC:seg = 7'b1000110;
                4'hD:seg = 7'b0100001;
                4'hE:seg = 7'b0000110;
                4'hF:seg = 7'b0001110;
                default: seg = ZERO;
            endcase
            end
        3'b100:begin // DIGIT 5
            case (digit5)
                // Handle digit 5 segments here
            4'b0000: seg = ZERO;
                4'b0001: seg = ONE;
                4'b0010: seg = TWO;
                4'b0011: seg = THREE;
                4'b0100: seg = FOUR;
                4'b0101: seg = FIVE;
                4'b0110: seg = SIX;
                4'b0111: seg = SEVEN;
                4'b1000: seg = EIGHT;
                4'b1001: seg = NINE;
                4'hA:seg = 7'b0001000;
                4'hB:seg = 7'b0000011;
                4'hC:seg = 7'b1000110;
                4'hD:seg = 7'b0100001;
                4'hE:seg = 7'b0000110;
                4'hF:seg = 7'b0001110;
                default: seg = ZERO;
            endcase
            end
        3'b101:begin // DIGIT 6
            case (digit6)
                // Handle digit 6 segments here
            4'b0000: seg = ZERO;
                4'b0001: seg = ONE;
                4'b0010: seg = TWO;
                4'b0011: seg = THREE;
                4'b0100: seg = FOUR;
                4'b0101: seg = FIVE;
                4'b0110: seg = SIX;
                4'b0111: seg = SEVEN;
                4'b1000: seg = EIGHT;
                4'b1001: seg = NINE;
                4'hA:seg = 7'b0001000;
                4'hB:seg = 7'b0000011;
                4'hC:seg = 7'b1000110;
                4'hD:seg = 7'b0100001;
                4'hE:seg = 7'b0000110;
                4'hF:seg = 7'b0001110;
                default: seg = ZERO;
            endcase
            end
        3'b110:begin // DIGIT 7
            case (digit7)
                // Handle digit 7 segments here
           4'b0000: seg = ZERO;
                4'b0001: seg = ONE;
                4'b0010: seg = TWO;
                4'b0011: seg = THREE;
                4'b0100: seg = FOUR;
                4'b0101: seg = FIVE;
                4'b0110: seg = SIX;
                4'b0111: seg = SEVEN;
                4'b1000: seg = EIGHT;
                4'b1001: seg = NINE;
                4'hA:seg = 7'b0001000;
```

```
                4'hB:seg = 7'b0000011;
                4'hC:seg = 7'b1000110;
                4'hD:seg = 7'b0100001;
                4'hE:seg = 7'b0000110;
                4'hF:seg = 7'b0001110;
                default: seg = ZERO;
            endcase
            end
        3'b111:begin // DIGIT 8
            case (digit8)
                // Handle digit 8 segments here
           4'b0000: seg = ZERO;
                4'b0001: seg = ONE;
                4'b0010: seg = TWO;
                4'b0011: seg = THREE;
                4'b0100: seg = FOUR;
                4'b0101: seg = FIVE;
                4'b0110: seg = SIX;
                4'b0111: seg = SEVEN;
                4'b1000: seg = EIGHT;
                4'b1001: seg = NINE;
                4'hA:seg = 7'b0001000;
                4'hB:seg = 7'b0000011;
                4'hC:seg = 7'b1000110;
                4'hD:seg = 7'b0100001;
                4'hE:seg = 7'b0000110;
                4'hF:seg = 7'b0001110;
                default: seg = ZERO;
            endcase
            end
    endcase
endmodule
```

5 Top Implementation

The `pong_top` module is the central orchestrator that implements a classic Pong game on an FPGA. It employs a Finite State Machine (FSM) to manage different game states, including "newgame," "play," "newball," and "over." The module interfaces with peripherals such as a PS2 receiver for keyboard input processing and a UART transmitter for serial communication. It integrates a clock divider for deriving timing signals and employs timer and counter modules to manage game timing. The inclusion of a music module allows for audio output, with the `music_toggle` signal controlled by the `music_enable` input. The display is handled through a VGA controller and a 7-segment display controller for visual feedback. Text and graphics are managed by dedicated modules (`pong_text` and `pong_graph`). The design also incorporates input handling mechanisms, including debouncing and processing of button inputs. Additionally, it keeps track of game progress through scoring logic, updating the score and controlling the pace of the game. Overall, the `pong_top` module brings together these components in a well-organized and modular design, providing a cohesive and interactive gaming experience for users interacting with the FPGA-based Pong game. rizing the key components and functionalities of the `pong_top` module:

1. Finite State Machine (FSM):
    - Manages game states: `newgame`, `play`, `newball`, and `over`.
    - Controls transitions based on user input and game events.
2. Peripheral Interfaces:
    - PS2 receiver (`PS2_Receiver`) for keyboard input processing.
    - UART transmitter (`uart_tx`) for serial communication.
3. Clock and Timing:
    - Clock divider (`clock_divider`) for deriving a clock signal for timing purposes.
    - Timer and counter modules (`timer`, `m100_counter`) for game timing.
4. Audio:
    - Music module (`music`) for generating audio output.
    - Music toggle signal (`music_toggle`) controlled by the `music_enable` input.
5. Display:
    - VGA controller (`vga_controller`) for managing VGA output.
    - 7-segment display controller (`seg7_control`) for visual feedback.
6. Text and Graphics:
    - Text module (`pong_text`) for rendering text on the display.
    - Graphics module (`pong_graph`) for rendering game graphics.
7. Input Handling:
    - Keyboard input processing and decoding through the PS2 receiver.
    - Debouncing and processing of button inputs.
8. Scorekeeping and Timing:
    - Scoring logic for tracking game progress and updating the score.
    - Timing mechanisms for controlling game pace.
9. User Interface:
    - Integration with peripherals for player interaction (keyboard, buttons).
    - Generation of visual and audio feedback for a user-friendly interface.
10. Modularity and Organization:

- Well-organized and modular design, leveraging FPGA resources efficiently.
- Coordinated functionality between different components for a cohesive gaming experience.

Figure 16: TOP code

```
`timescale 1ns / 1ps


module pong_top(
    input clk,              // 100MHz
    input reset,            // btnR
    input [1:0] btn_altern,
    input PS2_CLK,
    input PS2_DATA,
    input music_enable,
    input btn_enter,
    input [4:0] clk_factor,
    output [0:6]SEG,
    output [7:0]AN,
    output UART_TXD,
    output music_toggle,
    output hsync,           // to VGA Connector
    output vsync,           // to VGA Connector
    output [11:0] rgb,       // to DAC, to VGA Connector
    output  speaker,
    output [15:0] key_code_led
    );

    // state declarations for 4 states
    parameter newgame = 2'b00;
    parameter play    = 2'b01;
    parameter newball = 2'b10;
    parameter over    = 2'b11;

    //wire [1:0] btn;

    // signal declaration

    reg [1:0] state_reg, state_next;
    wire [9:0] w_x, w_y;
    wire w_vid_on, w_p_tick, graph_on, hit, miss;
    wire [3:0] text_on;
    wire [11:0] graph_rgb, text_rgb;
    reg [11:0] rgb_reg, rgb_next;
    wire [3:0] dig0, dig1;
    reg gra_still, d_inc, d_clr, timer_start;
    wire timer_tick, timer_up;
    reg [1:0] ball_reg, ball_next;
    wire clk_divider;

    reg music_reg;
    reg [1:0] buttons_reg;
    reg enter_reg;

    wire        tready;
    wire        ready;
    wire        tstart;
    reg         start=0;
    reg         CLK50MHZ=0;
    wire [31:0] tbuf;
    reg  [15:0] keycodev=0;
```

```
57      wire [15:0] keycode;
58      wire [ 7:0] tbus;
59      reg  [ 2:0] bcount=0;
60      wire        flag;
61      reg         cn=0;
62
63      always @(posedge(clk))begin
64          CLK50MHZ<=~CLK50MHZ;
65      end
66
67      PS2_Receiver uut (
68          .clk(CLK50MHZ),
69          .kclk(PS2_CLK),
70          .kdata(PS2_DATA),
71          .keycode(keycode),
72          .oflag(flag)
73      );
74
75
76      always@(keycode)
77          if (keycode[7:0] == 8'hf0) begin
78              cn <= 1'b0;
79              bcount <= 3'd0;
80          end else if (keycode[15:8] == 8'hf0) begin
81              cn <= keycode != keycodev;
82              bcount <= 3'd5;
83          end else begin
84              cn <= keycode[7:0] != keycodev[7:0] || keycodev[15:8] ==
                    ↪  8'hf0;
85              bcount <= 3'd2;
86          end
87
88      always@(posedge clk)
89          if (flag == 1'b1 && cn == 1'b1) begin
90              start <= 1'b1;
91              keycodev <= keycode;
92          end else
93              start <= 1'b0;
94
95         assign key_code_led = keycode;
96
97      bin2ascii #(
98          .NBYTES(2)
99      ) conv (
100         .I(keycodev),
101         .O(tbuf)
102     );
103
104     uart_buf_con tx_con (
105         .clk    (clk   ),
106         .bcount (bcount),
107         .tbuf   (tbuf  ),
108         .start  (start ),
109         .ready  (ready ),
110         .tstart (tstart),
111         .tready (tready),
112         .tbus   (tbus  )
113     );
114
115     uart_tx get_tx (
116         .clk    (clk),
117         .start  (tstart),
118         .tbus   (tbus),
119         .tx     (UART_TXD),
120         .ready  (tready)
```

```
    );

    always @* begin
    if(music_enable)
    begin
    music_reg = 1'b1;
    end
    else
    music_reg = 1'b0;
    end
    assign music_toggle = music_reg;



seg7_control display (
        .clk(clk),
        .reset(reset),
        .digit5(tbuf[19:16]),
        .digit6(tbuf[23:20]),
        .digit7(tbuf[27:24]),
        .digit8(tbuf[31:28]),
        .digit1(tbuf[3:0]),
        .digit2(tbuf[7:4]),
        .digit3(tbuf[11:8]),
        .digit4(tbuf[15:12]),
        .seg(SEG),  // Connect the 7-segment outputs to the display
         ↪  here
        .digit(AN)
    );

   clock_divider #(.SIZE(32)) myClock(
        .clk_gen_fsys(clk),
        .clk_gen_rst(reset),
        .clk_gen_factor(clk_factor),
        .clk_gen_out(clk_divider) // output goes into the digits block
        );

    music musicTOP(
        .clk(clk_divider),
        .speaker(speaker)
        );


    // Module Instantiations
    vga_controller vga_unit(
        .clk_100MHz(clk),
        .reset(reset),
        .video_on(w_vid_on),
        .hsync(hsync),
        .vsync(vsync),
        .p_tick(w_p_tick),
        .x(w_x),
        .y(w_y));

    pong_text text_unit(
        .clk(clk),
        .x(w_x),
        .y(w_y),
        .dig0(dig0),
        .dig1(dig1),
        .ball(ball_reg),
        .text_on(text_on),
        .text_rgb(text_rgb));

```

```
185     pong_graph graph_unit(
186         .clk(clk),
187         .reset(reset),
188         .btn(tbuf),
189         .btn_altern(btn_altern),
190         .gra_still(gra_still),
191         .video_on(w_vid_on),
192         .x(w_x),
193         .y(w_y),
194         .hit(hit),
195         .miss(miss),
196         .graph_on(graph_on),
197         .graph_rgb(graph_rgb));
198 
199     // 60 Hz tick when screen is refreshed
200     assign timer_tick = (w_x == 0) && (w_y == 0);
201     timer timer_unit(
202         .clk(clk),
203         .reset(reset),
204         .timer_tick(timer_tick),
205         .timer_start(timer_start),
206         .timer_up(timer_up));
207 
208     m100_counter counter_unit(
209         .clk(clk),
210         .reset(reset),
211         .d_inc(d_inc),
212         .d_clr(d_clr),
213         .dig0(dig0),
214         .dig1(dig1));
215 
216 
217     // FSMD state and registers
218     always @(posedge clk or posedge reset)
219         if(reset) begin
220             state_reg <= newgame;
221             ball_reg <= 0;
222             rgb_reg <= 0;
223         end
224 
225         else begin
226             state_reg <= state_next;
227             ball_reg <= ball_next;
228             if(w_p_tick)
229                 rgb_reg <= rgb_next;
230         end
231 
232     // FSMD next state logic
233     always @* begin
234         gra_still = 1'b1;
235         timer_start = 1'b0;
236         d_inc = 1'b0;
237         d_clr = 1'b0;
238         state_next = state_reg;
239         ball_next = ball_reg;
240 
241         case(state_reg)
242             newgame: begin
243 
244                 ball_next = 2'b11;          // three balls
245                 d_clr = 1'b1;               // clear score
246 
247                 if(keycode != 2'b00) begin      // button pressed
248                     state_next = play;
249                     ball_next = ball_reg - 1;
```

```
                end
            end

            play: begin
                gra_still = 1'b0;   // animated screen

                if(hit)
                    d_inc = 1'b1;   // increment score

                else if(miss) begin
                    if(ball_reg == 0)
                        state_next = over;

                    else
                        state_next = newball;

                    timer_start = 1'b1;     // 2 sec timer
                    ball_next = ball_reg - 1;
                end
            end

            newball: // wait for 2 sec and until button pressed
            if(timer_up && (tbuf == 32'h46303541 || btn_enter))
                state_next = play;

            over:   // wait 2 sec to display game over
                if(timer_up && ( tbuf == 32'h46303541 || btn_enter))
                    state_next = newgame;
        endcase
    end

    // rgb multiplexing
    always @*
        if(~w_vid_on)
            rgb_next = 12'h000; // blank

        else
            if(text_on[3] || ((state_reg == newgame) && text_on[1]) ||
             ↪  ((state_reg == over) && text_on[0]))
                rgb_next = text_rgb;    // colors in pong_text

            else if(graph_on)
                rgb_next = graph_rgb;   // colors in graph_text

            else if(text_on[2])
                rgb_next = text_rgb;    // colors in pong_text

            else
                rgb_next = 12'h222;     // grey background

    // output
    assign rgb = rgb_reg;

endmodule
```

6 Simulation

This Verilog testbench includes modules for testing various aspects of the design, such as the VGA controller, Pong game logic, music generation, and keyboard

input processing.

The testbench begins by declaring signals and modules necessary for testing, including clock signals (`clk` and `kclk`), buttons (`reset_btn`, `btn_controller`, `enter_btn`), switches (`clk_factor`, `music_enable_sw`), and internal signals for music testing (`tone`, `fullnote`, `octave`, `note`). It interfaces with the Pong game module and associated components like SSDs, VGA controller, Pong text, Pong graph, timer, and counters.

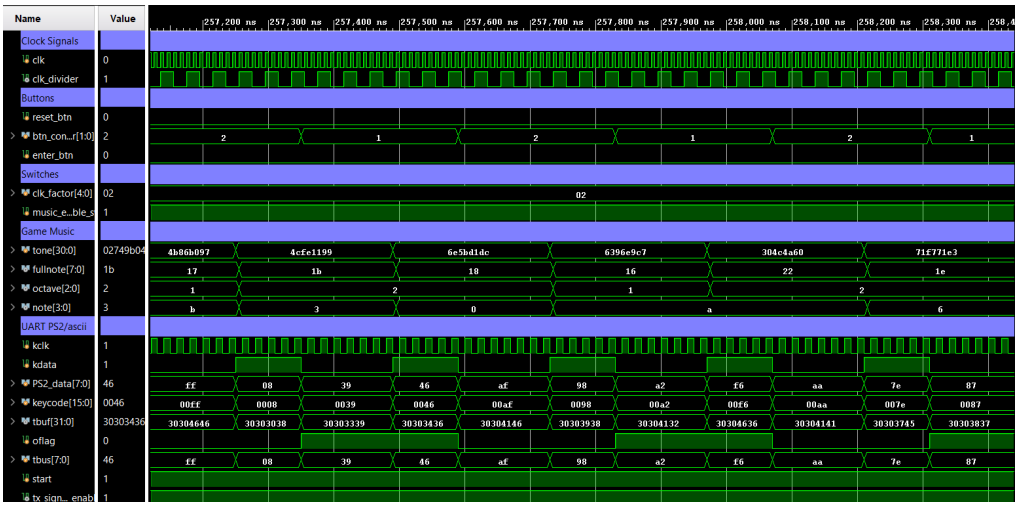
The clock generation is handled with an initial block, ensuring the clock toggles every 5 simulation time units. PS2 clock simulation is also included to stimulate keyboard input.

The initial stimulus block initializes various signals, simulating button presses, music notes, and keyboard inputs. It triggers the Pong game and graph, simulating ball movements and button presses. It also simulates PS2 data transmissions and tests the music module by generating random tones.

The testbench encapsulates an endless loop to continually test and monitor the Pong game. It generates random PS2 data, simulates key presses, and triggers music notes to assess the responsiveness and functionality of the implemented design. The simulation concludes after a specified duration.

This comprehensive testbench provides an effective means to verify the correct functionality and interaction of the Pong game and its associated components, aiding in the identification and resolution of any potential issues during the simulation phase.

Figure 17: Testbench Simulation



7 Demo

The upcoming live demonstration for this school project is an exciting opportunity to showcase the practical application of the designed hardware. Connecting the board to an actual screen will serve as a visual representation of its functionality, allowing for a hands-on exploration of the Pong game in action.

During the demonstration, we will focus on the hardware's seamless integration with the screen, highlighting how the VGA interface translates digital signals into a dynamic and interactive gaming experience. This direct connection will emphasize the practicality of the board, demonstrating its ability to deliver a real-time graphical display on an external screen.

Figure 18: Demo using the Nexys A7-100T board

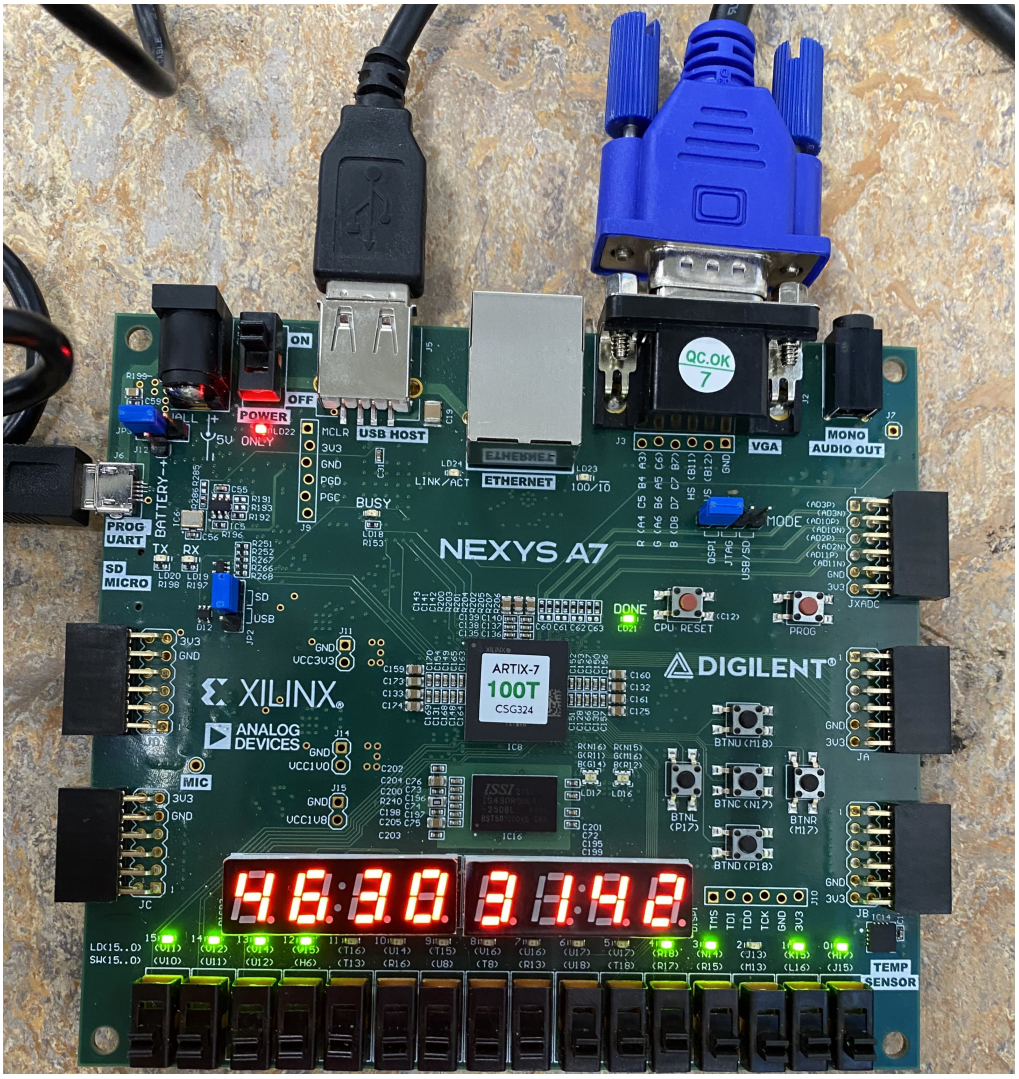


Figure 19: Main Screen

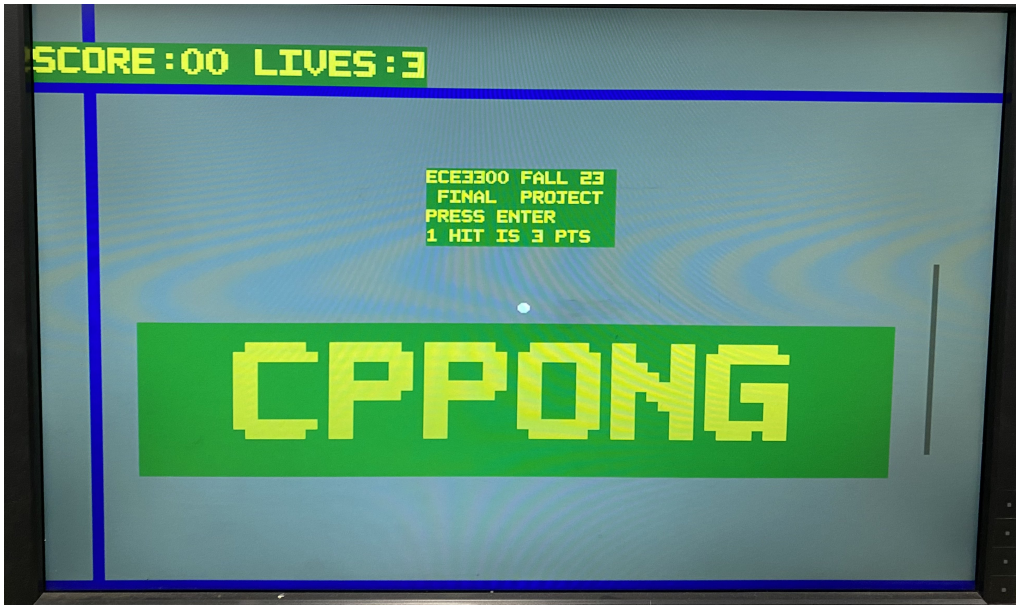


Figure 20: Gameplay Screen

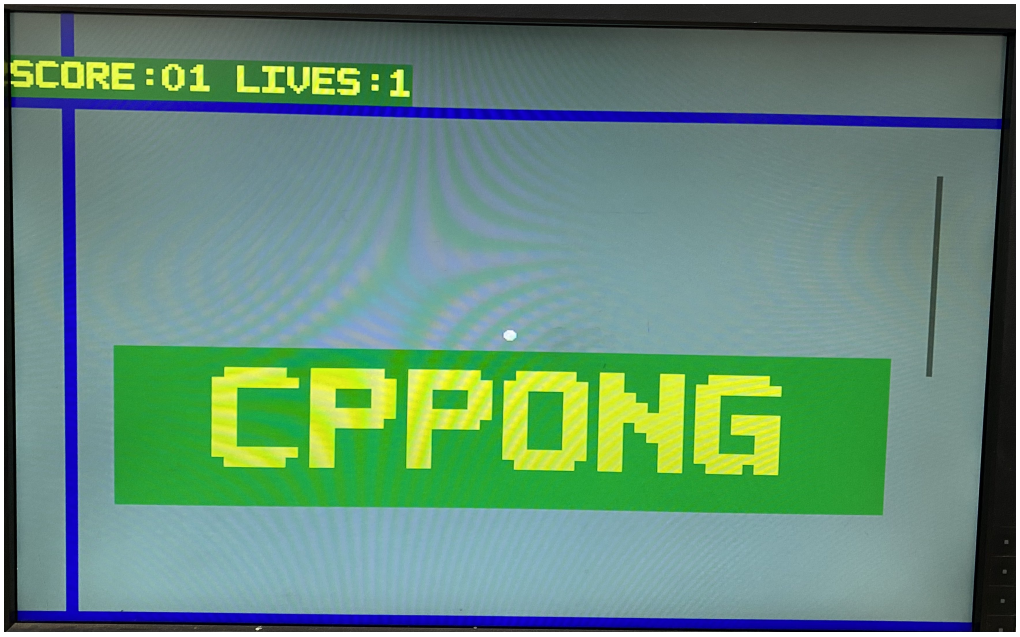
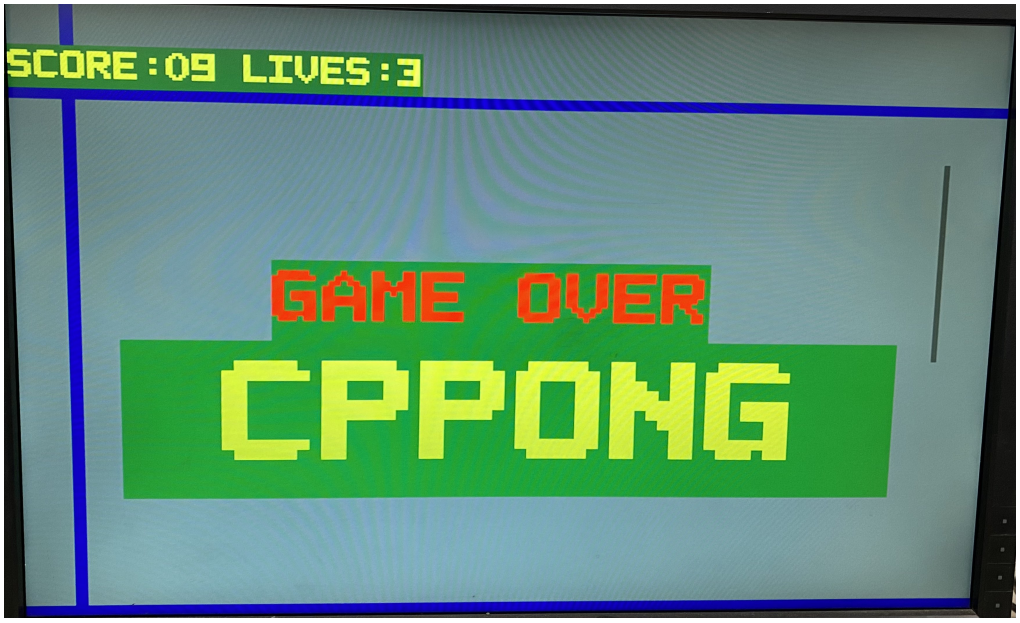


Figure 21: Game Over Screen



8 Discussion

The implementation of the Pong game posed several challenges, chief among them being the adaptation of various control inputs to the game logic and the comprehensive testing of each module. Integrating different control inputs, such as keyboard inputs and button presses, required careful consideration of synchronization and event handling to ensure seamless interaction with the game. The need to harmonize these diverse inputs within the `pong_top` module demanded a meticulous design approach, addressing potential conflicts and ensuring consistent user experiences across different input methods.

Moreover, the testing phase presented its own set of challenges. The intricate nature of the game, involving multiple modules like the VGA controller, Pong text, Pong graph, and music generation, necessitated a thorough and systematic testing strategy. Each module had to be individually validated for correctness and functionality, and then collectively assessed for overall system performance. Coordinating these tests and ensuring that each module operated harmoniously with others was a demanding yet essential task. The iterative testing process, particularly within the testbench, became pivotal in uncovering and rectifying potential issues, enhancing the robustness of the final design.

Despite these challenges, the successful adaptation of diverse control inputs and the meticulous testing procedures have resulted in a cohesive and functional Pong game. The difficulties encountered during this process have not only strengthened the design but also enriched the learning experience, providing valuable insights into handling complexity and ensuring the reliability of intricate hardware systems.

9 Conclusions

In conclusion, the development and simulation of the Pong game and its associated components have yielded a functional and enjoyable gaming experience. The successful integration of the VGA controller, Pong text, Pong graph, keyboard input processing, and music generation modules showcases the design's robustness and versatility. The testbench has effectively validated the system's responsiveness to user input. Future enhancements may explore additional features, optimizations, or expansions of the game logic. This project has not only provided valuable insights into Verilog hardware design and simulation but also offers a solid foundation for the development of more complex and feature-rich games or interactive applications. In summary, the Pong game implementation has proven to be a successful demonstration of hardware design principles, paving the way for further exploration and refinement in future projects.

References

GitHub - Digilent/Nexys-A7-100T-Keyboard — github.com (n.d.). https://github.com/Digilent/Nexys-A7-100T-Keyboard. [Accessed 11-12-2023].

Digital-Design/FPGA Projects/VGA Projects/Pong pt2 at main · FPGADude/Digital-Design — github.com (n.d.). https://github.com/FPGADude. [Accessed 11-12-2023].